
Tutorial: Creating an LLVM Backend for the Cpu0 Architecture

Release 12.0.19.4

Chen Chung-Shu

Jan 19, 2026

CONTENTS

1	About	3
2	Cpu0 Architecture and LLVM Structure	17
3	Backend structure	87
4	Arithmetic and Logic Instructions	211
5	Generating object files	253
6	Global Variables	285
7	Other data type	319
8	Control flow statements	345
9	Function call	397
10	ELF Support	509
11	Assembler	537
12	C++ support	595
13	Verify backend on Verilog simulator	637
14	The Concept of GPU Compiler	663
15	Appendix A: Getting Started: Installing LLVM and the Cpu0 Example Code	791
16	Appendix B: Cpu0 document and test	797
17	Appendix C: The concept of NPU (Neural Processor Unit) compiler	815
18	Todo List	827
19	Resources	829

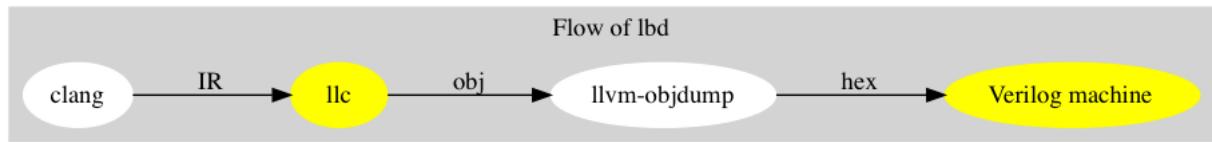


Fig. 1: This book's flow

ABOUT

- *Authors*
- *Contributors*
- *Acknowledgments*
- *Support*
- *Build steps*
- *Revision history*
- *Licensing*
- *Motivation*
- *Preface*
- *Prerequisites*
- *Outline of Chapters*

1.1 Authors

Chen Chung-Shu

gamma_chen@yahoo.com.tw

陳鍾樞

<http://jonathan2251.github.io/ws/en/index.html>

1.2 Contributors

Anoushe Jamshidi, ajamshidi@gmail.com, Chapters 1, 2, 3 English re-writing and Sphinx tool and format setting.

Chen Wei-Ren, chenwj@iis.sinica.edu.tw, assisted with text and code formatting.

Chen Zhong-Cheng, who is the author of original cpu0 verilog code.

1.3 Acknowledgments

We would like to thank Sean Silva, chisophugis@gmail.com, for his help, encouragement, and assistance with the Sphinx document generator. Without his help, this book would not have been finished and published online. We also thank those corrections from readers who make the book more accurate.

1.4 Support

We get the kind help from LLVM development mail list, llvmdev@cs.uiuc.edu, even we don't know them. So, our experience is you are not alone and can get help from the development list members in working with the LLVM project. Some of them are:

Akira Hatanaka <ahatanak@gmail.com> in va_arg question answer.

Ulrich Weigand <Ulrich.Weigand@de.ibm.com> in AsmParser question answer.

1.5 Build steps

<https://github.com/Jonathan2251/lbd/blob/master/README.md>

1.6 Revision history

Version 12.0.20, not released yet.

Version 12.0.19.4, January 19, 2026.

gpu.rst: add ‘section Mobile GPU 3D Rendering’, refine ‘section Mesh-Shader Pipeline’. llvmstructure.rst: refine ‘section Clang’.

Version 12.0.19.3, January 4, 2026.

gpu.rst: Add pc-lcd.png, in-out-rendering.gv, ‘section Background of GLSL (GL Shader Language)’, Table ‘Animation Types’. Refine ‘section Graphics HW and SW Stack’, refine ‘section 3D Rendering’, expand ‘section 3D Rendering’.

Version 12.0.19.2, December 28, 2025.

gpu.rst: Add subsections ‘Animation’ and ‘Node-Editor (shaders generator)’. Modify table: ‘OpenGL rendering pipeline from page 10 and page 36 of book “OpenGL Programming Guide 9th Edition”’. Add tables ‘Data Flow Through the OpenGL Shader Pipeline’ and ‘Examples of Common Varyings’. ctrlflow.rst: Add sections ‘Phi In Optimization’ and ‘LLVM IR φ-Node Optimization Examples’.

Version 12.0.19.1, November 30, 2025.

gpu.rst: thread state, configuration for SM. Correct section reference. Remove useless figures.

Version 12.0.19, November 2, 2025.

gpu.rst: l1-l2.png, threadblocks-map.png

Version 12.0.18, October 4, 2025.

gpu.rst: add subsection ‘Processor Units and Memory Hierarchy in NVIDIA GPU’, ‘Geometry Units’, ‘Rasterization Units’ and ‘Render Output Units (ROPs)’.

Version 12.0.17.3, Released September 26, 2025.

gpu.rst: gpu-hw.gv, ogl-pipeline-hw.gv, add subsection ‘Texture Mapping Units (TMUs)’ and add section ‘Unified IR Conversion Flows’.

Version 12.0.17.2, Released September 20, 2025.

gpu.rst: add subsection ‘RegLess-style architectures’.

Version 12.0.17.1, Released September 14, 2025.

gpu.rst: add ‘SISD, SIMD, and SPMD Pipelines’ in subsection SM (SIMT). backendstructure.rst: enc.gv.

Version 12.0.17, Released September 7, 2025.

c++.rst: Rewrite section ‘C++ Memory Order’. llvmstructure.rst: Refine the order of sections. gpu.rst: Refine subsection ‘GPU Hardware Units’.

Version 12.0.16.3, Released Augest 10, 2025.

llvmstructure.rst: Modify subsection ‘Why doesn’t the Clang compiler use YACC/LEX tools to parse C++?’
. gpu.rst: Modify subsection ‘SIMT’.

Version 12.0.16.2, Released July 22, 2025.

c++.rst: Rewrite section ‘C++ Memory Order’.

Version 12.0.16.1, Released July 20, 2025.

llvmstructure.rst: add section “BNF Auto-Generated Parsers vs. Handwritten Parsers”. c++.rst: Rewrite section ‘C++ Memory Order’. gpu.rst: section ‘OpenCL, Vulkan and spir-v’, subsections ‘Buffers’. add ‘Descrete GPU’, ‘Goals’, ‘GLSL vs. C: Feature Overview’ and ‘GLSL Qualifiers by Shader Stage’ in sub-section ‘GLSL (GL Shader Language)’.

Version 12.0.16, Released May 24, 2025.

Chatgpt: Refine my English for the input reStructuredText (reST) format I provide below, and output the corrected version in reST format with each line no longer than 80 characters.

Version 12.0.15.4, Released September 22, 2024.

backendstructure.rst: Fix Cpu0.td for CH2, Cpu0MCInstLower::Lower() and printAsm.gv, callFunctions.gv, cpu0-function.gv; elf.rst: ch3.disas.log, table to explain decodeInstruction();

Version 12.0.15.3, Released September 15, 2024.

gpu.rst: address coalescing, subgroup and trans-steps.png; backendstructure.rst: printAsm.gv, asm-emit.gv and llvm-data-structure.gv; elf.rst: disas.gv; asm.rst: asmFlow.gv; genobj.rst: getMemEncoding, obj-emit.gv;

Version 12.0.15.2, Released August 18, 2024.

gpu.rst: sm.png, gpu-block-diagram.png, spirv-lang-layers.gv, sycl.png, subgroup, threadblock.jpg, threads-lanes.png, grid.png.

Version 12.0.15.1, Released May 31, 2024.

gpu.rst: opengl-flow.gv, explanation of Vota, rotation, section Transformation, graphic-gpu-csf.png. llvm-structure.rst: DAG for two destination registers.

Version 12.0.15, Released March 16, 2024.

llvmstructure.rst: table: C, llvm-ir and Cpu0. gpu.rst: glsl-spirv.gv, AMD FreeSync, Fragment definition, comment for References/01-triangles.cpp, Refine sections and sub-sections.

Version 12.0.14, Released December 30, 2023.

gpu.rst: Table More Descriptive Name and Cuda term in GPU and Description, Table Mapping saxpy code.
gpu.rst: Table More Descriptive Name for Cuda term in Fermi GPU and Description. llvmstructure.rst: table: C, llvm-ir and Cpu0. llvmstructure.rst: note What and how the ISA of Cpu0 be selected

Version 12.0.13.9, Released November 30, 2023.

gpu.rst: Table More Descriptive Name and Cuda term in GPU and Description, Table Mapping saxpy code.

Version 12.0.13.8, Released November 30, 2023.

gpu.rst: graphic_cpu_gpu.png, additive-colors.

Version 12.0.13.7, Released November 16, 2023.

gpu.rst: 2d-vector-inward, ogl-pointing-outwards and short-rendering-pipeline, Double Buffering and VSync.

Version 12.0.13.6, Released November 13, 2023.

gpu.rst: VAO.

Version 12.0.13.5, Released September 10, 2023.

gpu.rst: Tessellation Shading and clearpage for pdf. llvmstructure.rst: section CFG and Register Allocation Passes.

Version 12.0.13.4, Released August 26, 2023.

llvm.rst: Sections of Options of llc for debug and Options of opt.

Version 12.0.13.3, Released August 13, 2023.

gpu.rst: animation, graphic-sw-stack.gv and opengl-flow.gv.

Version 12.0.13.2, Released August 7, 2023.

gpu.rst: Subsection of buffers, vao binding.

Version 12.0.13.1, Released July 24, 2023.

gpu.rst: Section of Basic geometry in computer graphics, a x b = -b x a in 2D, The role of GPU driver.

npu.rst: The role of GPU driver.

Version 12.0.13, Released July 15, 2023.

gpu.rst: Section of Basic geometry in computer graphics

Version 12.0.12, Released April 4, 2023.

gpu.rst: refine.

Version 12.0.11, Released February 27, 2023.

README.md. docs.rst: Note of Sphinx. c++.rst: Atomic.

Version 12.0.10, Released December 15, 2022.

gpu.rst: Refine Table 43. ctrl.rst: Refine section “Pipeline architecture”. Change test_memcpy.ll. Refine install.rst.

Version 12.0.9, Released November 19, 2022.

gpu.rst: Table 42 Map (Core,Thread) to saxpy and refine section of General purpose GPU. Move null_pointer.cpp from git/note to lbd/References.

Version 12.0.8, Released November 12, 2022.

install.rst: section Toolchain and Brew install in China. Section Work flow of genobj.rst. set-llvm-lit.

Version 12.0.7, Released September 24, 2022.

Atomic, section of Accelerate ML/DL on OpenCL/SYC and refine Makefile and install.rst

Version 12.0.6, Released August 16, 2022.

Fig/backendstructure/class_access_link.puml. Lock-free of chapter c++ and Vulkan link of gpu. Install & doc. Update spirvtoolchain link and grid.png in gpu chapter.

Version 12.0.5, Released February 1, 2022.

Fix regression test.

Version 12.0.4, Released January 22, 2022.

Fix bug: add CMPu, store uses GPROut register to exclude SW registe and Relocation Record: R_CPU0_HI16/fixup_Cpu0_HI16.

Version 12.0.3, Released January 9, 2022.

Expand memory size of cpu0.v to 0x1000000, 24-bit. Section LLVM vs GCC in structure. Add NOR instruction. Fix bug of SLTu SLTiU, SRA and SRAV in verilog code.

Version 12.0.2, Released December 18, 2021.

Remove regression test cases for large frame of not supporting.

Version 12.0.1, Released December 12, 2021.

Section: More about llvm. Table: The differences for speedup in architecture of CPU and GPU. Pipeline diagram and exception handling link. Update chapter Appendix A.

Version 12.0.0, Released August 11, 2021.

Writing and comment.

Version 3.9.4, Released August 5, 2021.

Writing and comment.

Version 3.9.3, Released March 1, 2020.

Add Appendix C: GPU compiler

Version 3.9.2, Released Feburary 17, 2020.

Add section “Add specific backend intrinsic function”. Add reasons for regression test. More phi node explanation.

Version 3.9.1, Released May 11, 2018

Fix tailcall bug. Fix return-vector.ll run slowly problem, bug from Cpu0ISelLowering.cpp. Add figure “Tblgen generate files for Cpu0 backend”. Modify section float and double of Chapter Other data type. Move storeRegToStack() and loadRegFromStack() from Chapter9_1 to Chapter3_5. Section DSA of chapter Cpu0 architecture and LLVM structure.

Version 3.9.0, Released November 22, 2016

Porting to llvm 3.9. Correct writing.

Version 3.7.4, Released December 7, 2016

Change bal instruction from with delay slot to without delay slot.

Version 3.7.3, Released July 20, 2016

Refine code-block according sphinx lexers. Add search this book.

Version 3.7.2, Released June 29, 2016

Add Verilog delay slot simulation. Explain “tablegen(” in CMakeLists.txt. Correct typing. Add lib-ex/install_llvm/*.sh for installation. Upgrade sphinx to 1.4.4.

Version 3.7.1, Released November 7, 2015

Remove EM_CPU0_EL. Add subsection Caller and callee saved registers. Add IR blockaddress and indirectbr support. Correct tglobaladdr, tblockaddress, tjmphtable and tglobalsaddr of Cpu0InstrInfo.td. Add stacksave and stackrestore support. Add sub-section frameaddress, returnaddress and eh.return support of chapter Function call. Match Mips 3.7 style. Add bswap in Chapter Function call. Add section “Vector type (SIMD) support” of Chapter “Other data type”. Add section “Long branch support” of Chapter “Control flow statements”. Add sub-section “eh.dwarf intrinsic” of Chapter Function call. Change display “ret \$rx” to “jr \$rx” where \$rx is not \$lr. Move sub-section Caller and callee saved registers. Add sub-sections Live

in and live out register. Add Phi node. Replace ch3-proepilog.ll with ch3_largeframe.cpp. Remove DecodeCMPIInstruction(). Re-organize testing ch4_2_1.cpp, ch4_2_2.cpp and ch9_4.cpp. Fix dynamic alloca bug. Move Cpu0AnalyzeImmediate.cpp and related functions from Chapter3_4 to Chapter3_5. Rename input files.

Version 3.7.0, Released September 24, 2015

Porting to lld 3.7. Change tricore_llvm.pdf web link. Add C++ atomic to regression test.

Version 3.6.4, Released July 15, 2015

Add C++ atomic support.

Version 3.6.3, Released May 25, 2015

Correct typing.

Version 3.6.2, Released May 3, 2015

Write Appendix B. Split chapter Appendix B from Appendix A. Move some test from lbt to lbd. Remove warning in build Cpu0 code.

Version 3.6.1, Released March 22, 2015

Add Cpu0 instructions ROLV and RORV.

Version 3.6.0, Released March 9, 2015

Update Appendix A for llvm 3.6. Replace cpp with ll for appearing in document. Move chapter lld, optimization, library to <https://github.com/Jonathan2251/lbt.git>.

Version 3.5.9, Released February 2, 2015

Fix bug of 64 bits shift. Fix global address error by replacing addiu with ori. Change encode of “cmp \$sw, \$3, \$2” from 0x10320000 to 0x10f32000.

Version 3.5.8, Released December 27, 2014

Correct typing. Fix typing error for update lbdex/src/modify/src/ of install.rst. Add libsoftfloat/compiler-rt and libc/avr-libc-1.8.1. Add LLVM-VPO in chapter Optimization.

Version 3.5.7, Released December 1, 2014

Fix over 16-bits frame prologue/epilogue error from 3.5.3. Call convention ABI S32 is enabled by option. Change from ADD to ADDu in copyPhysReg() of Cpu0SEInstrInfo.cpp. Add asm directive .weak back which exists in 3.5.3.

Version 3.5.6, Released November 18, 2014

Remove SWI and IRET instructions. Add Cpu0SetChapter.h for ex-build-test.sh. Correct typing. Fix thread variable error come from version 3.5.3 in static mode. Add sub-section “Cpu0 backend machine ID and relocation records” of Chapter 2.

Version 3.5.5, Released November 11, 2014

Rename SPR to C0R. Add ISR simulation.

Version 3.5.4, Released November 6, 2014

Adjust chapter 9 sections. Fix .cprestore bug. Re-organize sections. Add sub-section “Why not using ADD instead of SUB?” in chapter 2. Add overflow control option to use ADD and SUB instructions.

Version 3.5.3, Released October 29, 2014

Merge Cpu0 example code into one copy and it can be config by Cpu0Config.h.

Version 3.5.2, Released October 3, 2014

Move R_CPU0_32 from type of non-relocation record to type of relocation record. Correct logic error for setgt of BrcondPatsSlt of Cpu0InstrInfo.td.

Version 3.5.1, Released October 1, 2014

Add move alias instruction for addu \$reg, \$zero. Add cpu cycles count in verilog. Fix ISD::SIGN_EXTEND_INREG error in other types beside i1. Support DAG op br_jt and DAG node JumpTable.

Version 3.5.0, Released September 05, 2014

Issue NOP in delay slot.

Version 3.4.8, Released August 29, 2014

Add reason that set endian swap in memory module. Add presentation files.

Version 3.4.7, Released August 22, 2014

Fix wrapper_pic for cmov.ll. Add shift operations 64 bits support. Fix wrapper_pic for ch8_5.cpp. Add section thread of chapter 14. Add section Motivation of chapter about. Support little endian for cpu0 verilog. Move ch8_5.cpp test from Chapter Run backend to Chapter lld since it need lld linker. Support both big endian and little endian in cpu0 Verilog, elf2hex and lld. Make branch release_34_7.

Version 3.4.6, Released July 26, 2014

Add Chapter 15, optimization. Correct typing. Add Chapter 14, C++. Fix bug of generating cpu032II instruction in dynamic_linker.cpp.

Version 3.4.5, Released June 30, 2014

Correct typing.

Version 3.4.4, Released June 24, 2014

Correct typing. Add the reason of use SSA form. Move sections LLVM Code Generation Sequence, DAG and Instruction Selection from Chapter 3 to Chapter 2.

Version 3.4.3, Released March 31, 2014

Fix Disassembly bug for GPROut register class. Adjust Chapters. Remove hand copy Table of tblgen in AsmParser.

Version 3.4.2, Released February 9, 2014

Add ch12_2.cpp for slt instruction explanation and fix bug in Cpu0InstrInfo.cpp. Correct typing. Move Cpu0 Status Register from Number 20 to Number 10. Fix llc -mcpu option problem. Update example code build shell script. Add condition move instruction. Fix bug of branch pattern match in Cpu0InstrInfo.td.

Version 3.4.1, Released January 18, 2014

Add ch9_4.cpp to lld test. Fix the wrong reference in lbd/lib/Target/Cpu0 code. inlineasm. First instruction jmp X, where X changed from _Z5startv to start. Correct typing.

Version 3.4.0, Released January 9, 2014

Porting to llvm 3.4 release.

Version 3.3.14, Released January 4, 2014

lld support on iMac. Correct typing.

Version 3.3.13, Released December 27, 2013

Update section Install sphinx on install.rst. Add Fig/llvmstructure/cpu0_arch.odp.

Version 3.3.12, Released December 25, 2013

Correct typing error. Adjust Example Code. Add section Data operands DAGs of backendstructure.rst. Fix bug in instructions lb and lh of cpu0.v. Fix bug in itoa.cpp. Add ch7_2_2.cpp for othertype.rst. Add AsmParser reference web.

Version 3.3.11, Released December 11, 2013

Add Figure Code generation and execution flow in about.rst. Update backendstructure.rst. Correct otherinst.rst. Decoration. Correct typing error.

Version 3.3.10, Released December 5, 2013

Correct typing error. Dynamic linker in lld.rst. Correct errors came from old version of example code. lld.rst.

Version 3.3.9, Released November 22, 2013

Add LLD introduction and Cpu0 static linker document in lld.rst. Fix the plt bug in elf2hex.h for dynamic linker.

Version 3.3.8, Released November 19, 2013

Fix the reference file missing for make gh-page.

Version 3.3.7, Released November 17, 2013

lld.rst documentation. Add cpu032I and cpu032II in *llc -mcpu*. Reference only for Chapter12_2.

Version 3.3.6, Released November 8, 2013

Move example code from github to dropbox since the name is not work for download example code.

Version 3.3.5, Released November 7, 2013

Split the elf2hex code from modified llvm-objdump.cpp to elf2hex.h. Fix bug for tail call setting in LowerCall(). Fix bug for LowerCPLOAD(). Update elf.rst. Fix typing error. Add dynamic linker support. Merge cpu0 Chapter12_1 and Chapter12_2 code into one, and identify each of them by -mcpu=cpu0I and -mcpu=cpu0II. Update lld.rst for static linker. Change the name of example code from LLVMBackendTutorialExampleCode to lbdex.

Version 3.3.4, Released September 21, 2013

Fix Chapter Global variables error for LUI instructions and the material move to Chapter Other data type. Update regression test items.

Version 3.3.3, Released September 20, 2013

Add Chapter othertype

Version 3.3.2, Released September 17, 2013

Update example code. Fix bug sext_inreg. Fix llvm-objdump.cpp bug to support global variable of .data. Update install.rst to run on llvm 3.3.

Version 3.3.1, Released September 14, 2013

Add load bool type in chapter 6. Fix chapter 4 error. Add interrupt function in cpu0i.v. Fix bug in alloc() support of Chapter 8 by adding code of spill \$fp register. Add JSUB texternalsym for memcpy function call of llvm auto reference. Rename cpu0i.v to cpu0s.v. Modify itoa.cpp. Cpu0 of lld.

Version 3.3.0, Released July 13, 2013

Add Table: C operator ! corresponding IR of .bc and IR of DAG and Table: C operator ! corresponding IR of Type-legalized selection DAG and Cpu0 instructions. Add explanation in section Full support %. Add Table: Chapter 4 operators. Add Table: Chapter 3 .bc IR instructions. Rewrite Chapter 5 Global variables. Rewrite section Handle \$gp register in PIC addressing mode. Add Large Frame Stack Pointer support. Add

dynamic link section in elf.rst. Re-organize Chapter 3. Re-organize Chapter 8. Re-organize Chapter 10. Re-organize Chapter 11. Re-organize Chapter 12. Fix bug that ret not \$lr register. Porting to LLVM 3.3.

Version 3.2.15, Released June 12, 2013

Porting to llvm 3.3. Rewrite section Support arithmetic instructions of chapter Adding arithmetic and local pointer support with the table adding. Add two sentences in Preface. Add *llc -debug-pass* in section LLVM Code Generation Sequence. Remove section Adjust cpu0 instructions. Remove section Use cpu0 official LDI instead of ADDiu of Appendix-C.

Version 3.2.14, Released May 24, 2013

Fix example code disappeared error.

Version 3.2.13, Released May 23, 2013

Add sub-section “Setup llvm-lit on iMac” of Appendix A. Replace some code-block with literalinclude in *.rst. Add Fig 9 of chapter Backend structure. Add section Dynamic stack allocation support of chapter Function call. Fix bug of Cpu0DelUselessJMP.cpp. Fix cpu0 instruction table errors.

Version 3.2.12, Released March 9, 2013

Add section “Type of char and short int” of chapter “Global variables, structs and arrays, other type”.

Version 3.2.11, Released March 8, 2013

Fix bug in generate elf of chapter “Backend Optimization”.

Version 3.2.10, Released February 23, 2013

Add chapter “Backend Optimization”.

Version 3.2.9, Released February 20, 2013

Correct the “Variable number of arguments” such as sum_i(int amount, ...) errors.

Version 3.2.8, Released February 20, 2013

Add section llvm-objdump -t -r.

Version 3.2.7, Released February 14, 2013

Add chapter Run backend. Add Icarus Verilog tool installation in Appendix A.

Version 3.2.6, Released February 4, 2013

Update CMP instruction implementation. Add llvm-objdump section.

Version 3.2.5, Released January 27, 2013

Add “LLVMBBackendTutorialExampleCode/llvm3.1”. Add section “Structure type support”. Change reference from Figure title to Figure number.

Version 3.2.4, Released January 17, 2013

Update for LLVM 3.2. Change title (book name) from “Write An LLVM Backend Tutorial For Cpu0” to “Tutorial: Creating an LLVM Backend for the Cpu0 Architecture”.

Version 3.2.3, Released January 12, 2013

Add chapter “Porting to LLVM 3.2”.

Version 3.2.2, Released January 10, 2013

Add section “Full support %” and section “Verify DIV for operator %”.

Version 3.2.1, Released January 7, 2013

Add Footnote for references. Reorganize chapters (Move bottom part of chapter “Global variable” to chapter “Other instruction”; Move section “Translate into obj file” to new chapter “Generate obj file”. Fix errors in Fig/otherinst/2.png and Fig/otherinst/3.png.

Version 3.2.0, Released January 1, 2013

Add chapter Function. Move Chapter “Installing LLVM and the Cpu0 example code” from beginning to Appendix A. Add subsection “Install other tools on Linux”. Add chapter ELF.

Version 3.1.2, Released December 15, 2012

Fix section 6.1 error by add “def : Pat<(brcond RC:\$cond, bb:\$dst), (JNEOp (CMPOp RC:\$cond, ZEROReg), bb:\$dst)>;” in last pattern. Modify section 5.5 Fix bug Cpu0InstrInfo.cpp SW to ST. Correct LW to LD; LB to LDB; SB to STB.

Version 3.1.1, Released November 28, 2012

Add Revision history. Correct ldi instruction error (replace ldi instruction with addiu from the beginning and in the all example code). Move ldi instruction change from section of “Adjust cpu0 instruction and support type of local variable pointer” to Section “CPU0 processor architecture”. Correct some English & typing errors.

1.7 Licensing

<http://llvm.org/docs/DeveloperPolicy.html#license>

1.8 Motivation

My intention in writing this book stems from my curiosity about how a simple and robotic CPU ISA, along with an LLVM-based software toolchain, can be designed and implemented.

Table 1.1: Number of lines in source code (including spaces and comments) for Cpu0

Components	Number of lines
llvm	15,000
llvm-objdump	8
elf2hex	765
verilog	600
lld	140
clang	500
compiler-rt’s builtin	5 (abort.c)
total	17,018

- Though the LLVM backend’s source code can be ported from another backend, it still requires a lot of thought and effort to do so, making the process not entirely easy.

We all learned computer knowledge in school through conceptual books. Concepts provide an effective way to understand the big picture. However, when developing real, complex systems, we often find that the concepts from school or books are insufficient or lack detail.

A compiler is a highly complex system. Traditionally, students learn about compilers conceptually and complete homework assignments using yacc/lex tools to translate parts of C or another high-level language into an intermediate representation (IR) or assembly. This approach helps them understand parsing and tool applications.

On the other hand, compiler engineers who graduate from school often face real market CPUs and complex specifications. Due to market demands, there exist multiple CPU series and ABIs (Application Binary Interfaces) to handle. Furthermore,

for performance reasons, real compiler backend implementations are too complex to serve as learning materials, even for a CPU with a single ABI.

This book develops a compiler backend alongside a simple, educational CPU called Cpu0. It includes implementations of a compiler backend, linker, llvm-objdump, elf2hex, and the Verilog source code for Cpu0’s instruction set. We provide readers with full source code to compile C/C++ programs and observe how they run on the Cpu0 machine implemented in Verilog. Through this educational CPU, readers gain insight into compiler backends, linkers, system tools, and CPU design. In contrast, real-world CPUs and compilers are too complex for a single person to fully understand or develop alone.

From my observations, LLVM is favored by some software engineers over GCC for two reasons. The first is political, as LLVM uses the BSD license¹². The second is technical, as LLVM follows the three-tier compiler software structure and leverages C++ object-oriented programming. GCC was originally written in C and only adopted C++ nearly 20 years later³. Some speculate that GCC adopted C++ simply because LLVM did.

I learned object-oriented programming in C++ during my studies. After reading books on “Design Patterns,” “C++/STL,” and “Object-Oriented Design,” I realized that C is easier to trace, whereas C++ enables the creation of reusable software units, known as objects. If a programmer has a strong understanding of design patterns, C++ provides better reusability and modifiability. A book I read on “system languages” defined software quality based on readability, modifiability, reusability, and performance. Object-oriented programming was introduced to manage large and complex software projects.

Given that compilers and operating systems are undeniably complex, why do GCC and Linux still avoid using C++?⁴ This is one reason I chose to develop a backend under LLVM rather than GCC.

1.9 Preface

The LLVM Compiler Infrastructure provides a versatile framework for creating new backends. Once you familiarize yourself with this structure, creating a new backend should not be too difficult. However, the available backend documentation is fairly high level and omits many details. This tutorial provides step-by-step instructions for writing a new backend for a new target architecture from scratch.

We will use the Cpu0 architecture as an example to build our backend. Cpu0 is a simple RISC architecture designed for educational purposes. More information about Cpu0, including its instruction set, is available here⁵. The Cpu0 example code referenced in this book can be found <http://jonathan2251.github.io/lbd/lbdex.tar.gz>. As you progress through each chapter, you will incrementally build the backend’s functionality.

Since Cpu0 is a simple RISC CPU for educational purposes, the LLVM backend code for it is also simple and easy to learn. Additionally, Cpu0 provides Verilog source code that can be run on a PC or FPGA platform, as explained in the chapter “Verify Backend on Verilog Simulator.” To illustrate backend design, we carefully design C/C++ programs for each newly added function in every chapter. Through these example codes, readers can understand which LLVM intermediate representations (IRs) the backend transforms and how these IRs correspond to the original C/C++ code.

This tutorial initially used the LLVM 3.1 MIPS backend as a reference and was later synchronized with LLVM 3.5 MIPS at version 3.5.3. Based on our experience, referencing and synchronizing with an existing backend helps enhance features and fix bugs. By comparing differences across versions, you can leverage the LLVM development team’s efforts to improve your backend.

Since Cpu0 is an educational architecture, it lacks key documentation needed for compiler development, such as an Application Binary Interface (ABI). To implement our backend, we use the MIPS ABI as a reference. You may find it helpful to familiarize yourself with relevant parts of the MIPS ABI as you progress through this tutorial.

This document also serves as a tutorial for toolchain development for a new CPU architecture. Many programmers graduate with knowledge of compilers and computer architecture but lack professional experience in compiler or CPU design.

¹ <http://llvm.org/docs/DeveloperPolicy.html#license>

² http://www.phoronix.com/scan.php?page=news_item&px=MTU4MjA

³ http://en.wikipedia.org/wiki/GNU_Compiler_Collection

⁴ <http://en.wikipedia.org/wiki/C%2B%2B>

⁵ <http://ccckmit.wikidot.com/ocs:cpu0>

This document introduces these engineers to toolchain programming and CPU design using the LLVM infrastructure—without requiring the purchase of any software or hardware. A computer is the only device needed.

Finally, this book is not a conceptual compiler textbook. It is intended for readers interested in extending a compiler toolchain to support a new CPU based on LLVM. Programming on Linux does not require understanding every detail of the operating system. For example, when developing a USB device driver for Linux, a programmer studies the USB specification, the Linux USB subsystem, and the common device driver model and APIs. Similarly, this book focuses on practical implementation rather than compiler theory.

In the same way, when extending functions in a large software project like the LLVM umbrella project, you should focus on achieving your goal and ignore irrelevant details.

Trying to understand every line of source code in detail is unrealistic if your project involves extending a well-defined software structure. It only makes sense when rewriting the entire software structure.

Of course, if more books or documents about LLVM backend development were available, readers would have more opportunities to understand LLVM by studying them.

1.10 Prerequisites

Readers should be comfortable with the C++ language and Object-Oriented Programming concepts. LLVM is developed in C++ and follows a modular design, allowing various classes to be adapted and reused efficiently.

Having a conceptual understanding of how compilers work is beneficial. If you have implemented compilers before, you will likely have no trouble following this tutorial. Since this tutorial builds an LLVM backend step by step, we will introduce important concepts as needed.

This tutorial references the following materials. We highly recommend reading these documents to gain a deeper understanding of the topics covered:

[The Architecture of Open Source Applications Chapter on LLVM](#)

[LLVM's Target-Independent Code Generation documentation](#)

[LLVM's TableGen Fundamentals documentation](#)

[LLVM's Writing an LLVM Compiler Backend documentation](#)

[Description of the Tricore LLVM Backend](#)

[Mips ABI document](#)

1.11 Outline of Chapters

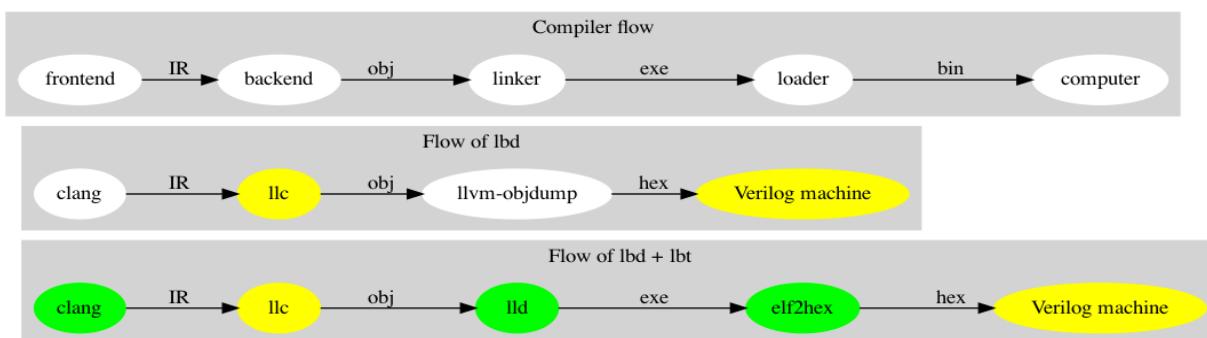


Fig. 1.1: Code generation and execution flow

The top part of Fig. 1.1 represents the workflow and software packages involved in generating and executing a computer program. IR stands for Intermediate Representation.

The middle part illustrates this book's workflow. Except for Clang, the other components need to be extended for a new backend development. Although the Cpu0 backend extends Clang as well, it uses the MIPS ABI and can utilize MIPS-Clang. This book implements the sections highlighted in yellow. The green sections, which include lld and elf2hex for the Cpu0 backend, can be found at: <http://jonathan2251.github.io/lbt/index.html>.

The hex format is an ASCII file representation that uses characters '0' to '9' and 'a' to 'f' to encode hexadecimal values, as the Verilog machine reads it as an input file.

This book includes 10,000 lines of source code covering:

1. Step-by-step creation of an LLVM backend for the Cpu0, from Chapters 2 to 11.
2. Cpu0 Verilog source code, discussed in Chapter 12.

With this code, readers can generate Cpu0 machine code through the Cpu0 LLVM backend compiler and observe how it executes on a computer. However, execution is only possible for code that does not contain global variables or relocation records requiring linker handling. The book is also available in PDF and EPUB formats online.

This tutorial is aimed at LLVM backend developers but is not intended for experts. It serves as a valuable resource for those familiar with compiler concepts and computer architecture who wish to learn how to extend the LLVM toolchain to support a new CPU.

Cpu0 Architecture and LLVM Structure:

This chapter introduces the Cpu0 architecture, provides a high-level overview of LLVM, and explains how Cpu0 will be targeted in an LLVM backend. It guides you through the initial steps of backend development, including target description (TD), CMake setup, and target registration. By the end of this chapter, around 750 lines of source code will be added.

Backend structure:

This chapter outlines the structure of an LLVM backend using UML diagrams. It continues the development of the Cpu0 backend, adding thousands of lines of source code. Many of these lines are common across LLVM backends, regardless of the target architecture.

By the end of this chapter, the Cpu0 LLVM backend will support fewer than ten instructions and be capable of generating some initial assembly output.

Arithmetic and Logic Instructions:

Over ten C operators and their corresponding LLVM IR instructions are introduced in this chapter.

A few hundred lines of source code, mostly in `.td` Target Description files, are added. With these lines of source code, the backend can now translate the `+`, `-`, `*`, `/`, `&`, `|`, `^`, `<<`, `>>`, `!` and `%` C operators into the appropriate Cpu0 assembly code.

Usage of the `llc` debug option and `Graphviz` as a debug tool are introduced in this chapter.

Generating object files:

Object file generation support for the Cpu0 backend is added in this chapter, as the Target Registration structure is introduced.

Based on the LLVM structure, the Cpu0 backend can generate big-endian and little-endian ELF object files with minimal effort.

Global Variables:

Global variable handling is added in this chapter. Cpu0 supports both PIC and static addressing modes. Both addressing modes are explained as their functionalities are implemented.

Other data type:

In addition to the `int` type, other data types such as pointers, `char`, `bool`, `long long`, structures, and arrays are added in this chapter.

Control flow statements:

Support for flow control statements, such as `if`, `else`, `while`, `for`, `goto`, `switch`, and `case`, as well as both a simple optimization software pass and hardware instructions for control statement optimization, are discussed in this chapter.

Function call:

This chapter details the implementation of function calls in the Cpu0 backend. The stack frame, handling of incoming and outgoing arguments, and their corresponding standard LLVM functions are introduced.

ELF Support:

This chapter details Cpu0 support for the well-known ELF object file format. The ELF format and binutils tools are not part of LLVM but are introduced. This chapter explains how to use ELF tools to verify and analyze the object files created by the Cpu0 backend.

The disassembly command `llvm-objdump -d` support for Cpu0 is added in the last section of this chapter.

Assembler:

Support for translating hand-written assembly language into object files under the LLVM infrastructure.

C++ support:

Support C++ language features. It's under working.

Verify backend on Verilog simulator:

First, create the Cpu0 virtual machine using the Verilog language with the Icarus tool. Using this tool, feed the hex file generated by `llvm-objdump` to the Cpu0 virtual machine and observe the execution results on a PC.

Appendix A: Getting Started: Installing LLVM and the Cpu0 Example Code:

This section details how to set up the LLVM source code, development tools, and environment configuration for macOS and Linux platforms.

Appendix B: Cpu0 document and test:

This book uses Sphinx to generate PDF and EPUB document formats. Details on how to install the necessary tools, generate these documents, and perform regression testing for the Cpu0 backend are included.

CPU0 ARCHITECTURE AND LLVM STRUCTURE

- *Cpu0 Processor Architecture Details*
 - Brief introduction
 - The Cpu0 Instruction Set
 - * Why Not Use ADD Instead of SUB?
 - The Status Register
 - Cpu0's Stages of Instruction Execution
 - Cpu0's Interrupt Vector
- *Clang*
 - Context Free Grammar
 - Why doesn't the Clang compiler use YACC/LEX tools to parse C++?
 - Compiler-Compiler Tools for Context-Sensitive C++ Parsing
- *LLVM Structure*
 - SSA Form
 - DSA Form
 - Three-Phase Design
 - LLVM's Target Description Files: .td
 - LLVM Code Generation Sequence
 - LLVM vs. GCC in Structure
 - LLVM Blog
 - CFG (Control Flow Graph)
 - DAG (Directed Acyclic Graph)
 - Instruction Selection
 - Caller and Callee Saved Registers
 - Live-In and Live-Out Registers
- *Create Cpu0 Backend*
 - Cpu0 Backend Machine ID and Relocation Records

- *Creating the Initial Cpu0 .td Files*
- *Target Registration*
- *Build Libraries and .td Files*
- *Debug options*
 - *Options for llc Debugging*
 - *opt Debugging Options*

Before you begin this tutorial, you should know that you can always try to develop your own backend by porting code from existing backends. The majority of the code you will want to investigate can be found in the `/lib/Target` directory of your root LLVM installation. As most major RISC instruction sets have some similarities, this may be the avenue you might try if you are an experienced programmer and knowledgeable of compiler backends.

On the other hand, there is a steep learning curve and you may easily get stuck debugging your new backend. You can easily spend a lot of time tracing which methods are callbacks of some function, or which are calling some overridden method deep in the LLVM codebase - and with a codebase as large as LLVM, all of this can easily become difficult to keep track of. This tutorial will help you work through this process while learning the fundamentals of LLVM backend design. It will show you what is necessary to get your first backend functional and complete, and it should help you understand how to debug your backend when it produces incorrect machine code using output provided by the compiler.

This chapter details the Cpu0 instruction set and the structure of LLVM. The LLVM structure information is adapted from Chris Lattner's LLVM chapter of the *Architecture of Open Source Applications* book¹⁰. You can read the original article from the AOSA website if you prefer.

At the end of this Chapter, you will begin to create a new LLVM backend by writing register and instruction definitions in the Target Description files which will be used in next chapter.

Finally, there are compiler knowledge like DAG (Directed-Acyclic-Graph) and instruction selection needed in llvm backend design, and they are explained here.

2.1 Cpu0 Processor Architecture Details

This section is based on materials available here² (Chinese) and here³ (English). However, I changed some ISA from original Cpu0 for designing a simple integer operational CPU and llvm backend. This is my intention for writing this book that I want to know what a simple and robotic CPU ISA and llvm backend can be.

2.1.1 Brief introduction

Cpu0 is a 32-bit architecture. It has 16 general purpose registers (`R0, ..., R15`), co-processor registers (like Mips), and other special registers. Its structure is illustrated in Fig. 2.1 below.

The registers are used for the following purposes:

¹⁰ Chris Lattner, **LLVM**. Published in The Architecture of Open Source Applications. <http://www.aosabook.org/en/llvm.html>

² Original Cpu0 architecture and ISA details (Chinese). <http://ccckmit.wikidot.com/ocs:cpu0>

³ English translation of Cpu0 description. http://translate.google.com.tw/translate?js=n&prev=_t&hl=zh-TW&ie=UTF-8&layout=2&eotf=1&sl=zh-CN&tl=en&u=http://ccckmit.wikidot.com/ocs:cpu0

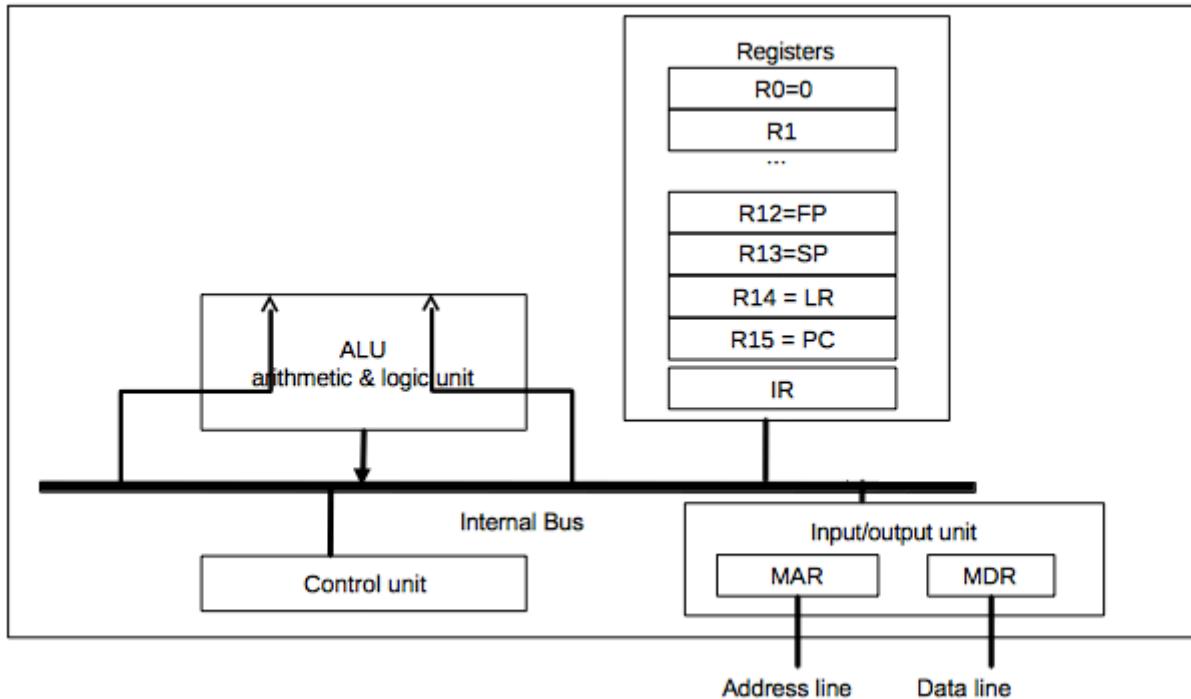


Fig. 2.1: Architectural block diagram of the Cpu0 processor

Table 2.1: Cpu0 general purpose registers (GPR)

Register	Description
R0	Constant register, value is 0
R1-R10	General-purpose registers
R11	Global Pointer register (GP)
R12	Frame Pointer register (FP)
R13	Stack Pointer register (SP)
R14	Link Register (LR)
R15	Status Word Register (SW)

Table 2.2: Cpu0 co-processor 0 registers (C0R)

Register	Description
0	Program Counter (PC)
1	Error Program Counter (EPC)

Table 2.3: Cpu0 other registers

Register	Description
IR	Instruction register
MAR	Memory Address Register (MAR)
MDR	Memory Data Register (MDR)
HI	High part of MULT result
LO	Low part of MULT result

2.1.2 The Cpu0 Instruction Set

The Cpu0 instruction set is categorized into three types:

- **L-type instructions:** Primarily used for memory operations.
- **A-type instructions:** Designed for arithmetic operations.
- **J-type instructions:** Typically used for altering control flow (e.g., jumps).

Fig. 2.2 illustrates the bitfield breakdown for each instruction type.

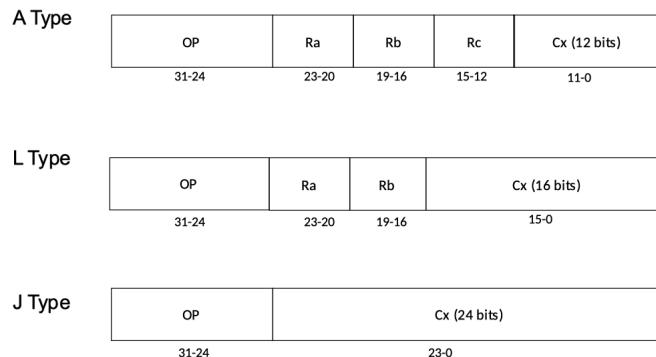


Fig. 2.2: Cpu0's three instruction formats

Table 2.4: C, llvm-ir¹³ and Cpu0

C	llvm-ir	Cpu0	I or II	Comment
=	load/store	ld/lb/lbu/lh/lhu	I	
&, &&	and	and	I	
,	or	or	I	
^	xor	xor/nor	I	! can be got from two ir
!		<ul style="list-style-type: none"> • %tobool = icmp ne i32 %6, 0 • %lnot = xor i1 %tobool, true 		
==, !=, <, <=, >, >=	icmp/fcmp <cond> cond: eq/ne,	cmp/ucmp ...+ floating-lib	I	
"	"	slt/sltu/slti/slтиu	II	slti/slтиu: ex. a == 3 reduce instructions
if (a <= b)	icmp/fcmp <cond> + br i1 <cond>, ...	cmp/uncmp + jeq/jne/jlt/jgt/jle/jge	I	Conditional branch
if (bool)	br i1 <cond>, ...	jeq/jne	I	
"	"	beq/bne	II	
goto	br <dest>	jmp	I	Unconditional branch
call sub-function	call	jsub	I	Provide 24-bit address range of calling sub-function (the address from caller to callee is within 24-bit)
"	"	jalr	I	Add for 32-bit address range of calling sub-function
return	ret	ret	I	
+, -, *	add/fadd, sub/fsub, mul/fmul	add/addu/addiu, sub/subu, mul	I	
/, %	udiv/sdiv/fdiv, urem/srem/frem	div, mfhi/mflo/mthi/m	I	
<<, >>	shl, lshr/ashr	shl/rol/rolv, srl/sra/ror/rorv	II	
float <-> int	fptoui, sitofp, ...			Cpu0 uses SW for floating value, and these two IR are for HW floating instruction
__builtin_c	llvm.clz/llvm.clo	floating-lib + clz, clo	I	For SW floating-lib, uses __builtin_clz / __builtin_clo in clang and clang generates llvm.clz/llvm.clo intrinsic function
__builtin_e	llvm.eh.xxx	st/ld	I	pass information to exception handler through \$4, \$5

 Table 2.5: C++, llvm-ir¹³ and Cpu0

C++	llvm-ir	Cpu0	I or II	Comment
try { }	invoke void @_Z15throw_exception	jsub _Z15throw_exception	I	
catch { }	landingpad...catch	st and ld	I	st/ld \$4 & \$5 to/from stack, \$4:exception address, \$5: exception typeid

¹³ <http://llvm.org/docs/LangRef.html>

Note

Selection of LLVM-IR and the ISA for a RISC CPU

- LLVM-IR and the ISA of a RISC CPU emerged after the C language. As shown in the table above, they can be selected based on C language constructs.
- Not listed in the table, LLVM-IR includes terminator instructions such as *switch*, *invoke*, and others, as well as atomic operations and a variety of LLVM intrinsics. These intrinsics provide better performance for backend implementations, such as *llvm.vector.reduce*. *.
- For vector processing on CPUs/GPUs, vector-type math LLVM-IR or LLVM intrinsics can be used for implementation.

Note

Selection of the ISA for Cpu0

- The original author of Cpu0 designed its ISA as a teaching material, without focusing on performance.
- My goal is to refine the ISA selection and design, considering both its role as an LLVM tutorial and its basic performance as an ISA. I am not interested in a poorly designed ISA.
 - As shown in the table above, “*if (a <= b)*” can be rewritten as “*t = (a <= b)*” followed by “*if (t)*”. Thus, I designed **ISA II of Cpu0** to use “*slt + beq*” instead of “*cmp + jeq*”, reducing six conditional jump instructions (*jeq/jne/jlt/jgt/jle/jge*) to just two (*beq/bne*). This balances complexity and performance in the Cpu0 ISA.
 - For the same reason, I adopted **slt** from **MIPS** instead of **cmp** from **ARM**. This allows the destination register to be any general-purpose register (GPR), avoiding bottlenecks caused by a shared “status register.”
 - Floating-point operations can be implemented in software, so Cpu0 only supports integer instructions. I added **clz** (count leading zeros) and **clo** (count leading ones) to Cpu0 since floating-point libraries, such as *compiler-rt/builtin*, rely on these built-in functions. Floating-point normalization can leverage **clz** and **clo** for performance improvements. Although Cpu0 could use multiple instructions to implement *llvm.clz* and *llvm.clo*, having dedicated **clz/clo** instructions allows execution in a single instruction.
 - I extended **ISA II of Cpu0** for better performance, following the principles of MIPS.

The following table provides details on the cpu032I instruction set:

- First column F.: meaning Format.

Table 2.6: cpu032I Instruction Set

F.	Mnemonic	Op-code	Meaning	Syntax	Operation
L	NOP	00	No Operation		
L	LD	01	Load word	LD Ra, [Rb+Cx]	Ra <= [Rb+Cx]
L	ST	02	Store word	ST Ra, [Rb+Cx]	[Rb+Cx] <= Ra
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx] ⁴
L	LBu	04	Load byte unsigned	LBu Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx] ^{Page 24, 4}
L	SB	05	Store byte	SB Ra, [Rb+Cx]	[Rb+Cx] <= (byte)Ra
L	LH	06	Load half word	LH Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx] ^{Page 24, 4}

continues on next page

Table 2.6 – continued from previous page

F	Mnemonic	Op-code	Meaning	Syntax	Operation
L	LHu	07	Load half word unsigned	LHu Ra, [Rb+Cx]	Ra [Rb+Cx] <= (2bytes)[Rb+Cx] ^{Page 24, 4}
L	SH	08	Store half word	SH Ra, [Rb+Cx]	[Rb+Cx] <= Ra
L	ADDiu	09	Add immediate	ADDiu Ra, Rb, Cx	Ra <= (Rb + Cx)
L	ANDi	0C	AND imm	ANDi Ra, Rb, Cx	Ra <= (Rb & Cx)
L	ORi	0D	OR	ORi Ra, Rb, Cx	Ra <= (Rb Cx)
L	XORi	0E	XOR	XORi Ra, Rb, Cx	Ra <= (Rb ^ Cx)
L	LUi	0F	Load upper	LUi Ra, Cx	Ra <= (Cx << 16)
A	ADDu	11	Add unsigned	ADD Ra, Rb, Rc	Ra <= Rb + Rc ⁵
A	SUBu	12	Sub unsigned	SUB Ra, Rb, Rc	Ra <= Rb - Rc ^{Page 24, 5}
A	ADD	13	Add	ADD Ra, Rb, Rc	Ra <= Rb + Rc ^{Page 24, 5}
A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra <= Rb - Rc ^{Page 24, 5}
A	CLZ	15	Count Leading Zero	CLZ Ra, Rb	Ra <= bits of leading zero on Rb
A	CLO	16	Count Leading One	CLO Ra, Rb	Ra <= bits of leading one on Rb
A	MUL	17	Multiply	MUL Ra, Rb, Rc	Ra <= Rb * Rc
A	AND	18	Bitwise and	AND Ra, Rb, Rc	Ra <= Rb & Rc
A	OR	19	Bitwise or	OR Ra, Rb, Rc	Ra <= Rb Rc
A	XOR	1A	Bitwise exclusive or	XOR Ra, Rb, Rc	Ra <= Rb ^ Rc
A	NOR	1B	Bitwise boolean nor	NOR Ra, Rb, Rc	Ra <= Rb nor Rc
A	ROL	1C	Rotate left	ROL Ra, Rb, Cx	Ra <= Rb rol Cx
A	ROR	1D	Rotate right	ROR Ra, Rb, Cx	Ra <= Rb ror Cx
A	SHL	1E	Shift left	SHL Ra, Rb, Cx	Ra <= Rb << Cx
A	SHR	1F	Shift right	SHR Ra, Rb, Cx	Ra <= Rb >> Cx
A	SRA	20	Shift right	SRA Ra, Rb, Cx	Ra <= Rb ' >> Cx ⁷
A	SRAV	21	Shift right	SRAV Ra, Rb, Rc	Ra <= Rb ' >> Rc ^{Page 24, 7}
A	SHLV	22	Shift left	SHLV Ra, Rb, Rc	Ra <= Rb << Rc
A	SHRV	23	Shift right	SHRV Ra, Rb, Rc	Ra <= Rb >> Rc
A	ROL	24	Rotate left	ROL Ra, Rb, Rc	Ra <= Rb rol Rc
A	ROR	25	Rotate right	ROR Ra, Rb, Rc	Ra <= Rb ror Rc
A	CMP	2A	Compare	CMP Ra, Rb	SW <= (Ra cond Rb) ⁶
A	CMPu	2B	Compare	CMPu Ra, Rb	SW <= (Ra cond Rb) ^{Page 24, 6}
J	JEQ	30	Jump if equal (==)	JEQ Cx	if SW(==), PC <= PC + Cx
J	JNE	31	Jump if not equal (!=)	JNE Cx	if SW(!=), PC <= PC + Cx
J	JLT	32	Jump if less than (<)	JLT Cx	if SW(<), PC <= PC + Cx
J	JGT	33	Jump if greater than (>)	JGT Cx	if SW(>), PC <= PC + Cx
J	JLE	34	Jump if less than or equals (<=)	JLE Cx	if SW(<=), PC <= PC + Cx
J	JGE	35	Jump if greater than or equals (>=)	JGE Cx	if SW(>=), PC <= PC + Cx
J	JMP	36	Jump (unconditional)	JMP Cx	PC <= PC + Cx
J	JALR	39	Indirect jump	JALR Rb	LR <= PC; PC <= Rb ⁸
J	BAL	3A	Branch and link	BAL Cx	LR <= PC; PC <= PC + Cx
J	JSUB	3B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	JR/RET	3C	Return from subroutine	JR \$1 or RET LR	PC <= LR ⁹
A	MULT	41	Multiply for 64 bits result	MULT Ra, Rb	(HI,LO) <= MULT(Ra,Rb)
A	MULTU	42	MULT for unsigned 64 bits	MULTU Ra, Rb	(HI,LO) <= MULTU(Ra,Rb)
A	DIV	43	Divide	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
A	DIVU	44	Divide unsigned	DIVU Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb

continues on next page

Table 2.6 – continued from previous page

F	Mnemonic	Op-code	Meaning	Syntax	Operation
A	MFHI	46	Move HI to GPR	MFHI Ra	Ra <= HI
A	MFLO	47	Move LO to GPR	MFLO Ra	Ra <= LO
A	MTHI	48	Move GPR to HI	MTHI Ra	HI <= Ra
A	MTLO	49	Move GPR to LO	MTLO Ra	LO <= Ra
A	MFC0	50	Move C0R to GPR	MFC0 Ra, Rb	Ra <= Rb
A	MTC0	51	Move GPR to C0R	MTC0 Ra, Rb	Ra <= Rb
A	C0MOV	52	Move C0R to C0R	C0MOV Ra, Rb	Ra <= Rb

The following table provides details on the newly added cpu032II instruction set:

Table 2.7: cpu032II Instruction Set

F	Mnemonic	Op-code	Meaning	Syntax	Operation
L	SLTi	26	Set less Then	SLTi Ra, Rb, Cx	Ra <= (Rb < Cx)
L	SLTi _u	27	SLTi unsigned	SLTi _u Ra, Rb, Cx	Ra <= (Rb < Cx)
A	SLT	28	Set less Then	SLT Ra, Rb, Rc	Ra <= (Rb < Rc)
A	SLTu	29	SLT unsigned	SLTu Ra, Rb, Rc	Ra <= (Rb < Rc)
L	BEQ	37	Branch if equal	BEQ Ra, Rb, Cx	if (Ra==Rb), PC <= PC + Cx
L	BNE	38	Branch if not equal	BNE Ra, Rb, Cx	if (Ra!=Rb), PC <= PC + Cx

⁴ The difference between LB and LBu is signed and unsigned byte value expand to a word size. For example, After LB Ra, [Rb+Cx], Ra is 0xffffffff80(= -128) if byte [Rb+Cx] is 0x80; Ra is 0x0000007f(= 127) if byte [Rb+Cx] is 0x7f. After LBu Ra, [Rb+Cx], Ra is 0x00000080(= 128) if byte [Rb+Cx] is 0x80; Ra is 0x0000007f(= 127) if byte [Rb+Cx] is 0x7f. Difference between LH and LHu is similar.

⁵ The only difference between ADDu instruction and the ADD instruction is that the ADDu instruction never causes an Integer Overflow exception. SUBu and SUB is similar.

⁷ Rb ‘>> Cx, Rb ‘>> Rc: Shift with signed bit remain.

⁶ CMP is signed-compare while CMPu is unsigned. Conditions include the following comparisons: >, >=, ==, !=, <, <=. SW is actually set by the subtraction of the two register operands, and the flags indicate which conditions are present.

⁸ jsrb cx is direct call for 24 bits value of cx while jalr \$rb is indirect call for 32 bits value of register \$rb.

⁹ Both JR and RET has same opcode (actually they are the same instruction for Cpu0 hardware). When user writes “jr \$t9” meaning it jumps to address of register \$t9; when user writes “jr \$lr” meaning it jump back to the caller function (since \$lr is the return address). For user readability, Cpu0 prints “ret \$lr” instead of “jr \$lr”.

Note

Cpu0 Unsigned Instructions

Like MIPS, except for *DIVU*, arithmetic unsigned instructions such as *ADDu* and *SUBu* do not trigger overflow exceptions. The *ADDu* and *SUBu* handle both signed and unsigned integers correctly.

For example:

- $(ADDu\ 1, -2) = -1$
- $(ADDu\ 0x01, 0xffffffffe) = 0xffffffff\ (4G - 1)$

If you interpret the result as a negative value, it is -1 . If interpreted as positive, it is $+4G - 1$.

Why Not Use ADD Instead of SUB?

From introductory computer science textbooks, we know that *SUB* can be replaced by *ADD* as follows:

- $(A - B) = (A + (-B))$

Since MIPS represents *int* in C using 32 bits, consider the case where $B = -2G$:

- $(A - (-2G)) = (A + 2G)$

However, the problem is that while $-2G$ can be represented in a 32-bit machine, $+2G$ cannot. This is because the range of 32-bit two's complement representation is $(-2G \dots 2G-1)$.

Two's complement representation allows for efficient computation in hardware design, making it widely used in real CPU implementations. This is why almost every CPU includes a *SUB* instruction rather than relying solely on *ADD*.

2.1.3 The Status Register

The Cpu0 status word register (*SW*) contains the state of the following flags:

- **Negative (N)**
- **Zero (Z)**
- **Carry (C)**
- **Overflow (V)**
- **Debug (D)**
- **Mode (M)**
- **Interrupt (I)**

The bit layout of the *SW* register is shown in Fig. 2.3 below.

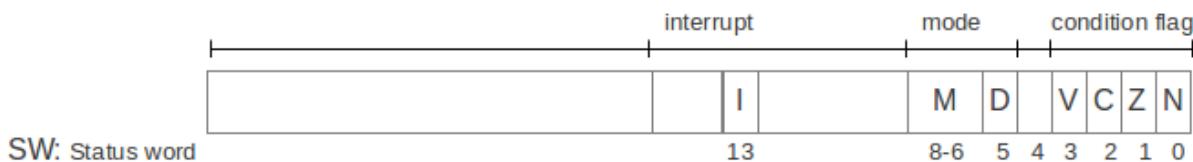


Fig. 2.3: Cpu0 status word (*SW*) register

When a *CMP Ra, Rb* instruction executes, it updates the condition flags in the status word (*SW*) register as follows:

- If $R_a > R_b$, then $N = 0, Z = 0$
- If $R_a < R_b$, then $N = 1, Z = 0$
- If $R_a = R_b$, then $N = 0, Z = 1$

The direction (i.e., taken or not taken) of conditional jump instructions (*JGT*, *JLT*, *JGE*, *JLE*, *JEQ*, *JNE*) is determined by the values of the N and Z flags in the *SW* register.

2.1.4 Cpu0's Stages of Instruction Execution

The Cpu0 architecture has a five-stage pipeline. The stages are: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write-back (WB).

Below is a description of what happens in each stage of the processor:

1) Instruction Fetch (IF)

- The Cpu0 fetches the instruction pointed to by the Program Counter (PC) into the Instruction Register (IR): $IR = [PC]$.
- The PC is then updated to point to the next instruction: $PC = PC + 4$.

2) Instruction Decode (ID)

- The control unit decodes the instruction stored in *IR*, routes necessary data from registers to the ALU, and sets the ALU's operation mode based on the instruction's opcode.

3) Execute (EX)

- The ALU executes the operation designated by the control unit on the data in registers.
- Except for load and store instructions, the result is stored in the destination register after execution.

4) Memory Access (MEM)

- If the instruction is a load, data is read from the data cache into the pipeline register *MEM/WB*.
- If the instruction is a store, data is written from the register to the data cache.

5) Write-Back (WB)

- If the instruction is a load, data is moved from the pipeline register *MEM/WB* to the destination register.

2.1.5 Cpu0's Interrupt Vector

Table 2.8: Cpu0's Interrupt Vector

Address	type
0x00	Reset
0x04	Error Handle
0x08	Interrupt

2.2 Clang

LLVM is middleware for compilers, with Clang as its frontend. The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, and CUDA) for the LLVM project.

2.2.1 Context Free Grammar

Definition:

- “A context-free grammar defines a language that can be parsed independently of surrounding input context; each production rule applies based solely on the current nonterminal, not on neighboring symbols.”
- All context-free grammars (CFGs) can be expressed in Backus-Naur Form (BNF).

Note

Computer languages have been adding complexity features for users’ programming:

☆ “(\approx 30 years ago) Programming languages are **context-free**.” \Rightarrow “(Today) The syntax is context-free, but the language is **context-sensitive**.”

1. What older textbooks said

Textbooks from the 1980s–1990s typically taught:

- “**Programming languages** are mostly **context-free**.”
- “**Parsing is done with CFGs**.”
- “Semantic analysis comes later.”

They treated semantic constraints as a separate phase, not as part of the grammar

2. What modern textbooks say

Modern compiler books (e.g., newer editions of Aho/Ullman, Appel, Cooper/Torczon, Muchnick, and engineering-oriented texts) now teach something closer to:

- The surface syntax is context-free.
- Real languages require context-sensitive semantic analysis.
- Some languages **require semantic information during parsing** (C++, Rust, Swift).
- Macro systems and type inference break the clean CFG model.

They **no longer pretend that a CFG fully describes a real language**.

So the modern interpretation is:

- “The grammar is context-free, but the language is not.”

This is a subtle but important shift.

Table 2.9: Context-Free Grammar Shifts in Compiler Theory

Statement	Meaning	How Modern Languages Changed
Old Textbook Statement (\approx 30 years ago) “ Programming languages are context-free .”	Focused only on the parser grammar . Treated semantic rules as a separate phase, not part of the language definition.	Languages were simpler (C, Pascal, early C++). Few features required semantic feedback during parsing. Grammar-based teaching matched real compilers more closely.
Modern Understanding (today) “ The syntax is context-free, but the language is context-sensitive .”	Parsing still uses a CFG, but real languages rely on name resolution, type inference, generics, macro expansion, and semantic disambiguation .	Modern languages (C++, Rust, Swift, Kotlin, Go) require semantic information during parsing. Macro systems and type systems break pure CFG boundaries. Compilers use multi-phase frontends to resolve context.

2.2.2 Why doesn't the Clang compiler use YACC/LEX tools to parse C++?

Clang does not use YACC/LEX because **C++ is too complex and context-sensitive** for traditional parser generators. YACC and LEX work with context-free grammars, but C++ has many context-sensitive features, especially in templates below:

Context-sensitive template instantiation

```
#include <iostream>

#if TEMPLATE==1
template <typename T>
void f(T x) {
    std::cout << "Template f(T)" << std::endl;
}
#endif

#if FUNCTION==1
void f(int x) {
    std::cout << "Non-template f(int)" << std::endl;
}
#endif

int main() {
#if (TEMPLATE==1) || (FUNCTION==1)
    f(42);           // Which one gets called?
    f('a');          // Template or non-template?
#endif
#if TEMPLATE==1
    f<int>('a');   // Explicit template instantiation
#endif
}
```

```
References % clang++ -DFUNCTION=1 -DTEMPLATE=0 cpp-template.cpp
References % ./a.out
Non-template f(int)
Non-template f(int)
References % clang++ -DFUNCTION=0 -DTEMPLATE=1 cpp-template.cpp
References % ./a.out
Template f(T)
Template f(T)
Template f(T)
References % clang++ -DFUNCTION=1 -DTEMPLATE=1 cpp-template.cpp
References % ./a.out
Non-template f(int)
Template f(T)
Template f(T)
```

In the C++ code above, both `f(42)` and `f('a')` can match either the template function or the non-template function.

✓ Why This Is Hard for YACC:

YACC operates on context-free grammars, but this example is context-sensitive. The expression `f('a')`; selects a template if a template definition exists; otherwise, it selects a function if a function definition exists. As a result, this behavior cannot be implemented using BNF-based tools like YACC/LEX.

To parse this example, the following are required:

- Template argument deduction: The compiler must infer T from the call.
- Overload resolution: It must choose between the template and non-template versions.
- Implicit conversions: ‘a’ can be converted to int, which affects overload ranking.
- Explicit template instantiation: `f<int>('a')` forces the template, but YACC doesn’t track template types.

To model this in YACC:

You’d need to simulate template instantiation and ranking —which is way beyond what YACC was designed for.

This kind of logic is not just syntactic —it’s deeply semantic. That’s why compilers like Clang use handwritten parsers with tight integration between parsing and semantic analysis.

Clang doesn’t use YACC/LEX because:

Feature	YACC/LEX	Hand-written Parser
Handles context-sensitive grammar	✗	✓
Good error recovery	✗	✓
Integration with semantic analysis	✗	✓
Easy to maintain/extend for C++	✗	✓
Fine-grained control	✗	✓

The GNU `g++` compiler abandoned BNF tools starting from version 3.x.

2.2.3 Compiler-Compiler Tools for Context-Sensitive C++ Parsing

While traditional tools like YACC/Lex are limited to context-free grammars, modern compiler construction requires handling context-sensitive features —especially in C++ templates, overload resolution, and semantic analysis. Below is a list of tools that attempt to address these challenges.

Tool	Generates Parser Code?	Context-Sensitive Support?	Notes
ANTLR	✓ Yes	△ Limited	Supports semantic predicates; struggles with full C++ complexity
BNFLite	✓ Yes	△ Partial	Lightweight C++ template library; ideal for DSLs, not full C++
PEGTL	✓ Yes	△ Limited	PEG-based parser combinator library in C++; expressive but limited
GLR Parsers (Elsa)	✓ Yes	✓ Yes	Can handle ambiguity and deferred resolution; used in research
Clang LibTool-ing	✓ Yes (via AST)	✓ Yes	Offers full C++ parsing + semantic analysis; industrial-grade tooling

Why Most Tools Fall Short:

- C++ templates are **Turing-complete**, making static analysis alone insufficient.
- Overload resolution requires understanding **types, scopes, and conversions**.
- C++ syntax is **deeply ambiguous**, defying context-free parsing strategies.

Recommended Approach:

For building C++ parsers:

- Use **GLR-based tools** like Elsa if ambiguity and template complexity must be handled directly.
- Or leverage **Clang LibTooling** for full semantic integration, AST manipulation, and robust code analysis.

In summary, while modern tools improve on YACC/LEX, **the complexity of C++ still requires a custom parser that deeply integrates with semantic analysis and type resolution. Clang's approach remains the most practical for full C++ support. Moreover the error messages and recovery are still weaker than Clang.**

While C++ compilers do not benefit from BNF generator tools, many other programming and scripting languages, which are more context-free, can take advantage of them. The following information comes from Wikipedia:

Java syntax has a context-free grammar that can be parsed by a simple LALR parser. Parsing C++ is more complicated¹.

2.3 LLVM Structure

This section introduces the compiler's data structures, algorithms, and mechanisms used in LLVM.

2.3.1 SSA Form

Static Single Assignment (SSA) form ensures that each variable is assigned exactly once. In SSA form, a single instruction has one variable (destination virtual registers). However one virtual register may map to two real registers. LLVM handles it by packing them into a single value, like a struct or a vector, or using multiple instructions as follows:

```
%res = call {i32, i1} @llvm.sadd.with.overflow.i32(i32 %a, i32 %b)
%sum = extractvalue {i32, i1} %res, 0
%overflow = extractvalue {i32, i1} %res, 1

%y = call <4 x float> @llvm.ceil.v4f32(<4 x float> %x)
```

LLVM IR follows SSA form, meaning it has an **unbounded number of virtual registers**—each variable is assigned exactly once and is stored in a separate virtual register.

As a result, the optimization steps in the code generation sequence—including **Instruction Selection, Scheduling and Formation**, and **Register Allocation**—retain all optimization opportunities.

For example, if we used a limited number of virtual registers instead of an unlimited set, as shown in the following code:

```
%a = add nsw i32 1, i32 0
store i32 %a, i32* %c, align 4
%a = add nsw i32 2, i32 0
store i32 %a, i32* %c, align 4
```

In the above example, a limited number of virtual registers is used, causing virtual register `%a` to be assigned twice.

As a result, the compiler must generate the following code, since `%a` is assigned as an output in two different statements.

```
=> %a = add i32 1, i32 0
    st %a, i32* %c, 1
    %a = add i32 2, i32 0
    st %a, i32* %c, 2
```

¹ https://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B

The above code must execute sequentially.

In contrast, the SSA form shown below can be reordered and executed in parallel using the following alternative version¹⁶.

```
%a = add nsw i32 1, i32 0
store i32 %a, i32* %c, align 4
%b = add nsw i32 2, i32 0
store i32 %b, i32* %d, align 4

// version 1
=> %a = add i32 1, i32 0
    st %a, i32* %c, 0
    %b = add i32 2, i32 0
    st %b, i32* %d, 0

// version 2
=> %a = add i32 1, i32 0
    %b = add i32 2, i32 0
    st %a, i32* %c, 0
    st %b, i32* %d, 0

// version 3
=> %b = add i32 2, i32 0
    st %b, i32* %d, 0
    %a = add i32 1, i32 0
    st %a, i32* %c, 0
```

2.3.2 DSA Form

```
for (int i = 0; i < 1000; i++) {
    b[i] = f(g(a[i]));
}
```

For the source program above, the following represent its SSA form at both the source code level and the LLVM IR level, respectively.

```
for (int i = 0; i < 1000; i++) {
    t = g(a[i]);
    b[i] = f(t);
}

%pi = alloca i32
store i32 0, i32* %pi
%i = load i32, i32* %pi
%cmp = icmp slt i32 %i, 1000
br i1 %cmp, label %true, label %end
true:
%a_idx = add i32 %i, i32 %a_addr
%val0 = load i32, i32* %a_idx
%t = call i64 %g(i32 %val0)
%val1 = call i64 %f(i32 %t)
%b_idx = add i32 %i, i32 %b_addr
```

(continues on next page)

¹⁶ Refer section 10.2.3 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

(continued from previous page)

```
store i32 %val1, i32* %b_idx
end:
```

The following represents the **DSA (Dynamic Single Assignment) form**.

```
for (int i = 0; i < 1000; i++) {
    t[i] = g(a[i]);
    b[i] = f(t[i]);
}
```

```
%pi = alloca i32
store i32 0, i32* %pi
%i = load i32, i32* %pi
%cmp = icmp slt i32 %i, 1000
br i1 %cmp, label %true, label %end
true:
%a_idx = add i32 %i, i32 %a_addr
%val0 = load i32, i32* %a_idx
%t_idx = add i32 %i, i32 %t_addr
%temp = call i64 %g(i32 %val0)
store i32 %temp, i32* %t_idx
%val1 = call i64 %f(i32 %temp)
%b_idx = add i32 %i, i32 %b_addr
store i32 %val1, i32* %b_idx
end:
```

In some internet video applications and multi-core (SMP) platforms, splitting *g()* and *f()* into two separate loops can improve performance.

DSA allows this transformation, whereas SSA does not. While extra analysis on *%temp* in SSA could reconstruct *%t_idx* and *%t_addr* as shown in the DSA form below, compiler transformations typically follow a high-to-low approach.

Additionally, LLVM IR already loses the *for* loop structure, even though part of the losted information can be reconstructed through further analysis.

For this reason, in this book—as well as in most compiler-related research—the discussion follows a high-to-low transformation premise. Otherwise, it would fall into the domain of **reverse engineering** in assemblers or compilers.

```
for (int i = 0; i < 1000; i++) {
    t[i] = g(a[i]);
}

for (int i = 0; i < 1000; i++) {
    b[i] = f(t[i]);
}
```

```
%pi = alloca i32
store i32 0, i32* %pi
%i = load i32, i32* %pi
%cmp = icmp slt i32 %i, 1000
br i1 %cmp, label %true, label %end
true:
%a_idx = add i32 %i, i32 %a_addr
```

(continues on next page)

(continued from previous page)

```
%val0 = load i32, i32* %a_idx
%t_idx = add i32 %i, i32 %t_addr
%temp = call i32 %g(i32 %val0)
store i32 %temp, i32* %t_idx
end:

%pi = alloca i32
store i32 0, i32* %pi
%i = load i32, i32* %pi
%cmp = icmp slt i32 %i, 1000
br i1 %cmp, label %true, label %end
true:
%t_idx = add i32 %i, i32 %t_addr
%temp = load i32, i32* %t_idx
%val1 = call i32 %f(i32 %temp)
%b_idx = add i32 %i, i32 %b_addr
store i32 %val1, i32* %b_idx
end:
```

Now, data dependencies exist only on $t[i]$ between “ $t[i] = g(a[i])$ ” and “ $b[i] = f(t[i])$ ” for each $i = (0..999)$.

As a result, the program can execute in various orders, offering significant parallel processing opportunities for **multi-core (SMP) systems and heterogeneous processors**.

For example, $g(x)$ can be executed on a **GPU**, while $f(x)$ runs on a **CPU**.

2.3.3 Three-Phase Design

This content and the following sub-section are adapted from the AOSA chapter on LLVM written by Chris Latner^{[Page 18, 10](#)}.

The most common design for a traditional static compiler (such as most C compilers) follows a three-phase structure, consisting of the front end, the optimizer, and the back end, as shown in Fig. 2.4.

The **front end** parses the source code, checks for errors, and constructs a language-specific Abstract Syntax Tree (AST) to represent the input code. The AST may then be converted into an intermediate representation for optimization, after which the **optimizer** and **back end** process the code.

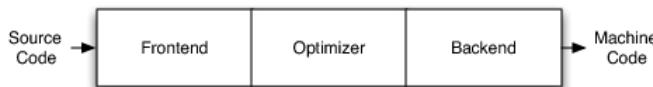


Fig. 2.4: Three Major Components of a Three Phase Compiler

The optimizer performs a wide range of transformations to improve code execution efficiency, such as eliminating redundant computations. It is generally independent of both the source language and the target architecture.

The back end, also known as the code generator, maps the optimized code onto the target instruction set. In addition to producing correct code, it is responsible for generating efficient code that leverages the unique features of the target architecture. Common components of a compiler back end include instruction selection, register allocation, and instruction scheduling.

This model applies equally well to interpreters and Just-In-Time (JIT) compilers. The Java Virtual Machine (JVM) is an example of this model, using Java bytecode as the interface between the front end and the optimizer.

The greatest advantage of this classical design becomes evident when a compiler supports multiple source languages or target architectures. If the compiler's optimizer uses a common intermediate representation, a front end can be written

for any language that compiles to this representation, and a back end can be developed for any target that compiles from it, as illustrated in Fig. 2.5.

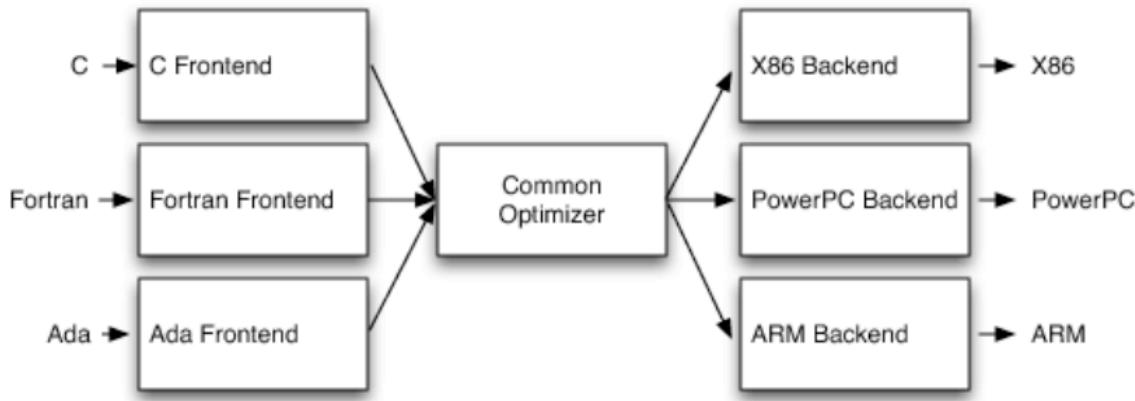


Fig. 2.5: Retargetability

With this design, porting the compiler to support a new source language (e.g., Algol or BASIC) requires implementing a new front end, while the existing optimizer and back end can be reused. If these components were not separated, adding a new source language would require rebuilding the entire compiler from scratch. Supporting N targets and M source languages would then necessitate developing $N * M$ compilers.

Another advantage of the three-phase design, which stems from its retargetability, is that the compiler can serve a broader range of programmers compared to one that supports only a single source language and target. For an open-source project, this translates to a larger community of potential contributors, leading to more enhancements and improvements.

This is why open-source compilers that cater to diverse communities, such as GCC, often generate better-optimized machine code than narrower compilers like FreePASCAL. In contrast, the quality of proprietary compilers depends directly on their development budget. For example, the Intel ICC compiler is widely recognized for producing high-quality machine code despite serving a smaller audience.

A final major benefit of the three-phase design is that the skills required to develop a front end differ from those needed for the optimizer and back end. By separating these concerns, “front-end developers” can focus on enhancing and maintaining their part of the compiler. While this is a social rather than a technical factor, it has a significant impact in practice—especially for open-source projects aiming to lower barriers to contribution.

The most critical aspect of this design is the **LLVM Intermediate Representation (IR)**, which serves as the compiler’s core code representation. LLVM IR is designed to support mid-level analysis and transformations commonly found in the optimization phase of a compiler.

It was created with several specific goals, including support for lightweight runtime optimizations, cross-function and interprocedural optimizations, whole-program analysis, and aggressive restructuring transformations. However, its most defining characteristic is that it is a first-class language with well-defined semantics.

To illustrate this, here is a simple example of an LLVM `.ll` file:

```

define i32 @add1(i32 %a, i32 %b) {
entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}
define i32 @add2(i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
  
```

(continues on next page)

(continued from previous page)

```

br i1 %tmp1, label %done, label %recurse
recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4
done:
  ret i32 %b
}

```

```

// Above LLVM IR corresponds to this C code, which provides two different ways to
// add integers:
unsigned add1(unsigned a, unsigned b) {
    return a+b;
}
// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}

```

As shown in this example, LLVM IR is a low-level, RISC-like virtual instruction set. Like a real RISC instruction set, it supports linear sequences of simple instructions such as *add*, *subtract*, *compare*, and *branch*.

These instructions follow a three-address format, meaning they take inputs and produce a result in a different register. LLVM IR supports labels and generally resembles an unusual form of assembly language.

Unlike most RISC instruction sets, LLVM IR is strongly typed and uses a simple type system (e.g., *i32* represents a 32-bit integer, and *i32*** is a pointer to a pointer to a 32-bit integer). Additionally, some machine-specific details are abstracted away.

For instance, the calling convention is handled through *call* and *ret* instructions with explicit arguments. Another key difference from machine code is that LLVM IR does not use a fixed set of named registers. Instead, it employs an infinite set of temporaries prefixed with *%*.

Beyond being a language, LLVM IR exists in three isomorphic forms:

- A **textual format** (as seen above).
- An **in-memory data structure** used by optimizations.
- A **compact binary “bitcode” format** stored on disk.

The LLVM project provides tools to convert between these forms:

- *llvm-as* assembles a textual *.ll* file into a *.bc* file containing bitcode.
- *llvm-dis* disassembles a *.bc* file back into a *.ll* file.

The intermediate representation (IR) of a compiler is crucial because it creates an ideal environment for optimizations. Unlike the front end and back end, the optimizer is not restricted to a specific source language or target machine.

However, it must effectively serve both. It should be easy for the front end to generate while remaining expressive enough to enable important optimizations for real hardware targets.

2.3.4 LLVM's Target Description Files: .td

The “mix and match” approach allows target authors to select components that best suit their architecture, enabling significant code reuse across different targets.

However, this introduces a challenge: each shared component must be capable of handling target-specific properties in a generic way. For instance, a shared register allocator must be aware of the register file of each target and the constraints that exist between instructions and their register operands.

LLVM addresses this challenge by requiring each target to provide a target description using a declarative domain-specific language, defined in a set of *.td* files. These files are processed by the *tblgen* tool to generate the necessary target-specific data structures.

The simplified build process for the x86 target is illustrated in Fig. 2.6.

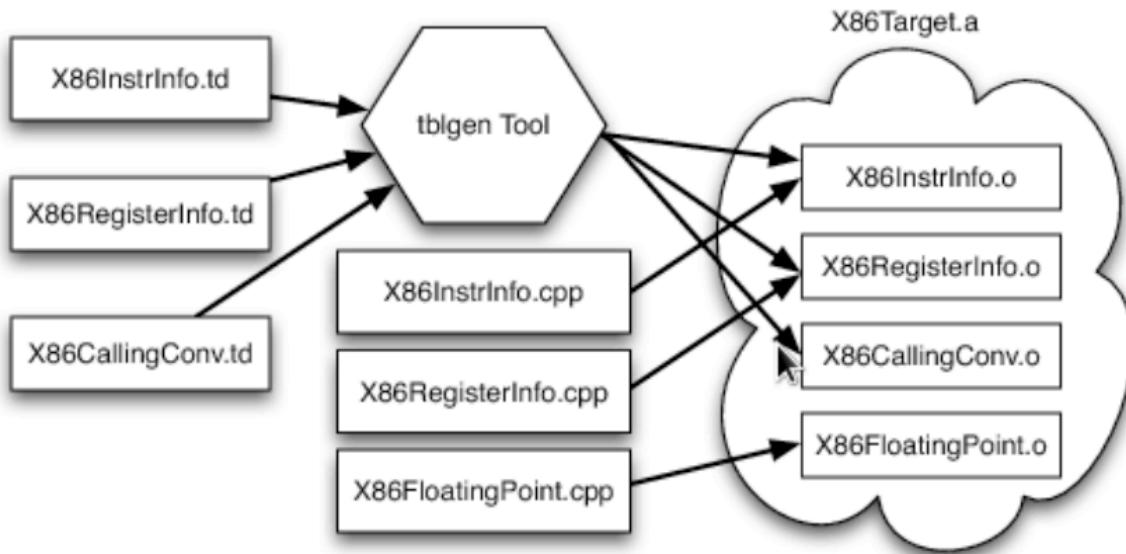


Fig. 2.6: Simplified x86 Target Definition

The different subsystems supported by *.td* files enable target authors to construct various components of their target architecture.

For example, the x86 backend defines a register class named “*GR32*”, which contains all 32-bit registers. In *.td* files, target-specific definitions are conventionally written in all capital letters. The definition is as follows:

```

def GR32 : RegisterClass<[i32], 32,
    [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
    R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
  
```

The language used in *.td* files is the Target (Hardware) Description Language, which allows LLVM backend compiler engineers to define the transformation from LLVM IR to machine instructions for their CPUs.

In the frontend, compiler development tools provide a **Parser Generator** for building compilers. In the backend, they offer a **Machine Code Generator** to facilitate instruction selection and code generation, as shown in Fig. 2.7 and Fig. 2.8.

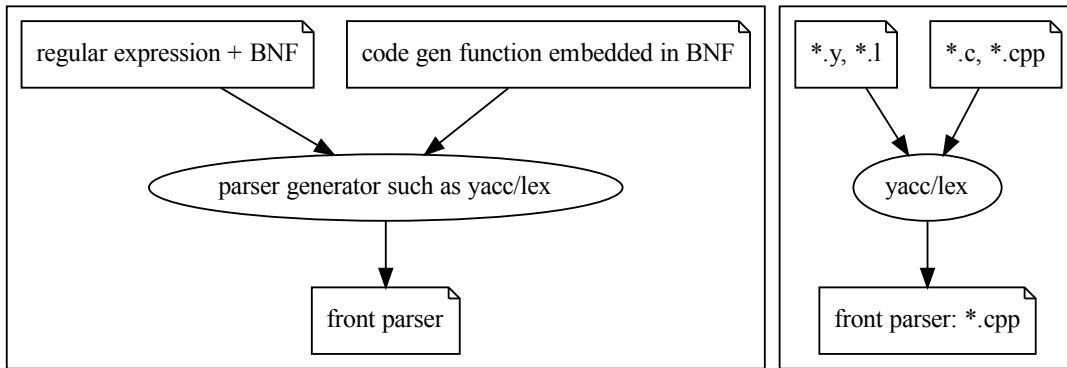


Fig. 2.7: Frontend TableGen Flow

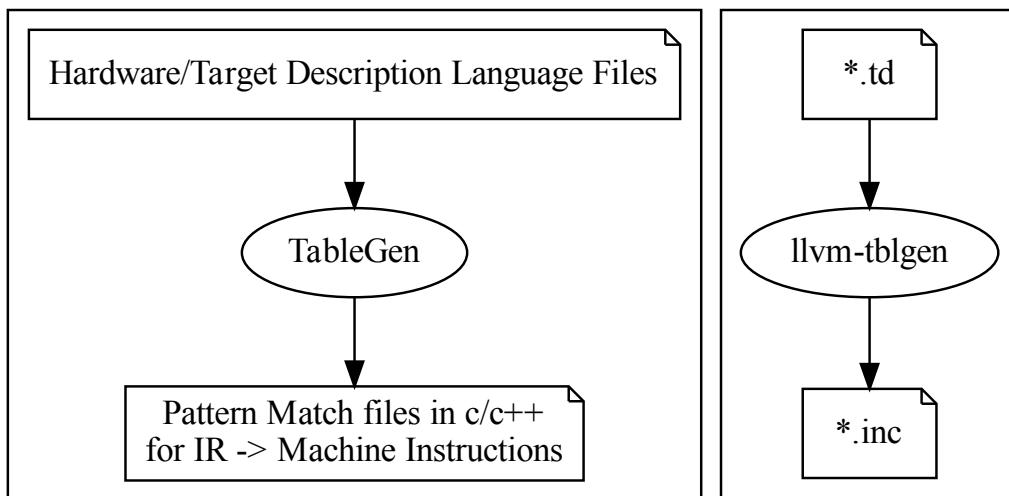


Fig. 2.8: Ivm TableGen Flow

2.3.5 LLVM Code Generation Sequence

Following diagram is from *tricore_llvm.pdf*.

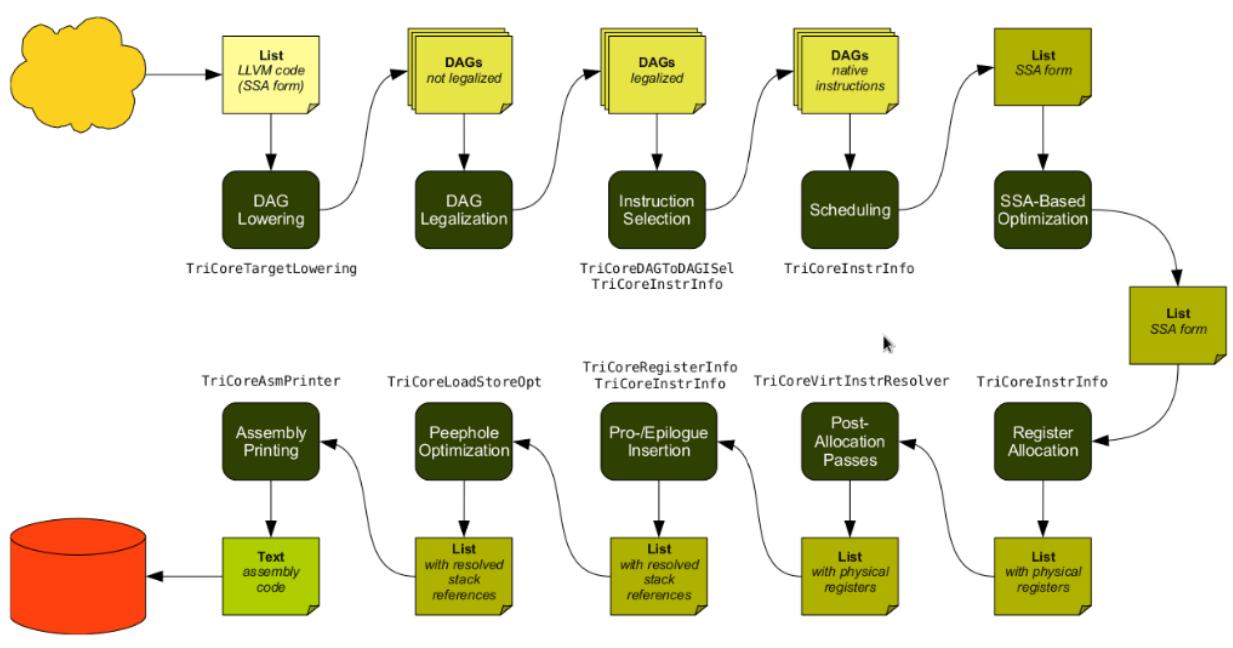


Fig. 2.9: *tricore_llvm.pdf*: **Code Generation Sequence** On the path from LLVM code to assembly code, numerous passes are executed, and several data structures are used to represent intermediate results.

LLVM is a **Static Single Assignment (SSA)**-based representation. It provides an infinite number of virtual registers that can hold values of primitive types, including integral, floating-point, and pointer values.

In LLVM's SSA representation, each operand is stored in a separate virtual register. Comments in LLVM IR are denoted by the ; symbol.

The following are examples of LLVM SSA instructions:

```

store i32 0, i32* %a ; store i32 type of 0 to virtual register %a, %a is
; pointer type which point to i32 value
store i32 %b, i32* %c ; store %b contents to %c point to, %b is i32 type virtual
; register, %c is pointer type which point to i32 value.
%a1 = load i32* %a ; load the memory value where %a point to and assign the
; memory value to %a1
%a3 = add i32 %a2, 1 ; add %a2 and 1 and save to %a3

```

We explain the code generation process below. If you are unfamiliar with the concepts, we recommend first reviewing Section 4.2 of *tricore_llvm.pdf*.

You may also refer to *The LLVM Target-Independent Code Generator*¹² and the *LLVM Language Reference Manual*^{Page 21, 13}. However, we believe that Section 4.2 of *tricore_llvm.pdf* provides sufficient information.

We suggest consulting the above web documents only if you still have difficulties understanding the material, even after reading this section and the next two sections on **DAG** and **Instruction Selection**.

1. Instruction Selection

¹² <http://llvm.org/docs/CodeGenerator.html>

```
// In this stage, the LLVM opcode is transformed into a machine opcode,
// but the operand remains an LLVM virtual operand.
    store i16 0, i16* %a // Store 0 of i16 type to the location pointed to by %a
=> st i16 0, i32* %a      // Use the Cpu0 backend instruction `st` instead of `store`.
```

2. Scheduling and Formation

```
// In this stage, instruction order is optimized for execution cycles
// or to reduce register pressure.
    st i32 %a, i16* %b, i16 5 // Store %a to *(%b + 5)
    st %b, i32* %c, i16 0
    %d = ld i32* %c

// The instruction order is rearranged. In RISC CPUs like MIPS,
// `ld %c` depends on the previous `st %c`, requiring a 1-cycle delay.
// This means `ld` cannot immediately follow `st`.
=> st %b, i32* %c, i16 0
    st i32 %a, i16* %b, i16 5
    %d = ld i32* %c, i16 0

// Without instruction reordering, a `nop` instruction must be inserted,
// adding an extra cycle. (In reality, MIPS dynamically schedules
// instructions and inserts `nop` between `st` and `ld` if necessary.)
    st i32 %a, i16* %b, i16 5
    st %b, i32* %c, i16 0
    nop
    %d = ld i32* %c, i16 0

// **Minimizing Register Pressure**
// Suppose `%c` remains live after the basic block, but `%a` and `%b` do not.
// Without reordering, at least 3 registers are required:
    %a = add i32 1, i32 0
    %b = add i32 2, i32 0
    st %a, i32* %c, 1
    st %b, i32* %c, 2

// The reordered version reduces register usage to 2 by allocating `%a`
// and `%b` in the same...

// Register allocation optimization
=> %a = add i32 1, i32 0
    st %a, i32* %c, 1
    %b = add i32 2, i32 0
    st %b, i32* %c, 2
```

3. SSA-Based Machine Code Optimization

For example, common subexpression elimination, as shown in the next section on **DAG**.

4. Register Allocation

Assign physical registers to virtual registers.

5. Prologue/Epilogue Code Insertion

Explained in the section **Add Prologue/Epilogue Functions**.

6. Late Machine Code Optimizations

Any “last-minute” peephole optimizations of the final machine code are applied in this phase. For example, replacing $x = x * 2$ with $x = x \ll 1$ for integer operands.

7. Code Emission

The final machine code is emitted. - For **static compilation**, the output is an assembly file. - For **JIT compilation**, machine instruction opcodes are written into memory.

The LLVM code generation sequence can also be viewed using:

```
llc -debug-pass=Structure
```

as shown below. The first four code generation stages from Fig. 2.9 appear in the ‘**DAG->DAG Pattern Instruction Selection**’ section of the llc -debug-pass=Structure output.

The order of **Peephole Optimizations** and **Prologue/Epilogue Insertion** differs between Fig. 2.9 and llc -debug-pass=Structure (marked with * in the output).

There is no need to be concerned about this, as LLVM is continuously evolving, and its internal sequence may change over time.

```
118-165-79-200:input Jonathan$ llc --help-hidden
OVERVIEW: llvm system compiler

USAGE: llc [options] <input bitcode>

OPTIONS:
...
-debug-pass           - Print PassManager debugging information
=None                - disable debug output
=Arguments            - print pass arguments to pass to 'opt'
=Structure            - print pass structure before run()
=Executions           - print pass name before it is executed
=Details              - print pass details when it is executed

118-165-79-200:input Jonathan$ llc -march=mips -debug-pass=Structure ch3.bc
...
Target Library Information
Target Transform Info
Data Layout
Target Pass Configuration
No Alias Analysis (always returns 'may' alias)
Type-Based Alias Analysis
Basic Alias Analysis (stateless AA impl)
Create Garbage Collector Module Metadata
Machine Module Information
Machine Branch Probability Analysis
ModulePass Manager
FunctionPass Manager
Preliminary module verification
Dominator Tree Construction
Module Verifier
Natural Loop Information
Loop Pass Manager
Canonicalize natural loops
```

(continues on next page)

(continued from previous page)

```
Scalar Evolution Analysis
Loop Pass Manager
    Canonicalize natural loops
    Induction Variable Users
    Loop Strength Reduction
Lower Garbage Collection Instructions
Remove unreachable blocks from the CFG
Exception handling preparation
Optimize for code generation
Insert stack protectors
Preliminary module verification
Dominator Tree Construction
Module Verifier
Machine Function Analysis
Natural Loop Information
Branch Probability Analysis
* MIPS DAG->DAG Pattern Instruction Selection
    Expand ISel Pseudo-instructions
    Tail Duplication
    Optimize machine instruction PHIs
    MachineDominator Tree Construction
    Slot index numbering
    Merge disjoint stack slots
    Local Stack Slot Allocation
    Remove dead machine instructions
    MachineDominator Tree Construction
    Machine Natural Loop Construction
    Machine Loop Invariant Code Motion
    Machine Common Subexpression Elimination
    Machine code sinking
* Peephole Optimizations
    Process Implicit Definitions
    Remove unreachable machine basic blocks
    Live Variable Analysis
    Eliminate PHI nodes for register allocation
    Two-Address instruction pass
    Slot index numbering
    Live Interval Analysis
    Debug Variable Analysis
    Simple Register Coalescing
    Live Stack Slot Analysis
    Calculate spill weights
    Virtual Register Map
    Live Register Matrix
    Bundle Machine CFG Edges
    Spill Code Placement Analysis
* Greedy Register Allocator
    Virtual Register Rewriter
    Stack Slot Coloring
    Machine Loop Invariant Code Motion
* Prologue/Epilogue Insertion & Frame Finalization
    Control Flow Optimizer
```

(continues on next page)

(continued from previous page)

```

Tail Duplication
Machine Copy Propagation Pass
* Post-RA pseudo instruction expansion pass
MachineDominator Tree Construction
Machine Natural Loop Construction
Post RA top-down list latency scheduler
Analyze Machine Code For Garbage Collection
Machine Block Frequency Analysis
Branch Probability Basic Block Placement
Mips Delay Slot Filler
Mips Long Branch
MachineDominator Tree Construction
Machine Natural Loop Construction
* Mips Assembly Printer
Delete Garbage Collector Information

```

- Since **Instruction Scheduling** and **Dead Code Elimination** affect **Register Allocation**, LLVM does not revisit earlier passes once a later pass is completed. **Register Allocation** occurs after **Instruction Scheduling**.

The passes from **Live Variable Analysis** to **Greedy Register Allocator** handle **Register Allocation**. More details on register allocation passes can be found here:¹⁴¹⁵.

2.3.6 LLVM vs. GCC in Structure

The official GCC documentation can be found here:¹⁷.

Table 2.10: clang vs gcc-frontend

frontend	clang	gcc-frontend ¹⁸
LANGUAGE	C/C++	C/C++
parsing	parsing	parsing
AST	clang-AST	GENERIC ¹⁹
optimization & codgen	clang-backend	gimplifier
IR	LLVM IR	GIMPLE ²⁰

Table 2.11: llvm vs gcc (kernal and target/backend)

backend	llvm	gcc
IR	LLVM IR	GIMPLE
transfer	optimziation & pass	optimization & plugins
DAG	DAG	RTL ²¹
codgen	tblgen for td	codgen for md ²²

Both **LLVM IR** and **GIMPLE** use SSA form.

¹⁴ <https://www.cs.cmu.edu/afs/cs/academic/class/15745-s16/www/lectures/L23-Register-Coalescing.pdf>

¹⁵ https://en.wikipedia.org/wiki/Register_allocation

¹⁷ https://en.wikipedia.org/wiki/GNU_Compiler_Collection

¹⁸ https://en.wikipedia.org/wiki/GNU_Compiler_Collection#Front_ends

¹⁹ <https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html>

²⁰ <https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html>

²¹ <https://gcc.gnu.org/onlinedocs/gccint/RTL.html>

²² <https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html#Machine-Desc>

LLVM IR was originally designed to be fully reusable across various tools, not just within the compiler itself. In contrast, the **GCC community** never intended for GIMPLE to be used beyond the compiler.

Richard Stallman actively resisted efforts to make GCC's IR more reusable to prevent third-party commercial tools from leveraging GCC frontends. As a result, **GIMPLE (GCC's IR)** was never designed to fully describe a compiled program.

For example, it lacks critical information such as the program's **call graph**, **type definitions**, **stack offsets**, and **alias information**²³.

2.3.7 LLVM Blog

A user may rely on a **null pointer** as a guard to ensure code correctness. However, **undef** values occur only during compiler optimizations²⁴.

If a user fails to explicitly bind a null pointer—either directly or indirectly—compilers like **LLVM** and **GCC** may interpret the null pointer as **undef**, leading to unexpected optimization behavior²⁵.

2.3.8 CFG (Control Flow Graph)

The SSA form can be represented using a **Control Flow Graph (CFG)** and optimized by analyzing it.

Each node in the graph represents a **basic block (BB)**—a straight-line sequence of code without any jumps or jump targets. A jump target always **starts** a basic block, while a jump **ends** one²⁶.

The following is an example of a **CFG**. **Jumps and branches always appear in the last statement of basic blocks (BBs)** as shown in Fig. 2.10.

Fig/llvmstructure/cfg-ex.cpp

```
int cfg_ex(int a, int b, int n)
{
    for (int i = 0; i <= n; i++) {
        if (a < b) {
            a = a + i;
            b = b - 1;
        }
        if (b == 0) {
            goto label_1;
        }
    }

label_1:
    switch (a) {
    case 10:
        a = a*a-b+2;
        a++;
        break;
    }
}
```

(continues on next page)

²³ <https://stackoverflow.com/questions/40799696/how-is-gcc-ir-different-from-llvm-ir/40802063>

²⁴ https://github.com/Jonathan2251/lbd/tree/master/References/null_pointer.cpp is an example.

²⁵ Dereferencing a NULL Pointer: contrary to popular belief, dereferencing a null pointer in C is undefined. It is not defined to trap, and if you mmap a page at 0, it is not defined to access that page. This falls out of the rules that forbid dereferencing wild pointers and the use of NULL as a sentinel, from <http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>. As link, https://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html. In this case, the developer forgot to call “set”, did not crash with a null pointer dereference, and their code broke when someone else did a debug build.

²⁶ https://en.wikipedia.org/wiki/Control-flow_graph

(continued from previous page)

```
return (a+b);
}
```

Fig/llvmstructure/cfg-ex.ll

```
define dso_local i32 @_Z6cfg_exiii(i32 signext %a, i32 signext %b, i32 signext %n) ↴
local_unnamed_addr nounwind {
entry:
%cmp.not23 = icmp slt i32 %n, 0
br i1 %cmp.not23, label %cleanup, label %for.body

for.cond:                                     ; preds = %for.body
%inc = add nuw i32 %i.026, 1
%exitcond.not = icmp eq i32 %i.026, %n
br i1 %exitcond.not, label %cleanup, label %for.body, !llvm.loop !2

for.body:                                      ; preds = %entry, %for.cond
%i.026 = phi i32 [ %inc, %for.cond ], [ 0, %entry ]
%a.addr.025 = phi i32 [ %a.addr.1, %for.cond ], [ %a, %entry ]
%b.addr.024 = phi i32 [ %b.addr.1, %for.cond ], [ %b, %entry ]
%cmp1 = icmp slt i32 %a.addr.025, %b.addr.024
%sub = sext i1 %cmp1 to i32
%b.addr.1 = add nsw i32 %b.addr.024, %sub
%add = select i1 %cmp1, i32 %i.026, i32 0
%a.addr.1 = add nsw i32 %add, %a.addr.025
%cmp2 = icmp eq i32 %b.addr.1, 0
br i1 %cmp2, label %cleanup, label %for.cond

cleanup:                                       ; preds = %for.cond, %for.body,
→%entry
%b.addr.2 = phi i32 [ %b, %entry ], [ 0, %for.body ], [ %b.addr.1, %for.cond ]
%a.addr.2 = phi i32 [ %a, %entry ], [ %a.addr.1, %for.body ], [ %a.addr.1, %for.
→cond ]
%cond = icmp eq i32 %a.addr.2, 10
%inc7 = sub i32 103, %b.addr.2
%spec.select = select i1 %cond, i32 %inc7, i32 %a.addr.2
%add8 = add nsw i32 %spec.select, %b.addr.2
ret i32 %add8
}

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.1"}
!2 = distinct !{!2, !3}
!3 = !{!"llvm.loop.mustprogress"}
```

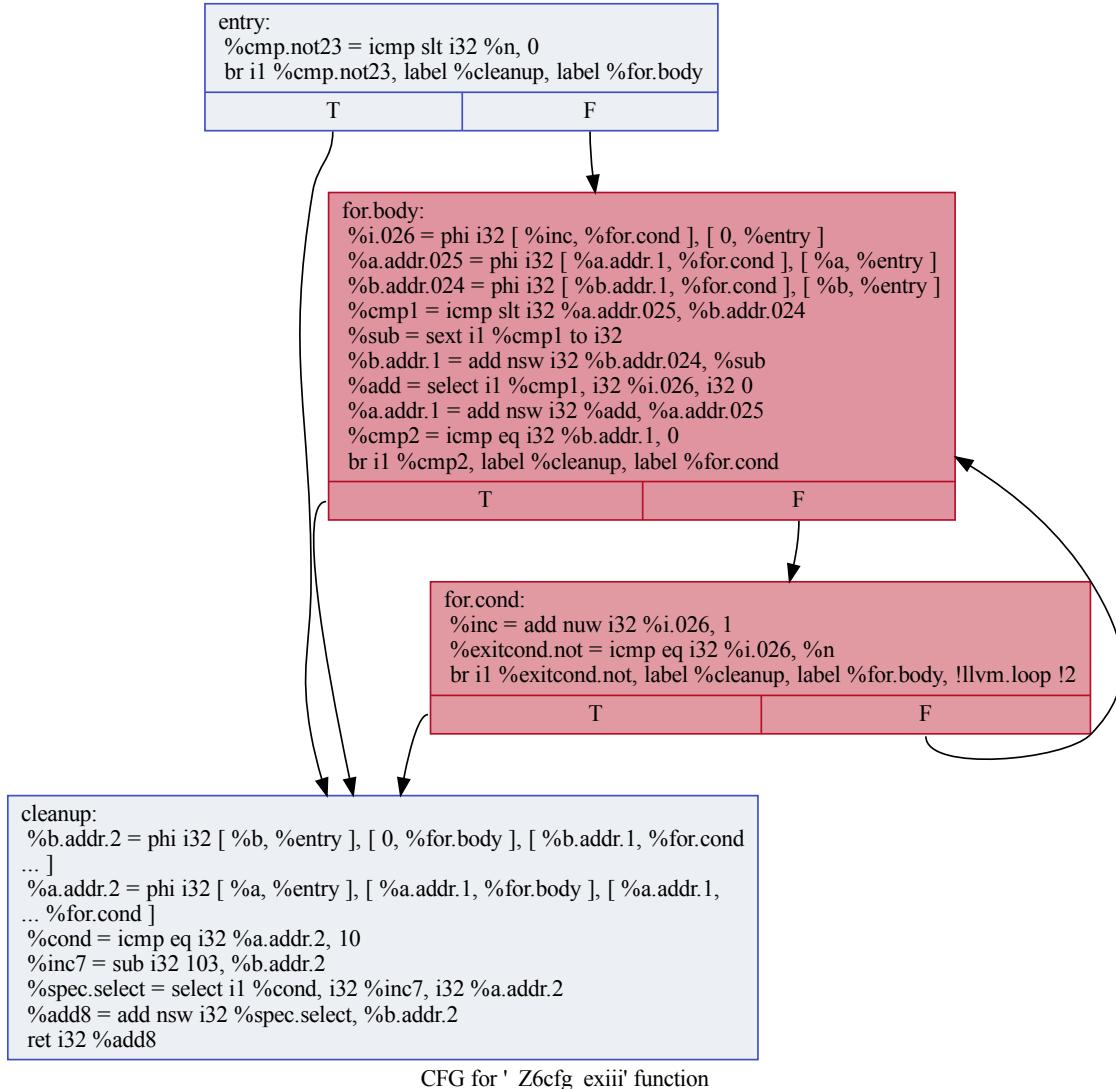


Fig. 2.10: CFG for cfg-ex.ll

2.3.9 DAG (Directed Acyclic Graph)

The SSA form within each **Basic Block (BB)** from the **Control Flow Graph (CFG)**, as discussed in the previous section, can be represented using a **Directed Acyclic Graph (DAG)**.

Many key **local optimization** techniques begin by transforming a basic block into a DAG²⁷.

For example, the basic block code and its corresponding DAG are illustrated in Fig. 2.11.

DAG and SSA allow instructions to have two destination virtual registers.

Assume the *ediv* operation performs integer division, storing the **quotient** in *a* and the **remainder** in *d*.

If only one destination register is used, the DAG may be simplified, as shown on the right in Fig. 2.11.

If *b* is not live at the exit of the block, we can apply **common subexpression elimination**, as demonstrated in the table below.

Table 2.12: Common Subexpression Elimination Process

Replace node b with node d	Replace b_0, c_0, d_0 with b, c, d
$a = b_0 + c_0$	$a = b + c$
$d = a - d_0$	$d = a - d$
$c = d + c$	$c = d + c$

After removing *b* and traversing the DAG from bottom to top (using **Depth-First In-Order Search** in a binary tree), we obtain the first column of the table above.

As you can imagine, **common subexpression elimination** can be applied both at the **IR** level and in **machine code**.

A **DAG** resembles a tree where **opcodes** are nodes, and **operands** (registers, constants, immediates, or offsets) are leaves. It can also be represented as a **prefix-ordered list** in a tree structure. For example, $(+ b, c)$ and $(+ b, 1)$ are IR DAG representations.

In addition to **DAG optimization**, **kill registers** are discussed in Section 8.5.5 of the compiler book²⁷. This optimization method is also applied in LLVM.

2.3.10 Instruction Selection

A major function of the backend is to **translate IR code into machine code** during **Instruction Selection**, as illustrated in Fig. 2.12.

For **machine instruction selection**, the best approach is to represent both **IR** and **machine instructions** as a **DAG**.

To simplify visualization, **register leaves** are omitted in Fig. 2.13.

The expression $r\overline{0} + r\overline{1}$ represents an **IR DAG** (used as a symbolic notation, not in LLVM SSA form). *ADD* is the corresponding machine instruction.

The **IR DAG** and **machine instruction DAG** can also be represented as lists. For example:

- **IR DAG lists:** $(+ r\overline{0}, r\overline{1})$ and $(- r\overline{0}, 1)$
- **Machine instruction DAG lists:** $(ADD r\overline{0}, r\overline{1})$ and $(SUBI r\overline{0}, 1)$

Now, let's examine the **ADDiu** instruction defined in *Cpu0InstrInfo.td*:

²⁷ Refer section 8.5 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

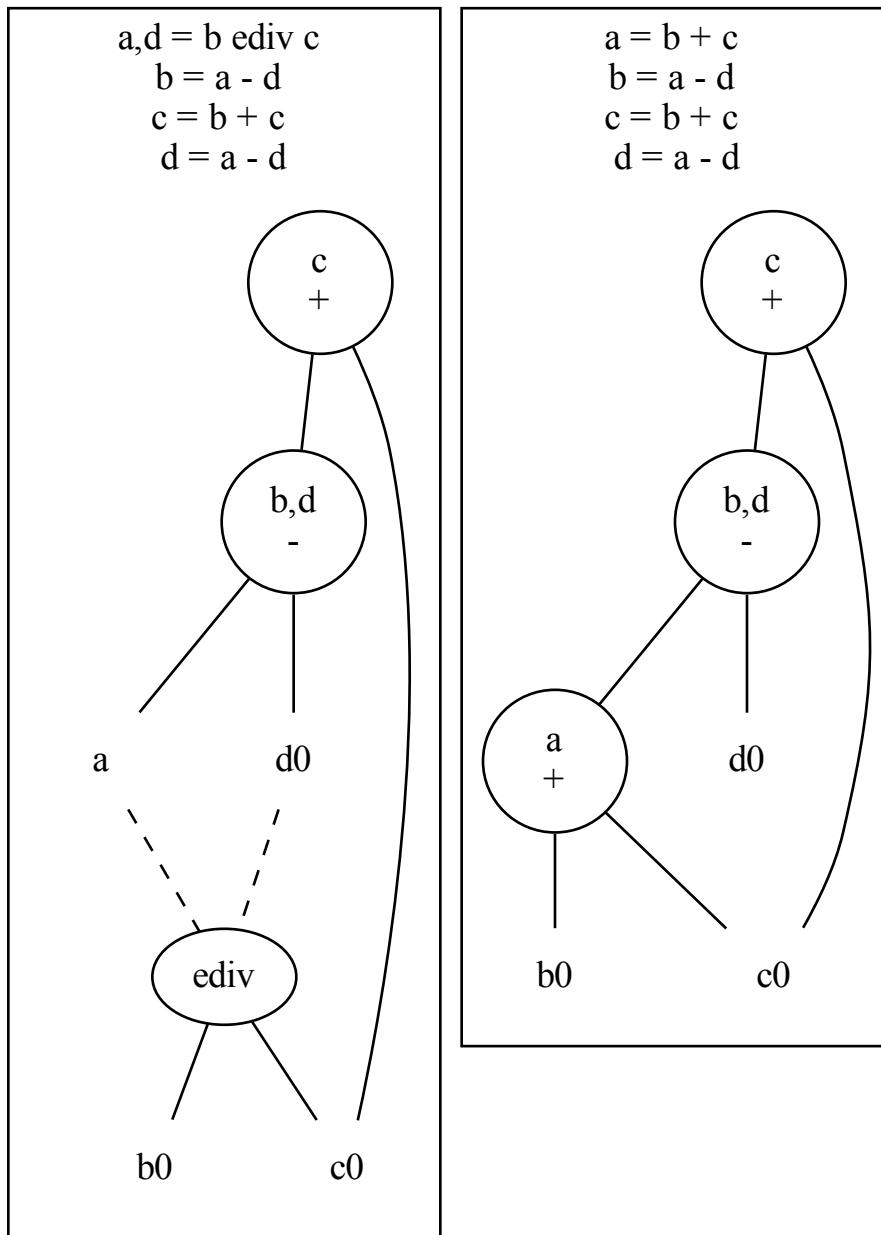


Fig. 2.11: The left example includes two destination registers, while the right has only one destination.

MOV	$r_d = r_s$	$r_d = r_s + 0$
MOV	$r_d = r_s$	ADD $r_d = r_s + r_0$
MOVI	$r_d = c$	ADDI $r_d = r_0 + c$

Fig. 2.12: IR and its corresponding machine instruction

Instruction Tree Patterns

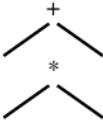
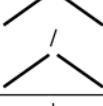
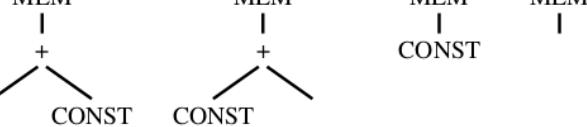
Name	Effect	Trees
—	r_i	TEMP
ADD	$r_i = r_j + r_k$	
MUL	$r_i = r_j \times r_k$	
SUB	$r_i = r_j - r_k$	
DIV	$r_i = r_j / r_k$	
ADDI	$r_i = r_j + c$	
SUBI	$r_i = r_j - c$	
LOAD	$r_i = M[r_j + c]$	

Fig. 2.13: Instruction DAG representation

Ibdex/chapters/Chapter2/Cpu0InstrFormats.td

```
//=====//  
// Format L instruction class in Cpu0 : <| opcode|ra|rb|cx|>  
//=====//  
  
class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,  
        InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>  
{  
    bits<4> ra;  
    bits<4> rb;  
    bits<16> imm16;  
  
    let Opcode = op;  
  
    let Inst{23-20} = ra;  
    let Inst{19-16} = rb;  
    let Inst{15-0} = imm16;  
}
```

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Node immediate fits as 16-bit sign extended on target immediate.  
// e.g. addi, andi  
def immSExt16 : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;  
  
// Arithmetic and logical instructions with 2 register operands.  
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,  
        Operand Od, PatLeaf imm_type, RegisterClass RC> :  
    FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),  
    !strconcat(instr_asm, "\t$ra, $rb, $imm16"),  
    [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {  
    let isReMaterializable = 1;  
}  
  
def ADDiu : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;
```

Fig. 2.14 illustrates how pattern matching works between the **IR node** *add* and the **instruction node** *ADDiu*, both defined in *Cpu0InstrInfo.td*.

In this example, the IR node “*add %a, 5*” is translated into “*addiu \$r1, 5*” after *%a* is allocated to register *\$r1* during the **register allocation** stage.

This translation occurs because the IR pattern (*set RC:\$ra, (OpNode RC:\$rb, imm_type:\$imm16)*) is defined for *ADDiu*, where the second operand is a **signed immediate** that matches *%a, 5*.

In addition to pattern matching, the *.td* file specifies the **assembly mnemonic** “*addiu*” and the **opcode** *0x09*.

Using this information, **LLVM TableGen** automatically generates both assembly instructions and binary encodings. The resulting **binary instruction** can be included in an **ELF object file**, which will be explained in a later chapter.

Similarly, **machine instruction DAG nodes** *LD* and *ST* are translated from the IR DAG nodes **load** and **store**.

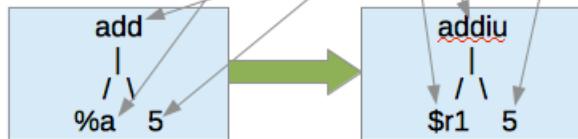
Note that in Fig. 2.14, *\$rb* represents a **virtual register** rather than a physical machine register. The details are further illustrated in Fig. 2.15.

```

class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
    Operand Od, PatLeaf imm_type, RegisterClass RC> :
    FL<op, (outs RC:$ra), (ins RC:$rb, Od:$imm16),
        !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
        [(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
    let isReMaterializable = 1;
}
def ADDiu : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

```

Tree



List

$$(\text{add } \%a, 5) \rightarrow (\text{addiu } \$r1, 5)$$

Fig. 2.14: Pattern matching for *ADDiu* instruction and IR node *add*

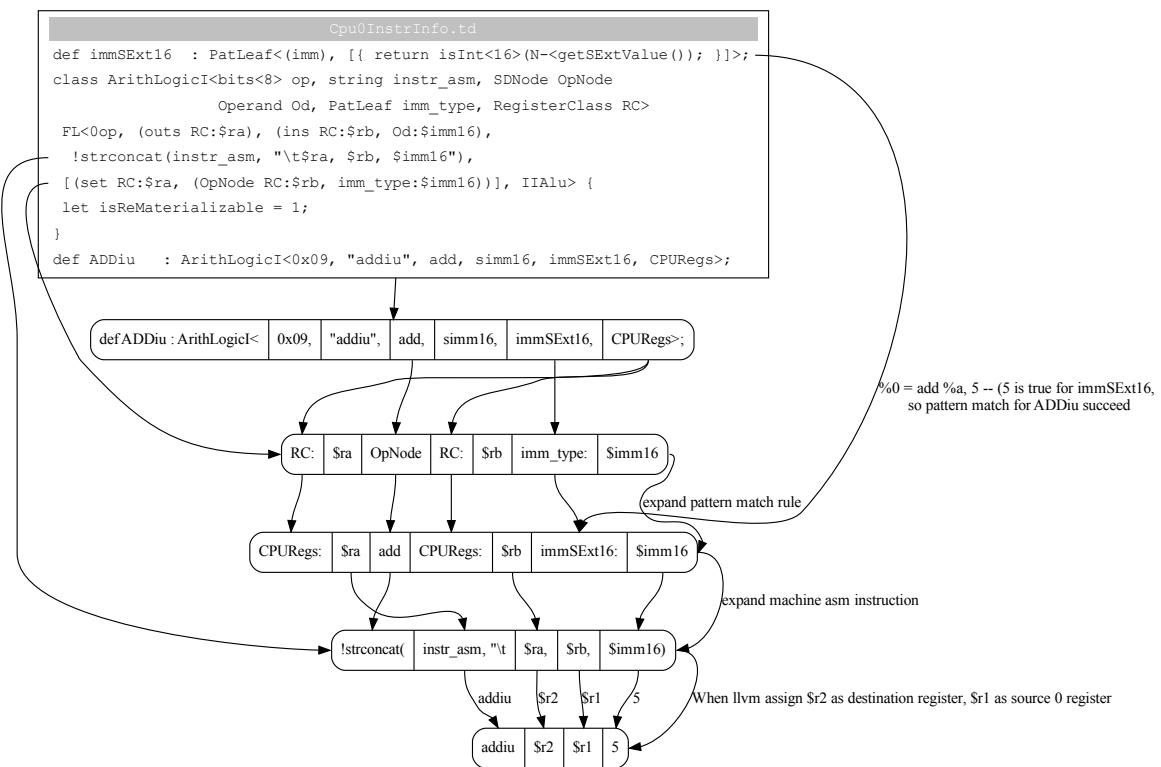


Fig. 2.15: Detailed pattern matching for *ADDiu* instruction and IR node *add*

During **DAG instruction selection**, the **leaf node must be a Data Node**. ***ADDiu*** follows the **L-type instruction format**, requiring the last operand to fit within a **16-bit signed range**.

To enforce this constraint, *Cpu0InstrInfo.td* defines a **PatLeaf** type *immSExt16*, allowing the LLVM system to recognize the valid operand range.

If the immediate value exceeds this range, “*isInt<16>(N->getSExtValue())*” returns *false*, and the **`ADDiu` pattern is not selected** during instruction selection.

Some CPUs and **floating-point units (FPUs)** include a **multiply-and-add** floating-point instruction, *fmadd*.

This instruction can be represented using a **DAG list** as follows: *(fadd (fmul ra, rc), rb)*.

To implement this, we define the **fmadd DAG pattern** in the instruction *.td* file as shown below:

```
def FMADDS : AForm_1<59, 29,
    (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
    "fmadds $FRT, $FRA, $FRC, $FRB",
    [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),
        F4RC:$FRB))];
```

Similar to *ADDiu*, the pattern *[(set F4RC:\$FRT, (fadd (fmul F4RC:\$FRA, F4RC:\$FRC), F4RC:\$FRB))]* includes both **fmul** and **fadd** nodes.

Now, consider the following **basic block notation IR** and **LLVM SSA IR** code:

```
d = a * c
e = d + b
...
```

```
%d = fmul %a, %c
%e = fadd %d, %b
...
```

The **Instruction Selection Process** will translate these two IR DAG nodes:

(fmul %a, %c) (fadd %d, %b)

into a **single** machine instruction DAG node:

*(**fmadd** %a, %c, %b)*

instead of translating them into **two separate** machine instruction nodes (****fmul**** and ****fadd****).

This optimization occurs **only if FMADDS appears before FMUL and FADD** in your *.td* file.

```
%e = fmadd %a, %c, %b
...
```

As you can see, the **IR notation representation** is easier to read than the **LLVM SSA IR** form.

For this reason, this notation is occasionally used in this book.

Now, consider the following **basic block code**:

```
a = b + c // in notation IR form
d = a - d
%e = fmadd %a, %c, %b // in llvm SSA IR form
```

We can apply [Fig. 2.6 Instruction Tree Patterns](#) to generate the following **machine code**:

```
load rb, M(sp+8); // assume b allocate in sp+8, sp is stack point register
load rc, M(sp+16);
add ra, rb, rc;
load rd, M(sp+24);
sub rd, ra, rd;
fmadd re, ra, rc, rb;
```

2.3.11 Caller and Callee Saved Registers

lbdex/input/ch9_caller_callee_save_registers.cpp

```
extern int add1(int x);

int callee()
{
    int t1 = 3;
    int result = add1(t1);
    result = result - t1;

    return result;
}
```

Running the **MIPS backend** with the above input will produce the following result:

```
JonathantekiiMac:input Jonathan$ ~/llvm/debug/build/bin/llc
-00 -march=mips -relocation-model=static -filetype=asm
ch9_caller_callee_save_registers.bc -o -
.text
.abicalls
.option pic0
.section      .mdebug.abi32,"",@progbits
.nan   legacy
.file   "ch9_caller_callee_save_registers.bc"
.text
.globl _Z6calleev
.align 2
.type   _Z6calleev,@function
.set    nomicromips
.set    nomips16
.ent    _Z6calleev
_Z6callerv:          # @_Z6callerv
.cfi_startproc
.frame $fp,32,$ra
.mask  0xc0000000,-4
.fmask 0x00000000,0
.set    noreorder
.set    nomacro
.set    noat
# BB#0:
    addiu  $sp, $sp, -32
$tmp0:
    .cfi_def_cfa_offset 32
```

(continues on next page)

(continued from previous page)

```

        sw      $ra, 28($sp)          # 4-byte Folded Spill
        sw      $fp, 24($sp)          # 4-byte Folded Spill

$tmp1:
        .cfi_offset 31, -4

$tmp2:
        .cfi_offset 30, -8
        move   $fp, $sp

$tmp3:
        .cfi_def_cfa_register 30
        addiu $1, $zero, 3
        sw     $1, 20($fp)    # store t1 to 20($fp)
        move   $4, $1
        jal    _Z4add1i
        nop
        sw     $2, 16($fp)    # $2 : the return vaule for fuction add1()
        lw     $1, 20($fp)    # load t1 from 20($fp)
        subu $1, $2, $1
        sw     $1, 16($fp)
        move   $2, $1    # move result to return register $2
        move   $sp, $fp
        lw     $fp, 24($sp)          # 4-byte Folded Reload
        lw     $ra, 28($sp)          # 4-byte Folded Reload
        addiu $sp, $sp, 32
        jr    $ra
        nop
        .set   at
        .set   macro
        .set   reorder
        .end   _Z6calleev

$func_end0:
        .size  _Z6calleev, ($func_end0)-_Z6calleev
        .cfi_endproc
    
```

Caller and callee saved registers definition as follows,

- If the **caller** wants to use **caller-saved registers** after calling a function, it must save their contents to memory before the function call and restore them afterward.
- If the **callee** wants to use **callee-saved registers**, it must save their contents to memory before using them and restore them before returning.

According to the definition above, if a register is **not** callee-saved, then it must be **caller-saved**, since the callee does not restore it and its value may change after the function call.

Thus, **MIPS** only defines **callee-saved registers** in *MipsCallingConv.td*, which can be found in *CSR_O32_SaveList* of *MipsGenRegisterInfo.inc* for the default ABI.

From the assembly output, MIPS allocates the **t1** variable to register **\$1**, which does **not** need to be spilled because **\$1** is a **caller-saved register**.

On the other hand, **\$ra** is a **callee-saved register**, so it is spilled at the beginning of the assembly output, as *jal* uses the **\$ra** register.

For **Cpu0**, the **\$lr** register corresponds to MIPS **\$ra**. Thus, the function *setAliasRegs(MF, SavedRegs, Cpu0::LR)* is called in *determineCalleeSaves()* within *Cpu0SEFrameLowering.cpp* when a function calls another function.

2.3.12 Live-In and Live-Out Registers

As seen in the previous subsection, `$ra` is a **live-in** register because the return address is determined by the caller.

Similarly, `$2` is a **live-out** register since the function's return value is stored in this register. The caller retrieves the result by reading `$2` directly, as noted in the previous example.

By marking **live-in** and **live-out** registers, the backend provides LLVM's **middle layer** with information to eliminate redundant variable access instructions.

LLVM applies **DAG analysis**, as discussed in the previous subsection, to perform this optimization.

Since **C supports separate compilation**, the **live-in** and **live-out** information from the backend offers additional optimization opportunities to LLVM.

LLVM provides the function `addLiveIn()` to mark a **live-in register**, but it does **not** offer a corresponding `addLiveOut()` function.

Instead, the **MIPS backend** marks **live-out** registers by using:

$DAG = DAG.getCopyToReg(..., \$2, ...)$

and then returning the modified **DAG**, as all local variables cease to exist after the function exits.

2.4 Create Cpu0 Backend

From this point onward, the **Cpu0 backend** will be created **step by step from scratch**.

To help readers understand the **backend structure**, the Cpu0 example code can be generated **chapter by chapter** using the command provided here¹¹.

The **Cpu0 example code** (``lbdex``) can be found near the bottom left of this website or downloaded from:

<http://jonathan2251.github.io/lbd/lbdex.tar.gz>

2.4.1 Cpu0 Backend Machine ID and Relocation Records

To create a **new backend**, several files in `<<llvm root dir>>` must be modified.

The required modifications include adding both the **machine ID and name**, as well as defining **relocation records**.

The **ELF Support** chapter provides an introduction to **relocation records**.

The following files are modified to **add the Cpu0 backend**:

Ibdex/llvm/modify/llvm/config-ix.cmake

```
...
elseif (LLVM_NATIVE_ARCH MATCHES "cpu0")
    set(LLVM_NATIVE_ARCH Cpu0)
...
```

Ibdex/llvm/modify/llvm/CMakeLists.txt

```
set(LLVM_ALL_TARGETS
...
Cpu0)
```

(continues on next page)

¹¹ <http://jonathan2251.github.io/lbd/doc.html#generate-cpu0-document>

(continued from previous page)

```
...
}
```

Ibdex/llvm/modify/llvm/include/llvm/ADT/Triple.h

```
...
#ifndef mips
#ifndef cpu0
...
class Triple {
public:
    enum ArchType {
        ...
        cpu0,           // For Tutorial Backend Cpu0
        cpu0el,
        ...
    };
    ...
}

```

Ibdex/llvm/modify/llvm/include/llvm/Object/ELFObjectFile.h

```
...
template <class ELFT>
StringRef ELFObjectFile<ELFT>::getFileFormatName() const {
    switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
        case ELF::ELFCLASS32:
            switch (EF.getHeader()->e_machine) {
                ...
                case ELF::EM_CPU0:           // llvm-objdump -t -r
                    return "ELF32-cpu0";
                ...
            }
        ...
    }
}

template <class ELFT>
unsigned ELFObjectFile<ELFT>::getArch() const {
    bool IsLittleEndian = ELFT::TargetEndianness == support::little;
    switch (EF.getHeader()->e_machine) {
        ...
        case ELF::EM_CPU0: // llvm-objdump -t -r
            switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
                case ELF::ELFCLASS32:
                    return IsLittleEndian ? Triple::cpu0el : Triple::cpu0;
                default:
                    report_fatal_error("Invalid ELFCLASS!");
            }
        ...
    }
}

```

Ibdex/llvm/modify/llvm/include/llvm/Support/ELF.h

```

enum {
    ...
    EM_CPU0          = 999 // Document LLVM Backend Tutorial Cpu0
};

...
// Cpu0 Specific e_flags
enum {
    EF_CPU0_NOREORDER = 0x00000001, // Don't reorder instructions
    EF_CPU0_PIC      = 0x00000002, // Position independent code
    EF_CPU0_ARCH_32   = 0x50000000, // CPU032 instruction set per linux not elf.h
    EF_CPU0_ARCH     = 0xf0000000 // Mask for applying EF_CPU0_ARCH_ variant
};

// ELF Relocation types for Mips
enum {
#include "ELFRelocs/Cpu0.def"
};
...

```

Ibdex/llvm/modify/llvm/lib/MC/MCSubtargetInfo.cpp

```

bool Cpu0DisableUnreconginizedMessage = false;

void MCSSubtargetInfo::InitMCProcessorInfo(StringRef CPU, StringRef FS) {
    #if 1 // Disable reconginized processor message. For Cpu0
    if (TargetTriple.getArch() == llvm::Triple::cpu0 ||
        TargetTriple.getArch() == llvm::Triple::cpu0el)
        Cpu0DisableUnreconginizedMessage = true;
    #endif
    ...
}

...
const MCSchedModel &MCSSubtargetInfo::getSchedModelForCPU(StringRef CPU) const {
    ...
    #if 1 // Disable reconginized processor message. For Cpu0
    if (TargetTriple.getArch() != llvm::Triple::cpu0 &&
        TargetTriple.getArch() != llvm::Triple::cpu0el)
        #endif
    ...
}

```

Ibdex/llvm/modify/llvm/lib/MC/SubtargetFeature.cpp

```

extern bool Cpu0DisableUnreconginizedMessage; // For Cpu0
...
FeatureBitset
SubtargetFeatures::ToggleFeature(FeatureBitset Bits, StringRef Feature,
                                ArrayRef<SubtargetFeatureKV> FeatureTable) {
    ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0

```

(continues on next page)

(continued from previous page)

```

...
}

FeatureBitset
SubtargetFeatures::ApplyFeatureFlag(FeatureBitset Bits, StringRef Feature,
                                    ArrayRef<SubtargetFeatureKV> FeatureTable) {
    ...
    if (!Cpu0DisableUnrecognizedMessage) // For Cpu0
    ...
}

FeatureBitset
SubtargetFeatures::getFeatureBits(StringRef CPU,
                                  ArrayRef<SubtargetFeatureKV> CPUPTable,
                                  ArrayRef<SubtargetFeatureKV> FeatureTable) {
    ...
    if (!Cpu0DisableUnrecognizedMessage) // For Cpu0
    ...
}

```

lib/object/ELF.cpp

```

...
StringRef getELFRelocationTypeName(uint32_t Machine, uint32_t Type) {
    switch (Machine) {
        ...
        case ELF::EM_CPU0:
            switch (Type) {
#include "llvm/Support/ELFRelocs/Cpu0.def"
                default:
                    break;
                }
                break;
        ...
    }
}
```

include/llvm/Support/ELFRelocs/Cpu0.def

```

#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif

ELF_RELOC(R_CPU0_NONE, 0)
ELF_RELOC(R_CPU0_32, 2)
ELF_RELOC(R_CPU0_HI16, 5)
ELF_RELOC(R_CPU0_LO16, 6)
ELF_RELOC(R_CPU0_GPREL16, 7)
ELF_RELOC(R_CPU0_LITERAL, 8)
ELF_RELOC(R_CPU0_GOT16, 9)

```

(continues on next page)

(continued from previous page)

ELF_RELOC(R_CPU0_PC16,	10)
ELF_RELOC(R_CPU0_CALL16,	11)
ELF_RELOC(R_CPU0_GPREL32,	12)
ELF_RELOC(R_CPU0_PC24,	13)
ELF_RELOC(R_CPU0_GOT_HI16,	22)
ELF_RELOC(R_CPU0_GOT_LO16,	23)
ELF_RELOC(R_CPU0_RELGOT,	36)
ELF_RELOC(R_CPU0_TLS_GD,	42)
ELF_RELOC(R_CPU0_TLS_LDM,	43)
ELF_RELOC(R_CPU0_TLS_DTP_HI16,	44)
ELF_RELOC(R_CPU0_TLS_DTP_LO16,	45)
ELF_RELOC(R_CPU0_TLS_GOTTPREL,	46)
ELF_RELOC(R_CPU0_TLS_TPREL32,	47)
ELF_RELOC(R_CPU0_TLS_TP_HI16,	49)
ELF_RELOC(R_CPU0_TLS_TP_LO16,	50)
ELF_RELOC(R_CPU0_GLOB_DAT,	51)
ELF_RELOC(R_CPU0_JUMP_SLOT,	127)

Ibdex/llvm/modify/llvm/lib/Support/Triple.cpp

```
const char *Triple::getArchTypeName(ArchType Kind) {
    switch (Kind) {
    ...
    case cpu0:      return "cpu0";
    case cpu0el:    return "cpu0el";
    ...
}
...
const char *Triple::getArchTypePrefix(ArchType Kind) {
    switch (Kind) {
    ...
    case cpu0:
    case cpu0el:    return "cpu0";
    ...
}
...
Triple::ArchType Triple::getArchTypeForLLVMName(StringRef Name) {
    return StringSwitch<Triple::ArchType>(Name)
    ...
    .Case("cpu0", cpu0)
    .Case("cpu0el", cpu0el)
    ...
}
...
static Triple::ArchType parseArch(StringRef ArchName) {
    return StringSwitch<Triple::ArchType>(ArchName)
    ...
    .Cases("cpu0", "cpu0eb", "cpu0allegrex", Triple::cpu0)
    .Cases("cpu0el", "cpu0allegrexel", Triple::cpu0el)
    ...
}
```

(continues on next page)

(continued from previous page)

```

}

...
static Triple::ObjectFormatType getDefaultFormat (const Triple &T) {
    ...
    case Triple::cpu0:
    case Triple::cpu0el:
    ...
}
...
static unsigned getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
    switch (Arch) {
        ...
        case llvm::Triple::cpu0:
        case llvm::Triple::cpu0el:
        ...
        return 32;
    }
}
...
Triple Triple::get32BitArchVariant() const {
    Triple T(*this);
    switch (getArch()) {
        ...
        case Triple::cpu0:
        case Triple::cpu0el:
        ...
        // Already 32-bit.
        break;
    }
    return T;
}
}

```

2.4.2 Creating the Initial Cpu0 .td Files

As discussed in the previous section, LLVM uses **target description files** (*.td* files) to define various components of a target's backend.

For example, these *.td* files may describe:

- A target's **register set**
- Its **instruction set**
- **Instruction scheduling details**
- **Calling conventions**

When the backend is compiled, LLVM's **TableGen** tool translates these *.td* files into **C++ source code**, which is then written to *.inc* files.

For more details on how to use **TableGen**, please refer to³².

Each backend has its own *.td* files to define target-specific information. These files have a **C++-like syntax**.

For **Cpu0**, the primary target description file is **Cpu0Other.td**, as shown below:

³² <http://llvm.org/docs/TableGen/index.html>

Ibdex/chapters/Chapter2/Cpu0.td

```
//===== Cpu0.td - Describe the Cpu0 Target Machine -----*- tablegen -*---//  
//  
//          The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
  
// Without this will have error: 'cpu032I' is not a recognized processor for  
// this target (ignoring processor)  
//===== //  
// Cpu0 Subtarget features  
//===== //  
  
def FeatureChapter3_1 : SubtargetFeature<"ch3_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter3_2 : SubtargetFeature<"ch3_2", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter3_3 : SubtargetFeature<"ch3_3", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter3_4 : SubtargetFeature<"ch3_4", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter3_5 : SubtargetFeature<"ch3_5", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter4_1 : SubtargetFeature<"ch4_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter4_2 : SubtargetFeature<"ch4_2", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter5_1 : SubtargetFeature<"ch5_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter6_1 : SubtargetFeature<"ch6_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter7_1 : SubtargetFeature<"ch7_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter8_1 : SubtargetFeature<"ch8_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter8_2 : SubtargetFeature<"ch8_2", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter9_1 : SubtargetFeature<"ch9_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter9_2 : SubtargetFeature<"ch9_2", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter9_3 : SubtargetFeature<"ch9_3", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter10_1 : SubtargetFeature<"ch10_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter11_1 : SubtargetFeature<"ch11_1", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;  
def FeatureChapter11_2 : SubtargetFeature<"ch11_2", "HasChapterDummy", "true",  
                         "Enable Chapter instructions.">;
```

(continues on next page)

(continued from previous page)

```

def FeatureChapter12_1 : SubtargetFeature<"ch12_1", "HasChapterDummy", "true",
                         "Enable Chapter instructions.">;
def FeatureChapterAll : SubtargetFeature<"chall", "HasChapterDummy", "true",
                         "Enable Chapter instructions.",
                         [FeatureChapter3_1, FeatureChapter3_2,
                          FeatureChapter3_3, FeatureChapter3_4,
                          FeatureChapter3_5,
                          FeatureChapter4_1, FeatureChapter4_2,
                          FeatureChapter5_1, FeatureChapter6_1,
                          FeatureChapter7_1, FeatureChapter8_1,
                          FeatureChapter8_2, FeatureChapter9_1,
                          FeatureChapter9_2, FeatureChapter9_3,
                          FeatureChapter10_1,
                          FeatureChapter11_1, FeatureChapter11_2,
                          FeatureChapter12_1]>;

def FeatureCmp : SubtargetFeature<"cmp", "HasCmp", "true",
                  "Enable 'cmp' instructions.">;
def FeatureSlt : SubtargetFeature<"slt", "HasSlt", "true",
                  "Enable 'slt' instructions.">;
def FeatureCpu032I : SubtargetFeature<"cpu032I", "Cpu0ArchVersion",
                      "Cpu032I", "Cpu032I ISA Support",
                      [FeatureCmp, FeatureChapterAll]>;
def FeatureCpu032II : SubtargetFeature<"cpu032II", "Cpu0ArchVersion",
                      "Cpu032II", "Cpu032II ISA Support (slt)",
                      [FeatureCmp, FeatureSlt, FeatureChapterAll]>

//=====
// Calling Conv, Instruction Descriptions
//=====

include "Cpu0Schedule.td"
include "Cpu0InstrInfo.td"

def Cpu0InstrInfo : InstrInfo;
//=====
// Cpu0 processors supported.
//=====

class Proc<string Name, list<SubtargetFeature> Features>
: Processor<Name, Cpu0GenericItineraries, Features>

def : Proc<"cpu032I", [FeatureCpu032I]>;
def : Proc<"cpu032II", [FeatureCpu032II]>;
// Above make Cpu0GenSubtargetInfo.inc set feature bit as the following order
// enum {
//     FeatureCmp = 1ULL << 0,
//     FeatureCpu032I = 1ULL << 1,
//     FeatureCpu032II = 1ULL << 2,
//     FeatureSlt = 1ULL << 3
// };

```

(continues on next page)

(continued from previous page)

```
// Will generate Cpu0GenAsmWrite.inc included by Cpu0InstPrinter.cpp, contents
// as follows,
// void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {...}
// const char *Cpu0InstPrinter::getRegisterName(unsigned RegNo) {...}
def Cpu0 : Target {
// def Cpu0InstrInfo : InstrInfo as before.
let InstructionSet = Cpu0InstrInfo;
}
```

Ibdex/chapters/Chapter2/Cpu0Other.td

```
===== Cpu0Other.td - Describe the Cpu0 Target Machine -----*- tablegen -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====-----=====
// This is the top level entry point for the Cpu0 target.
//=====-----=====

//=====-----=====
// Target-independent interfaces
//=====-----=====

include "llvm/Target/Target.td"

//=====-----=====
// Target-dependent interfaces
//=====-----=====

include "Cpu0RegisterInfo.td"
include "Cpu0RegisterInfoGPROutForOther.td" // except AsmParser
include "Cpu0.td"
```

Cpu0Other.td and *Cpu0.td* include several other *.td* files.

Cpu0RegisterInfo.td (shown below) describes the **Cpu0 register set**.

In this file, each register is assigned a name. For example, `def PC` defines a register named **PC**.

In addition to **register definitions**, this file also defines **register classes**. A target may have multiple register classes, such as: - **CPUREgs** - **SR** - **C0Regs** - **GPROut**

The **GPROut** register class, defined in *Cpu0RegisterInfoGPROutForOther.td*, includes all **CPUREgs** **except SW**, ensuring that **SW** is **not allocated** as an output register during the **register allocation stage**.

Ibdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
//===== Cpu0RegisterInfo.td - Cpu0 Register defs -----*- tablegen -*---//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
  
//===== //  
// Declarations that describe the CPU0 register file  
//===== //  
  
// We have banks of 16 registers each.  
class Cpu0Reg<bits<16> Enc, string n> : Register<n> {  
    // For tablegen(... -gen-emitter) in CMakeLists.txt  
    let HWEncoding = Enc;  
  
    let Namespace = "Cpu0";  
}  
  
// Cpu0 CPU Registers  
class Cpu0GPRReg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;  
  
// Co-processor 0 Registers  
class Cpu0C0Reg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;  
  
//===== //  
//@Registers  
//===== //  
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"  
//@ All registers definition  
let Namespace = "Cpu0" in {  
    //@ General Purpose Registers  
    def ZERO : Cpu0GPRReg<0, "zero">, DwarfRegNum<[0]>;  
    def AT : Cpu0GPRReg<1, "1">, DwarfRegNum<[1]>;  
    def V0 : Cpu0GPRReg<2, "2">, DwarfRegNum<[2]>;  
    def V1 : Cpu0GPRReg<3, "3">, DwarfRegNum<[3]>;  
    def A0 : Cpu0GPRReg<4, "4">, DwarfRegNum<[4]>;  
    def A1 : Cpu0GPRReg<5, "5">, DwarfRegNum<[5]>;  
    def T9 : Cpu0GPRReg<6, "t9">, DwarfRegNum<[6]>;  
    def T0 : Cpu0GPRReg<7, "7">, DwarfRegNum<[7]>;  
    def T1 : Cpu0GPRReg<8, "8">, DwarfRegNum<[8]>;  
    def S0 : Cpu0GPRReg<9, "9">, DwarfRegNum<[9]>;  
    def S1 : Cpu0GPRReg<10, "10">, DwarfRegNum<[10]>;  
    def GP : Cpu0GPRReg<11, "gp">, DwarfRegNum<[11]>;  
    def FP : Cpu0GPRReg<12, "fp">, DwarfRegNum<[12]>;  
    def SP : Cpu0GPRReg<13, "sp">, DwarfRegNum<[13]>;  
    def LR : Cpu0GPRReg<14, "lr">, DwarfRegNum<[14]>;  
    def SW : Cpu0GPRReg<15, "sw">, DwarfRegNum<[15]>;  
    // def MAR : Register< 16, "mar">, DwarfRegNum<[16]>;
```

(continues on next page)

(continued from previous page)

```
// def MDR : Register< 17, "mdr">, DwarfRegNum<[17]>;
def PC : Cpu0C0Reg<0, "pc">, DwarfRegNum<[20]>;
def EPC : Cpu0C0Reg<1, "epc">, DwarfRegNum<[21]>;
}

//=====
//@Register Classes
//=====

def CPUREgs : RegisterClass<"Cpu0", [i32], 32, (add
    // Reserved
    ZERO, AT,
    // Return Values and Arguments
    V0, V1, A0, A1,
    // Not preserved across procedure calls
    T9, T0, T1,
    // Callee save
    S0, S1,
    // Reserved
    GP, FP,
    SP, LR, SW)>;

//@Status Registers class
def SR : RegisterClass<"Cpu0", [i32], 32, (add SW)>;

//@Co-processor 0 Registers class
def C0Regs : RegisterClass<"Cpu0", [i32], 32, (add PC, EPC)>;
```

Index/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

```
//=====
// Register Classes
//=====

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPUREgs, SW))>;
```

In C++, a **class** typically defines a structure to organize data and functions, while **definitions** allocate memory for specific instances of the class.

For example:

```
class Date { // declare Date
    int year, month, day;
};

Date birthday; // define birthday, an instance of Date
```

The class **Date** has the members **year**, **month**, and **day**, but these do not yet belong to an actual object.

By defining an instance of **Date** called **birthday**, memory is allocated for a specific object, allowing you to set its **year**,

month, and **day**.

In `.td` files, a **class** describes the **structure** of how data is laid out, while **definitions** act as **specific instances** of the class.

If you refer back to the `Cpu0RegisterInfo.td` file, you will see a class called **Cpu0Reg**, which is derived from the **Register** class provided by **LLVM**. `Cpu0Reg` **inherits all fields** from the `Register` class.

The statement “**let HWEncoding = Enc**” assigns the field `HWEncoding` from the parameter `Enc`.

Since **Cpu0 reserves 4 bits for 16 registers** in its instruction format, the assigned value range is **0 to 15**.

Once values between `0` and `15` are assigned to `HWEncoding`, the **backend register number** is determined using **LLVM's register class functions**, as **TableGen** automatically sets this number.

The ``def`` keyword is used to create instances of a class.

In the following line, the `ZERO` register is defined as a member of the **Cpu0GPRReg** class:

```
def ZERO : Cpu0GPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
```

The **def ZERO** statement defines the name of this register. The parameters `<0, "ZERO">` are used to create this specific instance of the **Cpu0GPRReg** class.

Thus, the field `Enc` is set to `0`, and the string `n` is set to “`ZERO`”.

Since this register exists in the **Cpu0** namespace, it can be referenced in the backend C++ code using **Cpu0::ZERO**.

Overriding Values with *let* Expressions

The ``let`` expressions allow overriding values initially defined in a superclass.

For example, `let Namespace = "Cpu0"` in the **Cpu0Reg** class overrides the default namespace declared in the **Register** class.

Additionally, `Cpu0RegisterInfo.td` defines **CPURegs** as an instance of the **RegisterClass**, a built-in **LLVM class**.

A **RegisterClass** is essentially a **set of Register instances**, so **CPURegs** can be described as a set of registers.

Cpu0 Instruction Definition

The **Cpu0 instruction** description file is named **Cpu0InstrInfo.td**. Its contents are as follows:

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//==== Cpu0InstrInfo.td - Target Description for Cpu0 Target -*- tablegen -*- //  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
//  
// This file contains the Cpu0 implementation of the TargetInstrInfo class.  
//  
//===== //  
//===== //  
// Cpu0 profiles and nodes  
//===== //
```

(continues on next page)

(continued from previous page)

```
def SDT_Cpu0Ret      : SDTypeProfile<0, 1, [SDTCisInt<0>]>;  
  
// Return  
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,  
    [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;  
  
//===== //  
// Instruction format superclass  
//===== //  
  
include "Cpu0InstrFormats.td"  
  
//===== //  
// Cpu0 Operand, Complex Patterns and Transformations Definitions.  
//===== //  
// Instruction operand types  
  
// Signed Operand  
def simm16      : Operand<i32> {  
    let DecoderMethod= "DecodeSimm16";  
}  
  
// Address operand  
def mem : Operand<iPTR> {  
    let PrintMethod = "printMemOperand";  
    let MIOperandInfo = (ops GPROut, simm16);  
    let EncoderMethod = "getMemEncoding";  
}  
  
// Node immediate fits as 16-bit sign extended on target immediate.  
// e.g. addi, andi  
def immSExt16 : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;  
  
// Cpu0 Address Mode! SDNode frameindex could possibly be a match  
// since load and store instructions from stack used it.  
def addr :  
    ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;  
  
//===== //  
// Pattern fragment for load/store  
//===== //  
  
class AlignedLoad<PatFrag Node> :  
    PatFrag<(ops node:$ptr), (Node node:$ptr), [{  
        LoadSDNode *LD = cast<LoadSDNode>(N);  
        return LD->getMemoryVT().getSizeInBits()/8 <= LD->getAlignment();  
    }]>;  
  
class AlignedStore<PatFrag Node> :  
    PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr), [{  
        StoreSDNode *SD = cast<StoreSDNode>(N);  
        return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();  
}]>;
```

(continues on next page)

(continued from previous page)

```

}]>;

// Load/Store PatFrgs.
def load_a          : AlignedLoad<load>;
def store_a         : AlignedStore<store>;

//=====
// Instructions specific format
//=====

// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
                  Operand Od, PatLeaf imm_type, RegisterClass RC> :
    FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
    !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
    [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
let isReMaterializable = 1;
}

class FMem<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
           InstrItinClass itin>: FL<op, outs, ins, asmstr, pattern, itin> {
bits<20> addr;
let Inst{19-16} = addr{19-16};
let Inst{15-0} = addr{15-0};
let DecoderMethod = "DecodeMem";
}

// Memory Load/Store
let canFoldAsLoad = 1 in
class LoadM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
            Operand MemOpnd, bit Pseudo>:
    FMem<op, (outs RC:$ra), (ins MemOpnd:$addr),
    !strconcat(instr_asm, "\t$ra, $addr"),
    [(set RC:$ra, (OpNode addr:$addr))], IILoad> {
let isPseudo = Pseudo;
}

class StoreM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
            Operand MemOpnd, bit Pseudo>:
    FMem<op, (outs), (ins RC:$ra, MemOpnd:$addr),
    !strconcat(instr_asm, "\t$ra, $addr"),
    [(OpNode RC:$ra, addr:$addr)], IIStore> {
let isPseudo = Pseudo;
}

//@ 32-bit load.
class LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
              bit Pseudo = 0>
: LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {

// 32-bit store.

```

(continues on next page)

(continued from previous page)

```
class StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
    bit Pseudo = 0>
: StoreM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}

// @JumpFR {
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
    FL<op, (outs), (ins RC:$ra),
        !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
    let rb = 0;
    let imm16 = 0;
}
// @JumpFR }

// Return instruction
class RetBase<RegisterClass RC>: JumpFR<0x3c, "ret", RC> {
    let isReturn = 1;
    let isCodeGenOnly = 1;
    let hasCtrlDep = 1;
    let hasExtraSrcRegAllocReq = 1;
}

//=====
// Instruction definition
=====//

//=====
// Cpu0 Instructions
=====//

/// Load and Store Instructions
/// aligned
def LD      : LoadM32<0x01, "ld", load_a>;
def ST      : StoreM32<0x02, "st", store_a>;

/// Arithmetic Instructions (ALU Immediate)
// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

/// Arithmetic Instructions (3-Operand, R-Type)

/// Shift Instructions

def JR      : JumpFR<0x3c, "jr", GPROut>;
def RET     : RetBase<GPROut>;

/// No operation
let addr=0 in
```

(continues on next page)

(continued from previous page)

```

def NOP      : FJ<0, (outs), (ins), "nop", [], IIAlu>;

//=====
// Arbitrary patterns that map to one or more instructions
//=====

// Small immediates
def : Pat<(i32 immSExt16:$in),
      (ADDiu ZERO, imm:$in)>;

```

The *Cpu0InstrFormats.td* file is included in **Cpu0InstrInfo.td**, as shown:

Ibindex/chapters/Chapter2/Cpu0InstrFormats.td

```

//===== Cpu0InstrFormats.td - Cpu0 Instruction Formats -----*- tablegen -*---// 
// 
//          The LLVM Compiler Infrastructure
// 
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
// 
//=====

//=====

// Describe CPU0 instructions format
// 
// CPU INSTRUCTION FORMATS
// 
// opcode - operation code.
// ra    - dst reg, only used on 3 regs instr.
// rb    - src reg.
// rc    - src reg (on a 3 reg instr).
// cx    - immediate
// 
//=====

// Format specifies the encoding used by the instruction. This is part of the
// ad-hoc solution used to emit machine instruction encodings by our machine
// code emitter.
class Format<bits<4> val> {
    bits<4> Value = val;
}

def Pseudo      : Format<0>;
def FrmA       : Format<1>;
def FrmL       : Format<2>;
def FrmJ       : Format<3>;
def FrmOther   : Format<4>; // Instruction w/ a custom format

// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,

```

(continues on next page)

(continued from previous page)

```

        InstrItinClass itin, Format f>: Instruction
{
    // Inst and Size: for tablegen(... -gen-emitter) and
    // tablegen(... -gen-disassembler) in CMakeLists.txt
    field bits<32> Inst;
    Format Form = f;

    let Namespace = "Cpu0";

    let Size = 4;

    bits<8> Opcode = 0;

    // Top 8 bits are the 'opcode' field
    let Inst{31-24} = Opcode;

    let OutOperandList = outs;
    let InOperandList = ins;

    let AsmString = asmstr;
    let Pattern = pattern;
    let Itinerary = itin;

    //
    // Attributes specific to Cpu0 instructions...
    //
    bits<4> FormBits = Form.Value;

    // TSFlags layout should be kept in sync with Cpu0InstrInfo.h.
    let TSFlags{3-0} = FormBits;

    let DecoderNamespace = "Cpu0";

    field bits<32> SoftFail = 0;
}

//=====
// Format A instruction class in Cpu0 : <| opcode|ra|rb|rc|cx|>
//=====

class FA<bits<8> op, dag outs, dag ins, string asmstr,
    list<dag> pattern, InstrItinClass itin>:
    Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmA>
{
    bits<4> ra;
    bits<4> rb;
    bits<4> rc;
    bits<12> shamt;

    let Opcode = op;

    let Inst{23-20} = ra;
}

```

(continues on next page)

(continued from previous page)

```

let Inst{19-16} = rb;
let Inst{15-12} = rc;
let Inst{11-0} = shamt;
}

//@class FL {
//=====
// Format L instruction class in Cpu0 : <|opcode|ra|rb|cx|>
//=====

class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
           InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>
{
    bits<4> ra;
    bits<4> rb;
    bits<16> imm16;

    let Opcode = op;

    let Inst{23-20} = ra;
    let Inst{19-16} = rb;
    let Inst{15-0} = imm16;
}
//@class FL }

//=====
// Format J instruction class in Cpu0 : <|opcode|address|>
//=====

class FJ<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
           InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmJ>
{
    bits<24> addr;

    let Opcode = op;

    let Inst{23-0} = addr;
}

```

Expanding ADDiu

ADDiu is an instance of the **ArithLogicI** class, which inherits from *FL*. It can be further expanded to retrieve its member values as follows:

```

def ADDiu : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPUREgs>;

/// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
                  Operand Od, PatLeaf imm_type, RegisterClass RC> :
    FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
    !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
    [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {

```

(continues on next page)

(continued from previous page)

```
let isReMaterializable = 1;
}
```

So,

```
op = 0x09
instr_asm = "addiu"
OpNode = add
Od = simm16
imm_type = immSExt16
RC = CPURegs
```

Expanding .td Files: Key Principles

- **let**: Overrides an existing field from the parent class.
 - Example: `let isReMaterializable = 1;` This overrides `isReMaterializable` from the `Instruction` class in `Target.td`.
- **Declaration**: Defines a new field for the class.
 - Example: `bits<4> ra;` This declares the `ra` field in the `FL` class.

ADDiu Expansion Details

The details of expansion are shown in the following table:

Table 2.13: ADDiu Expansion Part I

ADDiu	ArithLogicl	FL
0x09	op = 0x09	Opcode = 0x09;
addiu	instr_asm = "addiu"	(outs GPROut:\$ra); !strconcat("addiu", "t\$ra, \$rb, \$imm16");
add	OpNode = add	[(set GPROut:\$ra, (add CPURegs:\$rb, immSExt16:\$imm16))]
simm16	Od = simm16	(ins CPURegs:\$rb, simm16:\$imm16);
imm-SExt16	imm_type = immSExt16	Inst{15-0} = imm16;
CPURegs	RC = CPURegs isReMaterializable=1;	Inst{23-20} = ra; Inst{19-16} = rb;

Table 2.14: ADDiu Expansion part II

Cpu0Inst	instruction
Namespace = "Cpu0"	Uses = [];
Inst{31-24} = 0x09;	Size = 0;
OutOperandList = GPROut:\$ra;	
InOperandList = CPURegs:\$rb,simm16:\$imm16;	
AsmString = "addiu\$tra, \$rb, \$imm16"	
pattern = [(set GPROut:\$ra, (add RC:\$rb, immSExt16:\$imm16))]	
Itinerary = IIAlu	
TSFlags{3-0} = FrmL.value	
DecoderNamespace = "Cpu0"	

The .td file expansion process can be cumbersome. Similarly, **LD** and **ST** instruction definitions can be expanded in the

same manner.

Note the **Pattern**:

[(set GPROut:\$ra, (add RC:\$rb, immSExt16:\$imm16))]

which includes the keyword “**add**”.

The **ADDiu** instruction with “**add**” was used in the *Instruction Selection* subsection of the previous section.

Cpu0 Scheduling Information

The *Cpu0Schedule.td* file includes details about **function units** and **pipeline stages**, as shown below:

Ibdex/chapters/Chapter2/Cpu0Schedule.td

```
//===== Cpu0Schedule.td - Cpu0 Scheduling Definitions -----*- tablegen -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====-----//



//=====-----// 
// Functional units across Cpu0 chips sets. Based on GCC/Cpu0 backend files.
//=====-----//


def ALU      : FuncUnit;
def IMULDIV : FuncUnit;

//=====-----// 
// Instruction Itinerary classes used for Cpu0
//=====-----//


def IIAlu      : InstrItinClass;
def II_CLO     : InstrItinClass;
def II_CLZ     : InstrItinClass;
def IILoad     : InstrItinClass;
def IIStore    : InstrItinClass;
def IIBranch   : InstrItinClass;

def IIPseudo   : InstrItinClass;

//=====-----// 
// Cpu0 Generic instruction itineraries.
//=====-----//


//@ http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
//@2
    InstrItinData<IIAlu      , [InstrStage<1,  [ALU]>]>,
    InstrItinData<II_CLO     , [InstrStage<1,  [ALU]>]>,
    InstrItinData<II_CLZ     , [InstrStage<1,  [ALU]>]>,
    InstrItinData<IILoad     , [InstrStage<3,  [ALU]>]>,
    InstrItinData<IIStore    , [InstrStage<1,  [ALU]>]>,
    InstrItinData<IIBranch   , [InstrStage<1,  [ALU]>]>
```

(continues on next page)

(continued from previous page)

```
] >;
```

Writing the CMake File

The *Target/Cpu0* directory contains the *CMakeLists.txt* file. Its contents are as follows:

Ibdex/chapters/Chapter2/CMakeLists.txt

```
add_llvm_component_group(Cpu0)

set(LLVM_TARGET_DEFINITIONS Cpu0Other.td)

# Generate Cpu0GenRegisterInfo.inc and Cpu0GenInstrInfo.inc which included by
# your hand code C++ files.
# Cpu0GenRegisterInfo.inc came from Cpu0RegisterInfo.td, Cpu0GenInstrInfo.inc
# came from Cpu0InstrInfo.td.
tablegen(LLVM Cpu0GenRegisterInfo.inc -gen-register-info)
tablegen(LLVM Cpu0GenInstrInfo.inc -gen-instr-info)
tablegen(LLVM Cpu0GenSubtargetInfo.inc -gen-subtarget)
tablegen(LLVM Cpu0GenMCPseudoLowering.inc -gen-pseudo-lowering)

# Cpu0CommonTableGen must be defined
add_public_tablegen_target(Cpu0CommonTableGen)

# Cpu0CodeGen should match with LLVMBuild.txt Cpu0CodeGen
add_llvm_target(Cpu0CodeGen
    Cpu0TargetMachine.cpp

LINK_COMPONENTS
Analysis
AsmPrinter
CodeGen
Core
MC
Cpu0Desc
Cpu0Info
SelectionDAG
Support
Target
GlobalISel

ADD_TO_COMPONENT
Cpu0
)

# Should match with "subdirectories = MCTargetDesc TargetInfo" in LLVMBuild.txt
add_subdirectory(TargetInfo)
add_subdirectory(MCTargetDesc)
```

CMakeLists.txt provides **build instructions** for **CMake**. Comments in this file are prefixed with #.

The “tablegen(” function in *CMakeLists.txt* is defined in:

cmake/modules/TableGen.cmake

as shown below:

llvm/cmake/modules/TableGen.cmake

```
function(tablegen project ofn)
    ...
    add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/${ofn}.tmp
        # Generate tablegen output in a temporary file.
        COMMAND ${${project}_TABLEGEN_EXE} ${ARGN} -I ${CMAKE_CURRENT_SOURCE_DIR}
    ...
endfunction()

macro(add_tablegen target project)
    ...
    if(LLVM_USE_HOST_TOOLS)
        if( ${${project}_TABLEGEN} STREQUAL "${target}" )
            if(NOT CMAKE_CONFIGURATION_TYPES)
                set(${${project}_TABLEGEN_EXE} "${LLVM_NATIVE_BUILD}/bin/${target}")
            else()
                set(${${project}_TABLEGEN_EXE} "${LLVM_NATIVE_BUILD}/Release/bin/${target}")
            endif()
        ...
    endmacro()
```

llvm/utils/TableGen/CMakeLists.txt

```
add_tablegen(llvm-tblgen LLVM
    ...
)
```

The “*add_tablegen*” function in *llvm/utils/TableGen/CMakeLists.txt* makes “*tablegen*” in *Cpu0 CMakeLists.txt* an alias for **llvm-tblgen** (where *project* = LLVM and *project*_TABLEGEN_EXE = *llvm-tblgen*).

The following elements define the **Cpu0CommonTableGen** target, which generates the output files **Cpu0Gen*.inc**:

- “*tablegen*”
- “*add_public_tablegen_target(Cpu0CommonTableGen)*”
- The following additional code

llvm/cmake/modules/TableGen.cmake

```
function(tablegen project ofn)
    ...
    set(TABLEGEN_OUTPUT ${TABLEGEN_OUTPUT} ${CMAKE_CURRENT_BINARY_DIR}/${ofn} PARENT_
    ↲SCOPE)
    ...
endfunction()

# Creates a target for publicly exporting tablegen dependencies.
function(add_public_tablegen_target target)
    ...
```

(continues on next page)

(continued from previous page)

```
add_custom_target(${target}
    DEPENDS ${TABLEGEN_OUTPUT})
...
endfunction()
```

Since the **llvm-tblgen** executable is built before compiling any LLVM backend source code, it is always available to handle TableGen requests during the build process.

This book introduces backend source code **incrementally**, adding code **chapter by chapter** based on function.

Understanding Source Code

- **Don't try to understand everything from the text alone**—the code added in each chapter serves as **learning material** too.
- **Conceptual understanding** of computer-related knowledge can be gained without reading source code.
- However, when implementing based on existing open-source software, reading source code is essential.
- **Documentation cannot fully replace source code** in programming.
- **Reading source code is a valuable skill in open-source development.**

CMakeLists.txt in Subdirectories

The *CMakeLists.txt* files exist in the **MCTargetDesc** and **TargetInfo** subdirectories.

These files instruct LLVM to generate the **Cpu0Desc** and **Cpu0Info** libraries, respectively.

After building, you will find three libraries in the *lib*/directory of your build folder:

- **libLLVMCpu0CodeGen.a**
- **libLLVMCpu0Desc.a**
- **libLLVMCpu0Info.a**

For more details, refer to “*Building LLVM with CMake*”²⁸.

2.4.3 Target Registration

You must **register your target** with the **TargetRegistry**. After registration, LLVM tools can **identify and use** your target at runtime.

Although the **TargetRegistry** can be used directly, most targets utilize **helper templates** to simplify the process.

All targets should declare a **global Target object**, which represents the target during registration.

Then, in the target’s **TargetInfo library**, the target should define this object and use the **RegisterTarget** template to register it.

For example, the file *TargetInfo/Cpu0TargetInfo.cpp* registers **TheCpu0Target** for **big-endian** and **TheCpu0elTarget** for **little-endian**, as shown below:

Ibdex/chapters/Chapter2/Cpu0.h

```
===== Cpu0.h - Top-level interface for Cpu0 representation -----*-- C++ -*---=//  
//  
// The LLVM Compiler Infrastructure  
//
```

(continues on next page)

²⁸ <http://llvm.org/docs/CMake.html>

(continued from previous page)

```
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file contains the entry points for global functions defined in
// the LLVM Cpu0 back-end.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0_H
#define LLVM_LIB_TARGET_CPU0_CPU0_H

#include "Cpu0Config.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
    class Cpu0TargetMachine;
    class FunctionPass;

} // end namespace llvm;

#endif
```

Ibdex/chapters/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

```
===== Cpu0TargetInfo.cpp - Cpu0 Target Implementation =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

#include "Cpu0.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;

extern "C" void LLVMInitializeCpu0TargetInfo() {
    RegisterTarget<Triple::cpu0,
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "CPU0 (32-bit big endian)", "Cpu0");

    RegisterTarget<Triple::cpu0el,
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "CPU0 (32-bit little endian)",
        "Cpu0");
}
```

Ibdex/chapters/Chapter2/TargetInfo/CMakeLists.txt

```
# llvm 10.0.0 change from add_llvm_library to add_llvm_component_library
add_llvm_component_library(LLVMCpu0Info
    Cpu0TargetInfo.cpp

    LINK_COMPONENTS
        Support

    ADD_TO_COMPONENT
        Cpu0
    )
```

The files **Cpu0TargetMachine.cpp** and **MCTargetDesc/Cpu0MCTargetDesc.cpp** currently define only an **empty initialization function**, as no components are being registered at this stage.

Ibdex/chapters/Chapter2/Cpu0TargetMachine.cpp

```
//===== Cpu0TargetMachine.cpp - Define TargetMachine for Cpu0 -----//  
//  
//           The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----//  
//  
// Implements the info about Cpu0 target spec.  
//  
//=====-----//  
  
#include "Cpu0TargetMachine.h"  
#include "Cpu0.h"  
  
#include "llvm/IR/Attributes.h"  
#include "llvm/IR/Function.h"  
#include "llvm/Support/CodeGen.h"  
#include "llvm/CodeGen/Passes.h"  
#include "llvm/CodeGen/TargetPassConfig.h"  
#include "llvm/Support/TargetRegistry.h"  
#include "llvm/Target/TargetOptions.h"  
  
using namespace llvm;  
  
#define DEBUG_TYPE "cpu0"  
  
extern "C" void LLVMInitializeCpu0Target() {  
}
```

Ibdex/chapters/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.h

```
===== Cpu0MCTargetDesc.h - Cpu0 Target Descriptions -----*- C++ -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file provides Cpu0 specific target descriptions.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCTARGETDESC_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCTARGETDESC_H

#include "Cpu0Config.h"
#include "llvm/Support/DataTypes.h"

#include <memory>

namespace llvm {
class Target;
class Triple;

extern Target TheCpu0Target;
extern Target TheCpu0elTarget;

} // End llvm namespace

// Defines symbolic names for Cpu0 registers. This defines a mapping from
// register name to register number.
#define GET_REGINFO_ENUM
#include "Cpu0GenRegisterInfo.inc"

// Defines symbolic names for the Cpu0 instructions.
#define GET_INSTRINFO_ENUM
#include "Cpu0GenInstrInfo.inc"

#define GET_SUBTARGETINFO_ENUM
#include "Cpu0GenSubtargetInfo.inc"

#endif
```

Ibdex/chapters/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
===== Cpu0MCTargetDesc.cpp - Cpu0 Target Descriptions =====/
//
// The LLVM Compiler Infrastructure
//
```

(continues on next page)

(continued from previous page)

```
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file provides Cpu0 specific target descriptions.
//
//=====

#include "Cpu0MCTargetDesc.h"
#include "llvm/MC/MachineLocation.h"
#include "llvm/MC/MCELFStreamer.h"
#include "llvm/MC/MCInstrAnalysis.h"
#include "llvm/MC/MCInstPrinter.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/FormattedStream.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define GET_INSTRINFO_MC_DESC
#include "Cpu0GenInstrInfo.inc"

#define GET_SUBTARGETINFO_MC_DESC
#include "Cpu0GenSubtargetInfo.inc"

#define GET_REGINFO_MC_DESC
#include "Cpu0GenRegisterInfo.inc"

//@2 {
extern "C" void LLVMInitializeCpu0TargetMC() {

}
//@2 }
```

Ibdex/chapters/Chapter2/MCTargetDesc/CMakeLists.txt

```
# MCTargetDesc/CMakeLists.txt
add_llvm_component_library(LLVMCpu0Desc
    Cpu0MCTargetDesc.cpp

    LINK_COMPONENTS
    MC
    Cpu0Info
    Support
```

(continues on next page)

(continued from previous page)

```
ADD_TO_COMPONENT
Cpu0
)
```

For reference, see “*Target Registration*”²⁹.

2.4.4 Build Libraries and .td Files

Build steps: <https://github.com/Jonathan2251/lbd/blob/master/README.md>. Illustrated in [Appendix A](#).

We set the LLVM source code in:

```
/Users/Jonathan/llvm/debug/llvm
```

and perform a debug build in:

```
/Users/Jonathan/llvm/debug/build
```

Note

Remember to build both clang and llvm(llc) in debug mode to generate the destination registers in LLVM IR and to print DAGs with identifiers such as t0, t1, ... (e.g. “t25: i32 = add t22, t28”, “t26: i32 = mul t25, Constant:i32<12>”).

For details on how to build LLVM, refer to³⁰.

In [Appendix A](#), we create a copy of the LLVM source directory:

```
/Users/Jonathan/llvm/debug/llvm
```

to:

```
/Users/Jonathan/llvm/test/llvm
```

for developing the **Cpu0 target backend**.

- The *llvm/* directory contains the **source code**.
- The *build/* directory is used for the **debug build**.

Modifying LLVM for Cpu0

Beyond *llvm/lib/Target/Cpu0*, several files have been modified to support the **new Cpu0 target**. These modifications include:

- **Adding the target's ID and name**
- **Defining relocation records** (as discussed in an earlier section)

To update your LLVM working copy and apply the modifications, use:

```
cp -rf lbdex/llvm/modify/llvm/* <yourllvm/workingcopy/sourcedir>/
```

²⁹ <http://llvm.org/docs/WritingAnLLVMBBackend.html#target-registration>

³⁰ http://clang.llvm.org/get_started.html

```
118-165-78-230:lbd Jonathan$ pwd  
/Users/Jonathan/git/lbd  
118-165-78-230:lbd Jonathan$ cp -rf lbdex/llvm/modify/llvm/* ~/llvm/test/llvm/.  
118-165-78-230:lbd Jonathan$ grep -R "cpu0" ~/llvm/test/llvm/include  
llvm/cmake/config-ix.cmake:elseif (LLVM_NATIVE_ARCH MATCHES "cpu0")  
llvm/include/llvm/ADT/Triple.h:#undef cpu0  
llvm/include/llvm/ADT/Triple.h:    cpu0,           // For Tutorial Backend Cpu0  
llvm/include/llvm/ADT/Triple.h:    cpu0el,  
llvm/include/llvm/Support/ELF.h:   EF_CPU0_ARCH_32R2 = 0x70000000, // cpu032r2  
llvm/include/llvm/Support/ELF.h:   EF_CPU0_ARCH_64R2 = 0x80000000, // cpu064r2  
...  
...
```

Next, configure the Cpu0 example code for **Chapter 2** as follows:

```
~/llvm/test/llvm/lib/Target/Cpu0/Cpu0SetChapter.h
```

```
#define CH          CH2
```

In addition to configuring the chapter as shown above, I provide **gen-chapters.sh**, which allows you to retrieve the code for each chapter as follows:

```
118-165-78-230:lbdex Jonathan$ pwd  
/Users/Jonathan/git/lbd/lbdex  
118-165-78-230:lbdex Jonathan$ bash gen-chapters.sh  
118-165-78-230:lbdex Jonathan$ ls chapters  
Chapter10_1    Chapter11_2    Chapter2        Chapter3_2...  
Chapter11_1    Chapter12_1    Chapter3_1        Chapter3_3...
```

Now, run the `cmake` and `make` commands to build `.td` files. (The following `cmake` command is based on my setup.)

```
118-165-78-230:build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++  
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" .. llvm/  
  
-- Targeting Cpu0  
...  
-- Targeting XCore  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /Users/Jonathan/llvm/test/build  
  
118-165-78-230:build Jonathan$ make -j4  
  
118-165-78-230:build Jonathan$
```

After building, you can run the command `llc --version` to verify that the Cpu0 backend is available.

```
118-165-78-230:build Jonathan$ /Users/Jonathan/llvm/test/  
build/bin/llc --version  
LLVM (http://llvm.org/):  
...  
Registered Targets:  
arm      - ARM  
...  
...
```

(continues on next page)

(continued from previous page)

```

cpp      - C++ backend
cpu0    - Cpu0
cpu0el  - Cpu0el
...

```

The command `llc --version` will display the registered targets “`cpu0`” and “`cpu0el`”, as defined in *Target-Info/Cpu0TargetInfo.cpp* from the previous section, *Target Registration*³¹.

Now, let’s build the *lbdex/chapters/Chapter2* code as follows:

```

118-165-75-57:test Jonathan$ pwd
/Users/Jonathan/test
118-165-75-57:test Jonathan$ cp -rf lbdex/Cpu0 ~/llvm/test/llvm/lib/Target/.

118-165-75-57:test Jonathan$ cd ~/llvm/test/build
118-165-75-57:build Jonathan$ pwd
/Users/Jonathan/llvm/test/build
118-165-75-57:build Jonathan$ rm -rf *
118-165-75-57:build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=Cpu0
-G "Unix Makefiles" ../llvm/
...
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/Jonathan/llvm/test/build

```

To save time, we build only the Cpu0 target using the option:

```
-DLLVM_TARGETS_TO_BUILD=Cpu0
```

After the build, you can find the generated `*.inc` files in:

```
/Users/Jonathan/llvm/test/build/lib/Target/Cpu0
```

as shown below:

build/lib/Target/Cpu0/Cpu0GenRegisterInfo.inc

```

namespace Cpu0 {
enum {
    NoRegister,
    AT = 1,
    EPC = 2,
    FP = 3,
    GP = 4,
    HI = 5,
    LO = 6,
    LR = 7,
    PC = 8,
    SP = 9,
    SW = 10,
}

```

(continues on next page)

³¹ <http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration>

(continued from previous page)

```
ZERO = 11,
A0 = 12,
A1 = 13,
S0 = 14,
S1 = 15,
T0 = 16,
T1 = 17,
T9 = 18,
V0 = 19,
V1 = 20,
NUM_TARGET_REGS // 21
};

}

...
```

These *.inc files are generated by `llvm-tblgen` in the `build/lib/Target/Cpu0` directory, using the Cpu0 backend *.td files as input.

The `llvm-tblgen` tool is invoked by **tablegen** in `/Users/Jonathan/llvm/test/llvm/lib/Target/Cpu0/CMakeLists.txt`.

These *.inc files are later included in Cpu0 backend .cpp or .h files and compiled into .o files.

TableGen is a crucial tool, as discussed earlier in the “*.td: LLVM’s Target Description Files*” section of this chapter. For reference, the TableGen documentation is available here:^{Page 59, 323334}.

Now, try running the `llc` command to compile the input file `ch3.cpp`:

lbdex/input/ch3.cpp

```
int main()
{
    return 0;
}
```

First step, compile it with clang and get output ch3.bc as follows,

```
118-165-78-230:input Jonathan$ pwd
/Users/Jonathan/git/lbd/lbdex/input
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
```

As shown above, compile C to .bc using:

```
clang -target mips-unknown-linux-gnu
```

since Cpu0 borrows its ABI from MIPS.

Next, convert the bitcode (.bc) to a human-readable text format as follows:

```
118-165-78-230:test Jonathan$ llvm-dis ch3.bc -o -
// ch3.ll
```

(continues on next page)

³³ <http://llvm.org/docs/TableGen/LangIntro.html>

³⁴ <http://llvm.org/docs/TableGen/LangRef.html>

(continued from previous page)

```

; ModuleID = 'ch3.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f3
2:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:f80:128:128-n8:16:32:6
4-S128"
target triple = "mips-unknown-linux-gnu"

define i32 @main() nounwind uwtable {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    ret i32 0
}

```

Now, compiling *ch3.bc* will result in the following error message:

```

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
...
... Assertion `target.get() && "Could not allocate target machine!"' failed
...

```

At this point, we have completed *Target Registration* for the Cpu0 backend. The *llc* compiler command now recognizes the Cpu0 backend.

Currently, we have only defined the target *.td* files (*Cpu0.td*, *Cpu0Other.td*, *Cpu0RegisterInfo.td*, etc.). According to the LLVM structure, we need to define our target machine and include these *.td* files.

The error message indicates that the target machine is not yet defined. This book follows a step-by-step approach to backend development. You can review the **hundreds** of lines in the Chapter 2 example code to understand how *Target Registration* is implemented.

2.5 Debug options

2.5.1 Options for *llc* Debugging

Run the following command to see hidden *llc* options:

```
llc --help-hidden
```

The following *llc* options require an input *.bc* or *.ll* file:

- *-debug*
- *-debug-pass=Structure*
- *-print-after-all*, *-print-before-all*
- *-print-before="pass"* and *-print-after="pass"*

Example: *-print-before="postra-machine-sink" -print-after="postra-machine-sink"*

The pass name can be obtained as follows:

```

CodeGen % pwd
~/llvm/debug/llvm/lib/CodeGen
CodeGen % grep -R "INITIALIZE_PASS" |grep sink
./MachineSink.cpp:INITIALIZE_PASS(PostRAMachineSinking, "postra-machine-sink",

```

- `-view-dag-combine1-dags` Displays the DAG after being built, before the first optimization pass.
- `-view-legalize-dags` Displays the DAG before legalization.
- `-view-dag-combine2-dags` Displays the DAG before the second optimization pass.
- `-view-isel-dags` Displays the DAG before the Select phase.
- `-view-sched-dags` Displays the DAG before scheduling.
- `-march=<string>` Specifies the target architecture (e.g., `-march=mips`).
- `-relocation-model=static/pic` Sets the relocation model.
- `-filetype=asm/obj` Specifies the output file type (assembly or object).

You can use `F.dump()` in the code, where `F` is an instance of the `Function` class, to inspect transformations in `llvm/lib/Transformation`.

2.5.2 *opt* Debugging Options

Check available options using:

```
opt --help-hidden
```

Refer to LLVM passes documentation³⁵. Examples:

- `opt -dot-cfg input.ll` Prints the CFG of a function to a `.dot` file.
- `-dot-cfg-only` Prints the CFG to a `.dot` file without function bodies.

³⁵ <https://llvm.org/docs/Passes.html>

BACKEND STRUCTURE

- *TargetMachine structure*
- *Add AsmPrinter*
- *Add Cpu0DAGToDAGISel class*
- *Handle return register \$lr*
- *Add Prologue/Epilogue functions*
 - *Concept*
 - *Prologue and Epilogue functions*
 - *Handle stack slot for local variables*
 - *Large stack*
- *Data operands DAGs*

From Fig. 2.9, llvm compiler transfer from llvm-ir to assembly or binary object with the following data structure as Fig. 3.1.

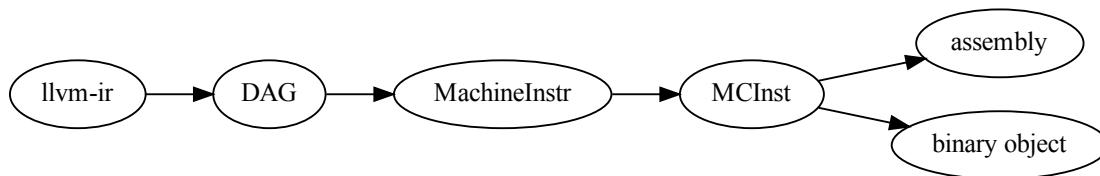


Fig. 3.1: LLVM data structure used in different stages

Cpu0 backend supports the following backend compiler, assembler and disassembler as Fig. 3.2, Fig. 3.3 and Fig. 3.4. However only the green part for printing assembly is implemented in this chapter. Others are implemented in later chapters Fig. 5.2, Fig. 11.5 and Fig. 10.2.

- Bytes: 4-byte (32-bits) for Cpu0.
- Each MInst contains one single instruction.

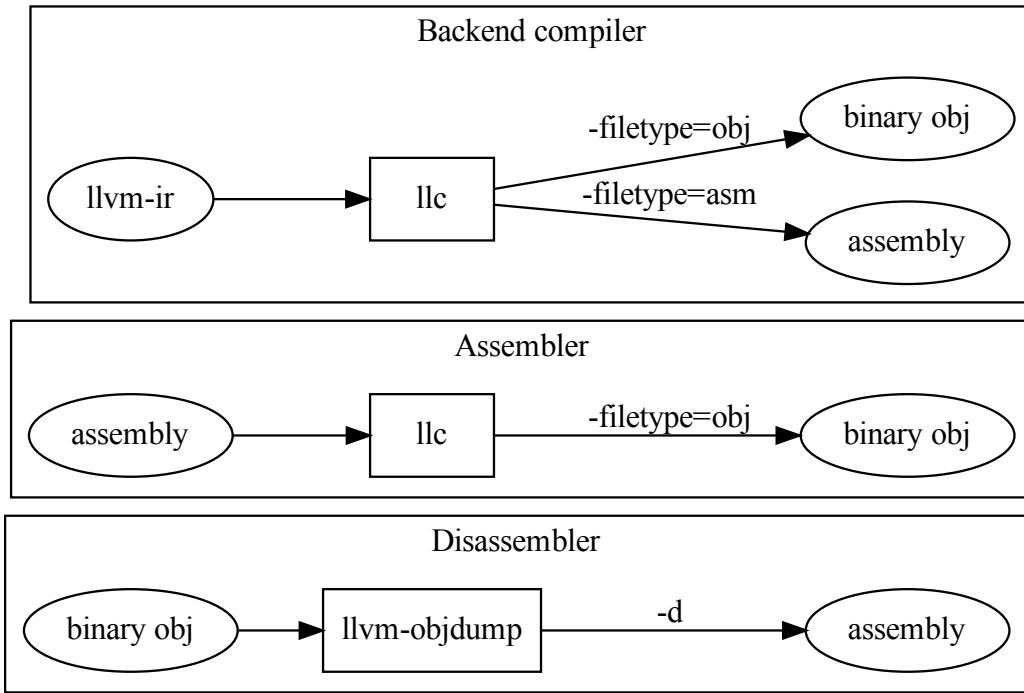


Fig. 3.2: Backend compiler, assembler and disassembler of Cpu0

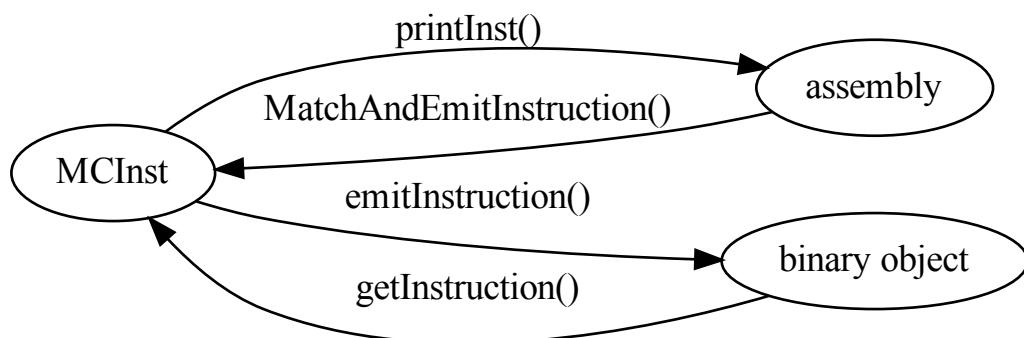


Fig. 3.3: Encode, assembler and disassembler

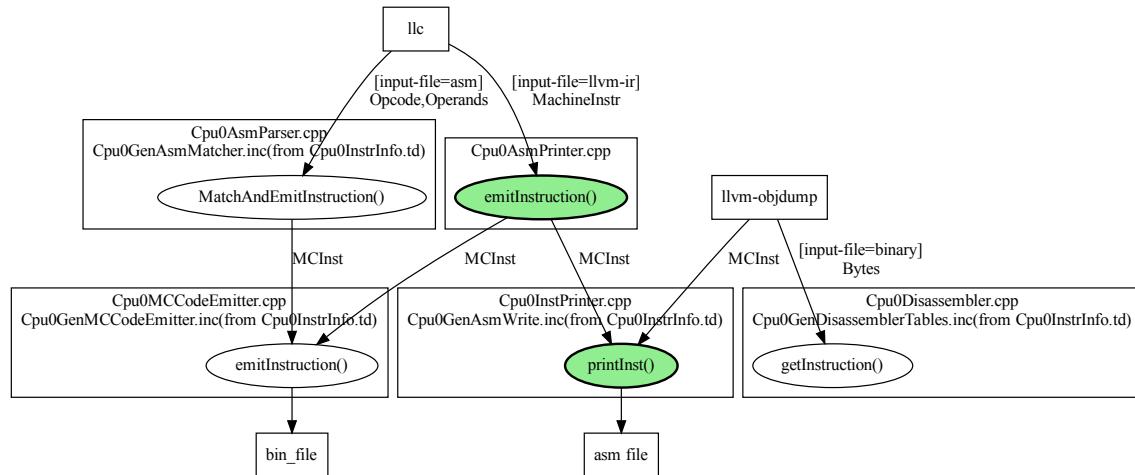


Fig. 3.4: The structure for backend compiler, assembler and disassembler of Cpu0

- The encoder extracts information from MachineInstr and places it into MCInst. It then calls printInst() to generate assembly instructions or emitInstruction() to output binary instruction.
- The assembler calls MatchAndEmitInstruction() to convert assembly back into MCInst, then calls emitInstruction() to output binary instruction.
- The disassembler calls getInstruction() to convert binary instructions back into MCInst, then calls printInst() to generate assembly instruction.
- The emitInstruction() of Cpu0MCCCodeEmitter.cpp: encode binary for an instruction reused for both llc (compiler) and llc (assembler).
- The printInst() of Cpu0InstPrinter.cpp: print assembly code for an instruction reused for both llc (compiler) and llvm-objdump (disassembler).

This chapter first introduces the backend class inheritance tree and its class members. Then, following the backend structure, we add individual class implementations in each section. By the end of this chapter, we will have a backend capable of compiling LLVM intermediate code into Cpu0 assembly code.

Many lines of code are introduced in this chapter. Most of them are common across different backends, except for the backend name (e.g., Cpu0 or Mips). In fact, we copy almost all the code from Mips and replace the name with Cpu0. Beyond understanding DAG pattern matching in theoretical compilers and the LLVM code generation phase, please focus on the relationships between classes in this backend structure. Once you grasp the structure, you will be able to create your backend as quickly as we did, even though this chapter introduces around 5000 lines of code.

3.1 TargetMachine structure

[Index/chapters/Chapter3_1/Cpu0TargetObjectFile.h](#)

```
//===== llvm/Target/Cpu0TargetObjectFile.h - Cpu0 Object Info ---*- C++ -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
```

(continues on next page)

(continued from previous page)

```
// License. See LICENSE.TXT for details.  
//  
//===== //  
  
#ifndef LLVM_LIB_TARGET_CPU0_CPU0TARGETOBJECTFILE_H  
#define LLVM_LIB_TARGET_CPU0_CPU0TARGETOBJECTFILE_H  
  
#include "Cpu0Config.h"  
  
#include "Cpu0TargetMachine.h"  
#include "llvm/CodeGen/TargetLoweringObjectFileImpl.h"  
  
namespace llvm {  
    class Cpu0TargetMachine;  
    class Cpu0TargetObjectFile : public TargetLoweringObjectFileELF {  
        MCSection *SmallDataSection;  
        MCSection *SmallBSSSection;  
        const Cpu0TargetMachine *TM;  
  
        public:  
            void Initialize(MCContext &Ctx, const TargetMachine &TM) override;  
        };  
} // end namespace llvm  
  
#endif
```

Index/chapters/Chapter3_1/Cpu0TargetObjectFile.cpp

```
//===== Cpu0TargetObjectFile.cpp - Cpu0 Object Files ===== //  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
  
#include "Cpu0TargetObjectFile.h"  
  
#include "Cpu0Subtarget.h"  
#include "Cpu0TargetMachine.h"  
#include "llvm/BinaryFormat/ELF.h"  
#include "llvm/IR/DataLayout.h"  
#include "llvm/IR/DerivedTypes.h"  
#include "llvm/IR/GlobalVariable.h"  
#include "llvm/MC/MCContext.h"  
#include "llvm/MC/MCSectionELF.h"  
#include "llvm/Support/CommandLine.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/Target/TargetMachine.h"
using namespace llvm;

static cl::opt<unsigned>
SSThreshold("cpu0-ssection-threshold", cl::Hidden,
            cl::desc("Small data and bss section threshold size (default=8)"),
            cl::init(8));

void Cpu0TargetObjectFile::Initialize(MCContext &Ctx, const TargetMachine &TM) {
    TargetLoweringObjectFileELF::Initialize(Ctx, TM);
    InitializeELF(TM.Options.UseInitArray);

    SmallDataSection = getContext().getELFSection(
        ".sdata", ELF::SHT_PROGBITS, ELF::SHF_WRITE | ELF::SHF_ALLOC);

    SmallBSSSection = getContext().getELFSection(".sbss", ELF::SHT_NOBITS,
                                                ELF::SHF_WRITE | ELF::SHF_ALLOC);
    this->TM = &static_cast<const Cpu0TargetMachine &>(TM);
}
```

Ibdex/chapters/Chapter3_1/Cpu0TargetMachine.h

```
//===== Cpu0TargetMachine.h - Define TargetMachine for Cpu0 -----*- C++ -*=====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
//
// This file declares the Cpu0 specific subclass of TargetMachine.
//
//=====//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0TARGETMACHINE_H
#define LLVM_LIB_TARGET_CPU0_CPU0TARGETMACHINE_H

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0ABIInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/TargetFrameLowering.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
class formatted_raw_ostream;
```

(continues on next page)

(continued from previous page)

```
class Cpu0RegisterInfo;

class Cpu0TargetMachine : public LLVMTargetMachine {
    bool isLittle;
    std::unique_ptr<TargetLoweringObjectFile> TLOF;
    // Selected ABI
    Cpu0ABIInfo ABI;
    Cpu0Subtarget DefaultSubtarget;

    mutable StringMap<std::unique_ptr<Cpu0Subtarget>> SubtargetMap;
public:
    Cpu0TargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                      StringRef FS, const TargetOptions &Options,
                      Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                      CodeGenOpt::Level OL, bool JIT, bool isLittle);
    ~Cpu0TargetMachine() override;

    const Cpu0Subtarget *getSubtargetImpl() const {
        return &DefaultSubtarget;
    }

    const Cpu0Subtarget *getSubtargetImpl(const Function &F) const override;

    // Pass Pipeline Configuration
    TargetPassConfig *createPassConfig(PassManagerBase &PM) override;

    TargetLoweringObjectFile *getObjFileLowering() const override {
        return TLOF.get();
    }
    bool isLittleEndian() const { return isLittle; }
    const Cpu0ABIInfo &getABI() const { return ABI; }
};

/// Cpu0ebTargetMachine - Cpu032 big endian target machine.
///
class Cpu0ebTargetMachine : public Cpu0TargetMachine {
    virtual void anchor();
public:
    Cpu0ebTargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                        StringRef FS, const TargetOptions &Options,
                        Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                        CodeGenOpt::Level OL, bool JIT);
};

/// Cpu0elTargetMachine - Cpu032 little endian target machine.
///
class Cpu0elTargetMachine : public Cpu0TargetMachine {
    virtual void anchor();
public:
    Cpu0elTargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                        StringRef FS, const TargetOptions &Options,
                        Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
```

(continues on next page)

(continued from previous page)

```

        CodeGenOpt::Level OL, bool JIT);
};

} // End llvm namespace

#endif

```

Ibdex/chapters/Chapter3_1/Cpu0TargetMachine.cpp

```

//===== Cpu0TargetMachine.cpp - Define TargetMachine for Cpu0 =====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// Implements the info about Cpu0 target spec.
//
//=====

#include "Cpu0TargetMachine.h"
#include "Cpu0.h"

#include "Cpu0Subtarget.h"
#include "Cpu0TargetObjectFile.h"
#include "llvm/IR/Attributes.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/TargetPassConfig.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0"

extern "C" void LLVMInitializeCpu0Target() {
    // Register the target.
    // Big endian Target Machine
    RegisterTargetMachine<Cpu0ebTargetMachine> X(TheCpu0Target);
    // Little endian Target Machine
    RegisterTargetMachine<Cpu0elTargetMachine> Y(TheCpu0elTarget);
}

static std::string computeDataLayout(const Triple &TT, StringRef CPU,
                                    const TargetOptions &Options,
                                    bool isLittle) {
    std::string Ret = "";

```

(continues on next page)

(continued from previous page)

```
// There are both little and big endian cpu0.
if (isLittle)
    Ret += "e";
else
    Ret += "E";

Ret += "-m:m";

// Pointers are 32 bit on some ABIs.
Ret += "-p:32:32";

// 8 and 16 bit integers only need to have natural alignment, but try to
// align them to 32 bits. 64 bit integers have natural alignment.
Ret += "-i8:8:32-i16:16:32-i64:64";

// 32 bit registers are always available and the stack is at least 64 bit
// aligned.
Ret += "-n32-S64";

return Ret;
}

static Reloc::Model getEffectiveRelocModel(bool JIT,
                                         Optional<Reloc::Model> RM) {
    if (!RM.HasValue() || JIT)
        return Reloc::Static;
    return *RM;
}

// DataLayout --> Big-endian, 32-bit pointer/ABI/alignment
// The stack is always 8 byte aligned
// On function prologue, the stack is created by decrementing
// its pointer. Once decremented, all references are done with positive
// offset from the stack/frame pointer, using StackGrowsUp enables
// an easier handling.
// Using CodeModel::Large enables different CALL behavior.
Cpu0TargetMachine::Cpu0TargetMachine(const Target &T, const Triple &TT,
                                      StringRef CPU, StringRef FS,
                                      const TargetOptions &Options,
                                      Optional<Reloc::Model> RM,
                                      Optional<CodeModel::Model> CM,
                                      CodeGenOpt::Level OL, bool JIT,
                                      bool isLittle)
    // Default is big endian
    : LLVMTargetMachine(T, computeDataLayout(TT, CPU, Options, isLittle), TT,
                        CPU, FS, Options, getEffectiveRelocModel(JIT, RM),
                        getEffectiveCodeModel(CM, CodeModel::Small), OL),
      isLittle(isLittle), TLOF(std::make_unique<Cpu0TargetObjectFile>()),
      ABI(Cpu0ABIInfo::computeTargetABI()),
      DefaultSubtarget(TT, CPU, FS, isLittle, *this) {
    // initAsmInfo will display features by llc -march=cpu0 -mcpu=help on 3.7 but
    // not on 3.6
```

(continues on next page)

(continued from previous page)

```

initAsmInfo();
}

Cpu0TargetMachine::~Cpu0TargetMachine() {}

void Cpu0ebTargetMachine::anchor() {}

Cpu0ebTargetMachine::Cpu0ebTargetMachine(const Target &T, const Triple &TT,
                                        StringRef CPU, StringRef FS,
                                        const TargetOptions &Options,
                                        Optional<Reloc::Model> RM,
                                        Optional<CodeModel::Model> CM,
                                        CodeGenOpt::Level OL, bool JIT)
    : Cpu0TargetMachine(T, TT, CPU, FS, Options, RM, CM, OL, JIT, false) {}

void Cpu0elTargetMachine::anchor() {}

Cpu0elTargetMachine::Cpu0elTargetMachine(const Target &T, const Triple &TT,
                                        StringRef CPU, StringRef FS,
                                        const TargetOptions &Options,
                                        Optional<Reloc::Model> RM,
                                        Optional<CodeModel::Model> CM,
                                        CodeGenOpt::Level OL, bool JIT)
    : Cpu0TargetMachine(T, TT, CPU, FS, Options, RM, CM, OL, JIT, true) {}

const Cpu0Subtarget *
Cpu0TargetMachine::getSubtargetImpl(const Function &F) const {
    std::string CPU = TargetCPU;
    std::string FS = TargetFS;

    auto &I = SubtargetMap[CPU + FS];
    if (!I) {
        // This needs to be done before we create a new subtarget since any
        // creation will depend on the TM and the code generation flags on the
        // function that reside in TargetOptions.
        resetTargetOptions(F);
        I = std::make_unique<Cpu0Subtarget>(TargetTriple, CPU, FS, isLittle,
                                             *this);
    }
    return I.get();
}

namespace {
// @Cpu0PassConfig
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {
public:
    Cpu0PassConfig(Cpu0TargetMachine &TM, PassManagerBase &PM)
        : TargetPassConfig(TM, PM) {}

    Cpu0TargetMachine &getCpu0TargetMachine() const {
        return getTM<Cpu0TargetMachine>();
    }
}

```

(continues on next page)

(continued from previous page)

```

    }

    const Cpu0Subtarget &getCpu0Subtarget() const {
        return *getCpu0TargetMachine().getSubtargetImpl();
    }
};

} // namespace

TargetPassConfig *Cpu0TargetMachine::createPassConfig(PassManagerBase &PM) {
    return new Cpu0PassConfig(*this, PM);
}

```

include/llvm/Target/TargetInstrInfo.h

```

class TargetInstrInfo : public MCInstrInfo {
    TargetInstrInfo(const TargetInstrInfo &) = delete;
    void operator=(const TargetInstrInfo &) = delete;
public:
    ...
}

class TargetInstrInfoImpl : public TargetInstrInfo {
protected:
    TargetInstrInfoImpl(int CallFrameSetupOpcode = -1,
                        int CallFrameDestroyOpcode = -1)
        : TargetInstrInfo(CallFrameSetupOpcode, CallFrameDestroyOpcode) {}
public:
    ...
}

```

lbdex/chapters/Chapter3_1/Cpu0.td

```
include "Cpu0CallingConv.td"
```

lbdex/chapters/Chapter3_1/Cpu0CallingConv.td

```

===== Cpu0CallingConv.td - Calling Conventions for Cpu0 --*- tablegen -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This describes the calling conventions for Cpu0 architecture.
//=====

/// CCIfSubtarget - Match if the current subtarget has a feature F.
class CCIfSubtarget<string F, CCAction A>:

```

(continues on next page)

(continued from previous page)

```
CCIf<!strconcat ("State.getTarget ().getSubtarget<Cpu0Subtarget> () .", F), A>;  
  
def CSR_O32 : CalleeSavedRegs<(add LR, FP,  
                                (sequence "S%u", 1, 0))>;
```

Ibdex/chapters/Chapter3_1/Cpu0FrameLowering.h

```
//===== Cpu0FrameLowering.h - Define frame lowering for Cpu0 -----*- C++ -*-----//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----=====//  
//  
//  
//=====-----=====//  
#ifndef LLVM_LIB_TARGET_CPU0_CPU0FRAMELOWERING_H  
#define LLVM_LIB_TARGET_CPU0_CPU0FRAMELOWERING_H  
  
#include "Cpu0Config.h"  
  
#include "Cpu0.h"  
#include "llvm/CodeGen/TargetFrameLowering.h"  
  
namespace llvm {  
    class Cpu0Subtarget;  
  
    class Cpu0FrameLowering : public TargetFrameLowering {  
protected:  
    const Cpu0Subtarget &STI;  
  
public:  
    explicit Cpu0FrameLowering(const Cpu0Subtarget &sti, unsigned Alignment)  
        : TargetFrameLowering(StackGrowsDown, Align(Alignment), 0, Align(Alignment)),  
          STI(sti) {}  
  
    static const Cpu0FrameLowering *create(const Cpu0Subtarget &ST);  
  
    bool hasFP(const MachineFunction &MF) const override;  
};  
  
/// Create Cpu0FrameLowering objects.  
const Cpu0FrameLowering *createCpu0SEFrameLowering(const Cpu0Subtarget &ST);  
}  
// End llvm namespace
```

(continues on next page)

(continued from previous page)

```
#endif
```

Ibdex/chapters/Chapter3_1/Cpu0FrameLowering.cpp

```
===== Cpu0FrameLowering.cpp - Cpu0 Frame Information =====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file contains the Cpu0 implementation of TargetFrameLowering class.
//
//=====

#include "Cpu0FrameLowering.h"

#include "Cpu0InstrInfo.h"
#include "Cpu0MachineFunction.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineModuleInfo.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

// emitPrologue() and emitEpilogue must exist for main().

//=====
//
// Stack Frame Processing methods
// +-----+
//
// The stack is allocated decrementing the stack pointer on
// the first instruction of a function prologue. Once decremented,
// all stack references are done thought a positive offset
// from the stack/frame pointer, so the stack is considering
// to grow up! Otherwise terrible hacks would have to be made
// to get this stack ABI compliant :)
//
// The stack frame required by the ABI (after call):
```

(continues on next page)

(continued from previous page)

```
// Offset
//
// 0           -----
// 4           Args to pass
// .           saved $GP (used in PIC)
// .           Alloca allocations
// .           Local Area
// .           CPU "Callee Saved" Registers
// .           saved FP
// .           saved RA
// .           FPU "Callee Saved" Registers
// StackSize   -----
//
// Offset - offset from sp after stack allocation on function prologue
//
// The sp is the stack pointer subtracted/added from the stack size
// at the Prologue/Epilogue
//
// References to the previous stack (to obtain arguments) are done
// with offsets that exceeds the stack size: (stacksize+(4*(num_arg-1)))
//
// Examples:
// - reference to the actual stack frame
//   for any local area var there is smt like : FI >= 0, StackOffset: 4
//   st REGX, 4(SP)
//
// - reference to previous stack frame
//   suppose there's a load to the 5th arguments : FI < 0, StackOffset: 16.
//   The emitted instruction will be something like:
//   ld REGX, 16+StackSize(SP)
//
// Since the total stack size is unknown on LowerFormalArguments, all
// stack references (ObjectOffset) created to reference the function
// arguments, are negative numbers. This way, on eliminateFrameIndex it's
// possible to detect those references and the offsets are adjusted to
// their real location.
//
//=====
const Cpu0FrameLowering *Cpu0FrameLowering::create(const Cpu0Subtarget &ST) {
    return llvm::createCpu0SEFrameLowering(ST);
}

// hasFP - Return true if the specified function should have a dedicated frame
// pointer register. This is true if the function has variable sized allocas,
// if it needs dynamic stack realignment, if frame pointer elimination is
// disabled, or if the frame address is taken.
bool Cpu0FrameLowering::hasFP(const MachineFunction &MF) const {
    const MachineFrameInfo &MFI = MF.getFrameInfo();
    const TargetRegisterInfo *TRI = STI.getRegisterInfo();

    return MF.getTarget().Options.DisableFramePointerElim(MF) ||

```

(continues on next page)

(continued from previous page)

```
    MFI.hasVarSizedObjects() || MFI.isFrameAddressTaken() ||  
    TRI->needsStackRealignment(MF);  
}
```

Ibdex/chapters/Chapter3_1/Cpu0SEFrameLowering.h

```
===== Cpu0SEFrameLowering.h - Cpu032/64 frame lowering -----*- C++ -*===/  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
//  
//  
//===== //  
  
#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEFRAMELOWERING_H  
#define LLVM_LIB_TARGET_CPU0_CPU0SEFRAMELOWERING_H  
  
#include "Cpu0Config.h"  
  
#include "Cpu0FrameLowering.h"  
  
namespace llvm {  
  
class Cpu0SEFrameLowering : public Cpu0FrameLowering {  
public:  
    explicit Cpu0SEFrameLowering(const Cpu0Subtarget &STI);  
  
    /// emitProlog/emitEpilog - These methods insert prolog and epilog code into  
    /// the function.  
    void emitPrologue(MachineFunction &MF, MachineBasicBlock &MBB) const override;  
    void emitEpilogue(MachineFunction &MF, MachineBasicBlock &MBB) const override;  
};  
  
} // End llvm namespace  
  
#endif
```

Ibdex/chapters/Chapter3_1/Cpu0SEFrameLowering.cpp

```
===== Cpu0SEFrameLowering.cpp - Cpu0 Frame Information -----//  
//  
// The LLVM Compiler Infrastructure  
//
```

(continues on next page)

(continued from previous page)

```
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file contains the Cpu0 implementation of TargetFrameLowering class.
//
//=====

#include "Cpu0SEFrameLowering.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0SEInstrInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineModuleInfo.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/RegisterScavenging.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

Cpu0SEFrameLowering::Cpu0SEFrameLowering(const Cpu0Subtarget &STI)
    : Cpu0FrameLowering(STI, STI.stackAlignment()) {}

//@emitPrologue {
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
}

//}

//@emitEpilogue {
void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
}

//}

const Cpu0FrameLowering *
llvm::createCpu0SEFrameLowering(const Cpu0Subtarget &ST) {
    return new Cpu0SEFrameLowering(ST);
}
```

Ibdex/chapters/Chapter3_1/Cpu0InstrInfo.h

```
//===== Cpu0InstrInfo.h - Cpu0 Instruction Information -----*- C++ -*=====//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//=====
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0INSTRINFO_H
#define LLVM_LIB_TARGET_CPU0_CPU0INSTRINFO_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/TargetInstrInfo.h"

#define GET_INSTRINFO_HEADER
#include "Cpu0GenInstrInfo.inc"

namespace llvm {

class Cpu0InstrInfo : public Cpu0GenInstrInfo {
    virtual void anchor();
protected:
    const Cpu0Subtarget &Subtarget;
public:
    explicit Cpu0InstrInfo(const Cpu0Subtarget &STI);

    static const Cpu0InstrInfo *create(Cpu0Subtarget &STI);

    /// getRegisterInfo - TargetInstrInfo is a superset of MRegister info. As
    /// such, whenever a client has an instance of instruction info, it should
    /// always be able to get register info as well (through this method).
    ///
    virtual const Cpu0RegisterInfo &getRegisterInfo() const = 0;

    /// Return the number of bytes of code the specified instruction may be.
    unsigned GetInstSizeInBytes(const MachineInstr &MI) const;

protected:
};

const Cpu0InstrInfo *createCpu0SEInstrInfo(const Cpu0Subtarget &STI);
}
```

(continues on next page)

(continued from previous page)

```
#endif
```

Ibdex/chapters/Chapter3_1/Cpu0InstrInfo.cpp

```
//===== Cpu0InstrInfo.cpp - Cpu0 Instruction Information =====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//=====

#include "Cpu0InstrInfo.h"

#include "Cpu0TargetMachine.h"
#include "Cpu0MachineFunction.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define GET_INSTRINFOCTOR_DTOR
#include "Cpu0GenInstrInfo.inc"

// Pin the vtable to this file.
void Cpu0InstrInfo::anchor() {}

//@Cpu0InstrInfo {
Cpu0InstrInfo::Cpu0InstrInfo(const Cpu0Subtarget &STI)
:
    Subtarget(STI) {}

const Cpu0InstrInfo *Cpu0InstrInfo::create(Cpu0Subtarget &STI) {
    return llvm::createCpu0SEInstrInfo(STI);
}

//@GetInstSizeInBytes {
/// Return the number of bytes of code the specified instruction may be.
unsigned Cpu0InstrInfo::GetInstSizeInBytes(const MachineInstr &MI) const {
//@GetInstSizeInBytes - body
    switch (MI.getOpcode()) {
    default:
        return MI.getDesc().getSize();
```

(continues on next page)

(continued from previous page)

```
}
```

Ibdex/chapters/Chapter3_1/Cpu0InstrInfo.td

```
//===== /  
// Cpu0 Instruction Predicate Definitions.  
//===== /  
  
def Ch3_1 : Predicate<"Subtarget->hasChapter3_1()",  
               AssemblerPredicate<(all_of FeatureChapter3_1)>;  
def Ch3_2 : Predicate<"Subtarget->hasChapter3_2()",  
               AssemblerPredicate<(all_of FeatureChapter3_2)>;  
def Ch3_3 : Predicate<"Subtarget->hasChapter3_3()",  
               AssemblerPredicate<(all_of FeatureChapter3_3)>;  
def Ch3_4 : Predicate<"Subtarget->hasChapter3_4()",  
               AssemblerPredicate<(all_of FeatureChapter3_4)>;  
def Ch3_5 : Predicate<"Subtarget->hasChapter3_5()",  
               AssemblerPredicate<(all_of FeatureChapter3_5)>;  
def Ch4_1 : Predicate<"Subtarget->hasChapter4_1()",  
               AssemblerPredicate<(all_of FeatureChapter4_1)>;  
def Ch4_2 : Predicate<"Subtarget->hasChapter4_2()",  
               AssemblerPredicate<(all_of FeatureChapter4_2)>;  
def Ch5_1 : Predicate<"Subtarget->hasChapter5_1()",  
               AssemblerPredicate<(all_of FeatureChapter5_1)>;  
def Ch6_1 : Predicate<"Subtarget->hasChapter6_1()",  
               AssemblerPredicate<(all_of FeatureChapter6_1)>;  
def Ch7_1 : Predicate<"Subtarget->hasChapter7_1()",  
               AssemblerPredicate<(all_of FeatureChapter7_1)>;  
def Ch8_1 : Predicate<"Subtarget->hasChapter8_1()",  
               AssemblerPredicate<(all_of FeatureChapter8_1)>;  
def Ch8_2 : Predicate<"Subtarget->hasChapter8_2()",  
               AssemblerPredicate<(all_of FeatureChapter8_2)>;  
def Ch9_1 : Predicate<"Subtarget->hasChapter9_1()",  
               AssemblerPredicate<(all_of FeatureChapter9_1)>;  
def Ch9_2 : Predicate<"Subtarget->hasChapter9_2()",  
               AssemblerPredicate<(all_of FeatureChapter9_2)>;  
def Ch9_3 : Predicate<"Subtarget->hasChapter9_3()",  
               AssemblerPredicate<(all_of FeatureChapter9_3)>;  
def Ch10_1 : Predicate<"Subtarget->hasChapter10_1()",  
               AssemblerPredicate<(all_of FeatureChapter10_1)>;  
def Ch11_1 : Predicate<"Subtarget->hasChapter11_1()",  
               AssemblerPredicate<(all_of FeatureChapter11_1)>;  
def Ch11_2 : Predicate<"Subtarget->hasChapter11_2()",  
               AssemblerPredicate<(all_of FeatureChapter11_2)>;  
def Ch12_1 : Predicate<"Subtarget->hasChapter12_1()",  
               AssemblerPredicate<(all_of FeatureChapter12_1)>;  
def Ch_all : Predicate<"Subtarget->hasChapterAll()",  
               AssemblerPredicate<(all_of FeatureChapterAll)>;
```

(continues on next page)

(continued from previous page)

```
def EnableOverflow : Predicate<"Subtarget->enableOverflow() ">;
def DisableOverflow : Predicate<"Subtarget->disableOverflow() ">;

def HasCmp      : Predicate<"Subtarget->hasCmp() ">;
def HasSlt      : Predicate<"Subtarget->hasSlt() ">;
```

Ibdex/chapters/Chapter3_1/Cpu0ISelLowering.h

```
===== Cpu0ISelLowering.h - Cpu0 DAG Lowering Interface -----*- C++ -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file defines the interfaces that Cpu0 uses to lower LLVM code into a
// selection DAG.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0ISELLOWERING_H
#define LLVM_LIB_TARGET_CPU0_CPU0ISELLOWERING_H

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0ABIInfo.h"
#include "Cpu0.h"
#include "llvm/CodeGen/CallingConvLower.h"
#include "llvm/CodeGen/SelectionDAG.h"
#include "llvm/IR/Function.h"
#include "llvm/CodeGen/TargetLowering.h"
#include <deque>

namespace llvm {
    namespace Cpu0ISD {
        enum NodeType {
            // Start the numbering from where ISD::NodeType finishes.
            FIRST_NUMBER = ISD::BUILTIN_OP_END,

            // Jump and link (call)
            JmpLink,

            // Tail call
            TailCall,

            // Get the Higher 16 bits from a 32-bit immediate
            // No relation with Cpu0 Hi register
            Hi,
            // Get the Lower 16 bits from a 32-bit immediate
        };
    }
}
```

(continues on next page)

(continued from previous page)

```
// No relation with Cpu0 Lo register
Lo,

// Handle gp_rel (small data/bss sections) relocation.
GPRel,

// Thread Pointer
ThreadPointer,

// Return
Ret,

EH_RETURN,

// DivRem(u)
DivRem,
DivRemU,

Wrapper,
DynAlloc,

Sync
};

}

//=====
// TargetLowering Implementation
//=====

class Cpu0FunctionInfo;
class Cpu0Subtarget;

//@class Cpu0TargetLowering
class Cpu0TargetLowering : public TargetLowering {
public:
    explicit Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                const Cpu0Subtarget &STI);

    static const Cpu0TargetLowering *create(const Cpu0TargetMachine &TM,
                                            const Cpu0Subtarget &STI);

    /// getTargetNodeName - This method returns the name of a target specific
    // DAG node.
    const char *getTargetNodeName(unsigned Opcode) const override;

protected:

    /// ByValArgInfo - ByVal argument information.
    struct ByValArgInfo {
        unsigned FirstIdx; // Index of the first register used.
        unsigned NumRegs; // Number of registers used for this argument.
        unsigned Address; // Offset of the stack area used to pass this argument.
    };
}
```

(continues on next page)

(continued from previous page)

```

    ByValArgInfo() : FirstIdx(0), NumRegs(0), Address(0) {}

};

protected:
    // Subtarget Info
    const Cpu0Subtarget &Subtarget;
    // Cache the ABI from the TargetMachine, we use it everywhere.
    const Cpu0ABIInfo &ABI;

private:
    // Lower Operand specifics
    SDValue lowerGlobalAddress(SDValue Op, SelectionDAG &DAG) const;

        // must be exist even without function all
    SDValue
    LowerFormalArguments(SDValue Chain,
                        CallingConv::ID CallConv, bool IsVarArg,
                        const SmallVectorImpl<ISD::InputArg> &Ins,
                        const SDLoc &dl, SelectionDAG &DAG,
                        SmallVectorImpl<SDValue> &InVals) const override;

    SDValue LowerReturn(SDValue Chain,
                        CallingConv::ID CallConv, bool IsVarArg,
                        const SmallVectorImpl<ISD::OutputArg> &Outs,
                        const SmallVectorImpl<SDValue> &OutVals,
                        const SDLoc &dl, SelectionDAG &DAG) const override;

};

const Cpu0TargetLowering *
createCpu0SETargetLowering(const Cpu0TargetMachine &TM, const Cpu0Subtarget &STI);
}

#endif // Cpu0ISELLOWERING_H

```

Ibdex/chapters/Chapter3_1/Cpu0ISEllowering.cpp

```

===== Cpu0ISEllowering.cpp - Cpu0 DAG Lowering Implementation ===== //
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== //
//
// This file defines the interfaces that Cpu0 uses to lower LLVM code into a
// selection DAG.
//
===== //

```

(continues on next page)

(continued from previous page)

```
#include "Cpu0ISelLowering.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0TargetMachine.h"
#include "Cpu0TargetObjectFile.h"
#include "Cpu0Subtarget.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/CallingConvLower.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAG.h"
#include "llvm/CodeGen/ValueTypes.h"
#include "llvm/IR/CallingConv.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-lower"

//@3_1 1 {
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
    switch (Opcode) {
        case Cpu0ISD::JmpLink:           return "Cpu0ISD::JmpLink";
        case Cpu0ISD::TailCall:          return "Cpu0ISD::TailCall";
        case Cpu0ISD::Hi:               return "Cpu0ISD::Hi";
        case Cpu0ISD::Lo:               return "Cpu0ISD::Lo";
        case Cpu0ISD::GPRel:            return "Cpu0ISD::GPRel";
        case Cpu0ISD::Ret:              return "Cpu0ISD::Ret";
        case Cpu0ISD::EH_RETURN:         return "Cpu0ISD::EH_RETURN";
        case Cpu0ISD::DivRem:           return "Cpu0ISD::DivRem";
        case Cpu0ISD::DivRemU:          return "Cpu0ISD::DivRemU";
        case Cpu0ISD::Wrapper:          return "Cpu0ISD::Wrapper";
        default:                         return NULL;
    }
}
//@3_1 }

//@Cpu0TargetLowering {
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                      const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {

}

const Cpu0TargetLowering *Cpu0TargetLowering::create(const Cpu0TargetMachine &TM,
```

(continues on next page)

(continued from previous page)

```

        const Cpu0Subtarget &STI) {
    return llvm::createCpu0SETargetLowering(TM, STI);
}

//=====
// Lower helper functions
//=====

//=====
// Misc Lower Operation implementation
//=====

#include "Cpu0GenCallingConv.inc"

//=====
//@          Formal Arguments Calling Convention Implementation
//=====

//@LowerFormalArguments {
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {

    return Chain;
}
// @LowerFormalArguments }

//=====
//@          Return Value Calling Convention Implementation
//=====

SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                               CallingConv::ID CallConv, bool IsVarArg,
                               const SmallVectorImpl<ISD::OutputArg> &Outs,
                               const SmallVectorImpl<SDValue> &OutVals,
                               const SDLoc &DL, SelectionDAG &DAG) const {
    return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other,
                        Chain, DAG.getRegister(Cpu0::LR, MVT::i32));
}

```

Ibdex/chapters/Chapter3_1/Cpu0SEISelLowering.h

```
===== Cpu0ISEISelLowering.h - Cpu0ISE DAG Lowering Interface -----* C++ -*=====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
//  
// Subclass of Cpu0ITargetLowering specialized for cpu032/64.  
//  
//===== //  
  
#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEISELLOWERING_H  
#define LLVM_LIB_TARGET_CPU0_CPU0SEISELLOWERING_H  
  
#include "Cpu0Config.h"  
  
#include "Cpu0ISelLowering.h"  
#include "Cpu0RegisterInfo.h"  
  
namespace llvm {  
    class Cpu0SETargetLowering : public Cpu0TargetLowering {  
    public:  
        explicit Cpu0SETargetLowering(const Cpu0TargetMachine &TM,  
                                      const Cpu0Subtarget &STI);  
  
        SDValue LowerOperation(SDValue Op, SelectionDAG &DAG) const override;  
    private:  
    };  
}  
  
#endif // Cpu0ISEISELLOWERING_H
```

Ibdex/chapters/Chapter3_1/Cpu0SEISelLowering.cpp

```
===== Cpu0SEISelLowering.cpp - Cpu0SE DAG Lowering Interface ---* C++ -*=====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
//  
// Subclass of Cpu0TargetLowering specialized for cpu032.  
//  
//===== //  
#include "Cpu0MachineFunction.h"  
#include "Cpu0SEISelLowering.h"
```

(continues on next page)

(continued from previous page)

```
#include "Cpu0RegisterInfo.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

static cl::opt<bool>
EnableCpu0TailCalls("enable-cpu0-tail-calls", cl::Hidden,
                     cl::desc("CPU0: Enable tail calls."), cl::init(false));

// @Cpu0SETargetLowering {
Cpu0SETargetLowering::Cpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                            const Cpu0Subtarget &STI)
    : Cpu0TargetLowering(TM, STI) {
// @Cpu0SETargetLowering body {
    // Set up the register classes
    addRegisterClass(MVT::i32, &Cpu0::CPURegsRegClass);

    // must, computeRegisterProperties - Once all of the register classes are
    // added, this allows us to compute derived properties we expose.
    computeRegisterProperties(Subtarget.getRegisterInfo());
}

SDValue Cpu0SETargetLowering::LowerOperation(SDValue Op,
                                             SelectionDAG &DAG) const {

    return Cpu0TargetLowering::LowerOperation(Op, DAG);
}

const Cpu0TargetLowering *
llvm::createCpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                 const Cpu0Subtarget &STI) {
    return new Cpu0SETargetLowering(TM, STI);
}
```

Ibdex/chapters/Chapter3_1/Cpu0MachineFunction.h

```
===== Cpu0MachineFunctionInfo.h - Private data used for Cpu0 -----*-- C++ -*-=//
//
//                                     The LLVM Compiler Infrastructure
//
```

(continues on next page)

(continued from previous page)

```
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file declares the Cpu0 specific subclass of MachineFunctionInfo.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0MACHINEFUNCTION_H
#define LLVM_LIB_TARGET_CPU0_CPU0MACHINEFUNCTION_H

#include "Cpu0Config.h"

#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineMemOperand.h"
#include "llvm/CodeGen/PseudoSourceValue.h"
#include "llvm/Target/TargetMachine.h"
#include <map>

namespace llvm {

//@1 {
/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
    Cpu0FunctionInfo(MachineFunction& MF)
        : MF(MF),
        VarArgsFrameIndex(0),
        MaxCallFrameSize(0)
    {}

    ~Cpu0FunctionInfo();

    int getVarArgsFrameIndex() const { return VarArgsFrameIndex; }
    void setVarArgsFrameIndex(int Index) { VarArgsFrameIndex = Index; }

private:
    virtual void anchor();

    MachineFunction& MF;

    /// VarArgsFrameIndex - FrameIndex for start of varargs area.
    int VarArgsFrameIndex;

    unsigned MaxCallFrameSize;
};

//@1 }

} // end of namespace llvm
```

(continues on next page)

(continued from previous page)

```
#endif // CPU0_MACHINE_FUNCTION_INFO_H
```

Ibdex/chapters/Chapter3_1/Cpu0MachineFunction.cpp

```
//===== Cpu0MachineFunctionInfo.cpp - Private data used for Cpu0 =====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====  
  
#include "Cpu0MachineFunction.h"  
  
#include "Cpu0InstrInfo.h"  
#include "Cpu0Subtarget.h"  
#include "llvm/IR/Function.h"  
#include "llvm/CodeGen/MachineInstrBuilder.h"  
#include "llvm/CodeGen/MachineRegisterInfo.h"  
  
using namespace llvm;  
  
bool FixGlobalBaseReg;  
  
Cpu0FunctionInfo::~Cpu0FunctionInfo() {}  
  
void Cpu0FunctionInfo::anchor() {}
```

Ibdex/chapters/Chapter3_1/MCTargetDesc/Cpu0ABIInfo.h

```
//===== Cpu0ABIInfo.h - Information about CPU0 ABI's =====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====  
  
#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ABIINFO_H  
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ABIINFO_H  
  
#include "Cpu0Config.h"  
  
#include "llvm/ADT/ArrayRef.h"  
#include "llvm/ADT/Triple.h"  
#include "llvm/IR/CallingConv.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/MC/MCRegisterInfo.h"

namespace llvm {

class MCTargetOptions;
class StringRef;
class TargetRegisterClass;

class Cpu0ABIInfo {
public:
    enum class ABI { Unknown, O32, S32 };

protected:
    ABI ThisABI;

public:
    Cpu0ABIInfo(ABI ThisABI) : ThisABI(ThisABI) {}

    static Cpu0ABIInfo Unknown() { return Cpu0ABIInfo(ABI::Unknown); }
    static Cpu0ABIInfo O32() { return Cpu0ABIInfo(ABI::O32); }
    static Cpu0ABIInfo S32() { return Cpu0ABIInfo(ABI::S32); }
    static Cpu0ABIInfo computeTargetABI();

    bool IsKnown() const { return ThisABI != ABI::Unknown; }
    bool IsO32() const { return ThisABI == ABI::O32; }
    bool IsS32() const { return ThisABI == ABI::S32; }
    ABI GetEnumValue() const { return ThisABI; }

    /// The registers to use for byval arguments.
    const ArrayRef<MCPhysReg> GetByValArgRegs() const;

    /// The registers to use for the variable argument list.
    const ArrayRef<MCPhysReg> GetVarArgRegs() const;

    /// Obtain the size of the area allocated by the callee for arguments.
    /// CallingConv::FastCall affects the value for O32.
    unsigned GetCalleeAllocdArgSizeInBytes(CallingConv::ID CC) const;

    /// Ordering of ABI's
    /// Cpu0GenSubtargetInfo.inc will use this to resolve conflicts when given
    /// multiple ABI options.
    bool operator<(const Cpu0ABIInfo Other) const {
        return ThisABI < Other.GetEnumValue();
    }

    unsigned GetStackPtr() const;
    unsigned GetFramePtr() const;
    unsigned GetNullPtr() const;

    unsigned GetEhDataReg(unsigned I) const;
    int EhDataRegSize() const;
};

}
```

(continues on next page)

(continued from previous page)

```
}
```

```
#endif
```

Ibdex/chapters/Chapter3_1/MCTargetDesc/Cpu0ABIInfo.cpp

```
===== Cpu0ABIInfo.cpp - Information about CPU0 ABI's =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== /
```

```
#include "Cpu0Config.h"

#include "Cpu0ABIInfo.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/MC/MCTargetOptions.h"
#include "llvm/Support/CommandLine.h"

using namespace llvm;

static cl::opt<bool>
EnableCpu0S32Calls("cpu0-s32-calls", cl::Hidden,
                    cl::desc("CPU0 S32 call: use stack only to pass arguments.\n"),
                    cl::init(false));

namespace {
static const MCPhysReg O32IntRegs[4] = {Cpu0::A0, Cpu0::A1};
static const MCPhysReg S32IntRegs = {};
}

const ArrayRef<MCPhysReg> Cpu0ABIInfo::GetByValArgRegs() const {
    if (IsO32())
        return makeArrayRef(O32IntRegs);
    if (IsS32())
        return makeArrayRef(S32IntRegs);
    llvm_unreachable("Unhandled ABI");
}

const ArrayRef<MCPhysReg> Cpu0ABIInfo::GetVarArgRegs() const {
    if (IsO32())
        return makeArrayRef(O32IntRegs);
    if (IsS32())
        return makeArrayRef(S32IntRegs);
    llvm_unreachable("Unhandled ABI");
}
```

(continues on next page)

(continued from previous page)

```
}

unsigned Cpu0ABIInfo::GetCalleeAllocdArgSizeInBytes(CallingConv::ID CC) const {
    if (IsO32())
        return CC != 0;
    if (IsS32())
        return 0;
    llvm_unreachable("Unhandled ABI");
}

Cpu0ABIInfo Cpu0ABIInfo::computeTargetABI() {
    Cpu0ABIInfo abi(ABI::Unknown);

    if (EnableCpu0S32Calls)
        abi = ABI::S32;
    else
        abi = ABI::O32;
    // Assert exactly one ABI was chosen.
    assert(abi.ThisABI != ABI::Unknown);

    return abi;
}

unsigned Cpu0ABIInfo::GetStackPtr() const {
    return Cpu0::SP;
}

unsigned Cpu0ABIInfo::GetFramePtr() const {
    return Cpu0::FP;
}

unsigned Cpu0ABIInfo::GetNullPtr() const {
    return Cpu0::ZERO;
}

unsigned Cpu0ABIInfo::GetEhDataReg(unsigned I) const {
    static const unsigned EhDataReg[] = {
        Cpu0::A0, Cpu0::A1
    };

    return EhDataReg[I];
}

int Cpu0ABIInfo::EhDataRegSize() const {
    if (ThisABI == ABI::S32)
        return 0;
    else
        return 2;
}
```

Ibdex/chapters/Chapter3_1/Cpu0Subtarget.h

```
#include "Cpu0FrameLowering.h"
#include "Cpu0ISelLowering.h"
#include "Cpu0InstrInfo.h"
#include "llvm/CodeGen/SelectionDAGTargetInfo.h"
#include "llvm/CodeGen/TargetSubtargetInfo.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/MC/MCInstrItineraries.h"
#include <string>

#define GET_SUBTARGETINFO_HEADER
#include "Cpu0GenSubtargetInfo.inc"
```

```
namespace llvm {
class StringRef;

class Cpu0TargetMachine;

class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
    virtual void anchor();

public:

    bool HasChapterDummy;
    bool HasChapterAll;

    bool hasChapter3_1() const {
#if CH >= CH3_1
        return true;
#else
        return false;
#endif
    }

    bool hasChapter3_2() const {
#if CH >= CH3_2
        return true;
#else
        return false;
#endif
    }

    bool hasChapter3_3() const {
#if CH >= CH3_3
        return true;
#else
        return false;
#endif
    }
}
```

(continues on next page)

(continued from previous page)

```
bool hasChapter3_4() const {
#ifndef CH3_4
    return true;
#else
    return false;
#endif
}

bool hasChapter3_5() const {
#ifndef CH3_5
    return true;
#else
    return false;
#endif
}

bool hasChapter4_1() const {
#ifndef CH4_1
    return true;
#else
    return false;
#endif
}

bool hasChapter4_2() const {
#ifndef CH4_2
    return true;
#else
    return false;
#endif
}

bool hasChapter5_1() const {
#ifndef CH5_1
    return true;
#else
    return false;
#endif
}

bool hasChapter6_1() const {
#ifndef CH6_1
    return true;
#else
    return false;
#endif
}

bool hasChapter7_1() const {
#ifndef CH7_1
    return true;
#else

```

(continues on next page)

(continued from previous page)

```

    return false;
#endif
}

bool hasChapter8_1() const {
#if CH >= CH8_1
    return true;
#else
    return false;
#endif
}

bool hasChapter8_2() const {
#if CH >= CH8_2
    return true;
#else
    return false;
#endif
}

bool hasChapter9_1() const {
#if CH >= CH9_1
    return true;
#else
    return false;
#endif
}

bool hasChapter9_2() const {
#if CH >= CH9_2
    return true;
#else
    return false;
#endif
}

bool hasChapter9_3() const {
#if CH >= CH9_3
    return true;
#else
    return false;
#endif
}

bool hasChapter10_1() const {
#if CH >= CH10_1
    return true;
#else
    return false;
#endif
}

```

(continues on next page)

(continued from previous page)

```
bool hasChapter11_1() const {
#ifndef CH >= CH11_1
    return true;
#else
    return false;
#endif
}

bool hasChapter11_2() const {
#ifndef CH >= CH11_2
    return true;
#else
    return false;
#endif
}

bool hasChapter12_1() const {
#ifndef CH >= CH12_1
    return true;
#else
    return false;
#endif
}

protected:
    enum Cpu0ArchEnum {
        Cpu032I,
        Cpu032II
    };

    // Cpu0 architecture version
    Cpu0ArchEnum Cpu0ArchVersion;

    // IsLittle - The target is Little Endian
    bool IsLittle;

    bool EnableOverflow;

    // HasCmp - cmp instructions.
    bool HasCmp;

    // HasSlt - slt instructions.
    bool HasSlt;

    InstrItineraryData InstrItins;
```

```
const Cpu0TargetMachine &TM;

Triple TargetTriple;
```

(continues on next page)

(continued from previous page)

```

const SelectionDAGTargetInfo TSInfo;

std::unique_ptr<const Cpu0InstrInfo> InstrInfo;
std::unique_ptr<const Cpu0FrameLowering> FrameLowering;
std::unique_ptr<const Cpu0TargetLowering> TLInfo;

public:
    bool isPositionIndependent() const;
    const Cpu0ABIInfo &getABI() const;

    /// This constructor initializes the data members to match that
    /// of the specified triple.
    Cpu0Subtarget(const Triple &TT, StringRef CPU, StringRef FS,
                  bool little, const Cpu0TargetMachine &_TM);

// Virtual function, must have
/// ParseSubtargetFeatures - Parses features string setting specified
/// subtarget options. Definition of function is auto generated by tblgen.
void ParseSubtargetFeatures(StringRef CPU, StringRef TuneCPU, StringRef FS);

bool isLittle() const { return IsLittle; }
bool hasCpu032I() const { return Cpu0ArchVersion >= Cpu032I; }
bool isCpu032I() const { return Cpu0ArchVersion == Cpu032I; }
bool hasCpu032II() const { return Cpu0ArchVersion >= Cpu032II; }
bool isCpu032II() const { return Cpu0ArchVersion == Cpu032II; }

    /// Features related to the presence of specific instructions.
    bool enableOverflow() const { return EnableOverflow; }
    bool disableOverflow() const { return !EnableOverflow; }
    bool hasCmp() const { return HasCmp; }
    bool hasSlt() const { return HasSlt; }

```

```

bool abiUsesSoftFloat() const;

bool enableLongBranchPass() const {
    return hasCpu032II();
}

unsigned stackAlignment() const { return 8; }

Cpu0Subtarget &initializeSubtargetDependencies(StringRef CPU, StringRef FS,
                                                const TargetMachine &TM);

const SelectionDAGTargetInfo *getSelectionDAGInfo() const override {
    return &TSInfo;
}
const Cpu0InstrInfo *getInstrInfo() const override { return InstrInfo.get(); }
const TargetFrameLowering *getFrameLowering() const override {
    return FrameLowering.get();
}

```

(continues on next page)

(continued from previous page)

```
const Cpu0RegisterInfo *getRegisterInfo() const override {
    return &InstrInfo->getRegisterInfo();
}
const Cpu0TargetLowering *getTargetLowering() const override {
    return TLInfo.get();
}
const InstrItineraryData *getInstrItineraryData() const override {
    return &InstrItins;
}
};

} // End llvm namespace

#endif
```

Index/chapters/Chapter3_1/Cpu0Subtarget.cpp

```
===== Cpu0Subtarget.cpp - Cpu0 Subtarget Information =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file implements the Cpu0 specific subclass of TargetSubtargetInfo.
//
//=====

#include "Cpu0Subtarget.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"

#include "Cpu0TargetMachine.h"
#include "llvm/IR/Attributes.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-subtarget"

#define GET_SUBTARGETINFO_TARGET_DESC
#define GET_SUBTARGETINFOCTOR
#include "Cpu0GenSubtargetInfo.inc"
```

(continues on next page)

(continued from previous page)

```

extern bool FixGlobalBaseReg;

void Cpu0Subtarget::anchor() { }

//@1 {
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                             StringRef FS, bool little,
                             const Cpu0TargetMachine &_TM) :
//@1 }
// Cpu0GenSubtargetInfo will display features by llc -march=cpu0 -mcpu=help
Cpu0GenSubtargetInfo(TT, CPU, /*TuneCPU*/ CPU, FS),
IsLittle(little), TM(_TM), TargetTriple(TT), TSInfo(),
InstrInfo(
    Cpu0InstrInfo::create(initializeSubtargetDependencies(CPU, FS, TM))),
FrameLowering(Cpu0FrameLowering::create(*this)),
TLInfo(Cpu0TargetLowering::create(TM, *this)) {

}

bool Cpu0Subtarget::isPositionIndependent() const {
    return TM.isPositionIndependent();
}

Cpu0Subtarget &
Cpu0Subtarget::initializeSubtargetDependencies(StringRef CPU, StringRef FS,
                                               const TargetMachine &TM) {
    if (TargetTriple.getArch() == Triple::cpu0 || TargetTriple.getArch() ==_
        Triple::cpu0el) {
        if (CPU.empty() || CPU == "generic") {
            CPU = "cpu032II";
        }
        else if (CPU == "help") {
            CPU = "";
            return *this;
        }
        else if (CPU != "cpu032I" && CPU != "cpu032II") {
            CPU = "cpu032II";
        }
    }
    else {
        errs() << "!!!Error, TargetTriple.getArch() = " << TargetTriple.getArch()
            << "CPU = " << CPU << "\n";
        exit(0);
    }

    if (CPU == "cpu032I")
        Cpu0ArchVersion = Cpu032I;
    else if (CPU == "cpu032II")
        Cpu0ArchVersion = Cpu032II;

    if (isCpu032I())
        HasCmp = true;
}

```

(continues on next page)

(continued from previous page)

```

        HasSlt = false;
    }
else if (isCpu032II()) {
    HasCmp = false;
    HasSlt = true;
}
else {
    errs() << "-mcpu must be empty(default:cpu032II), cpu032I or cpu032II" << "\n";
}

// Parse features string.
ParseSubtargetFeatures(CPU, /*TuneCPU*/ CPU, FS);
// Initialize scheduling itinerary for the specified CPU.
InstrItins = getInstrItineraryForCPU(CPU);

return *this;
}

bool Cpu0Subtarget::abiUsesSoftFloat() const {
// return TM->Options.UseSoftFloat;
return true;
}

const Cpu0ABIInfo &Cpu0Subtarget::getABI() const { return TM.getABI(); }

```

[Index/chapters/Chapter3_1/Cpu0RegisterInfo.h](#)

```

//===== Cpu0RegisterInfo.h - Cpu0 Register Information Impl -----*- C++ -*=====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file contains the Cpu0 implementation of the TargetRegisterInfo class.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0REGISTERINFO_H
#define LLVM_LIB_TARGET_CPU0_CPU0REGISTERINFO_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "llvm/CodeGen/TargetRegisterInfo.h"

#define GET_REGINFO_HEADER
#include "Cpu0GenRegisterInfo.inc"

```

(continues on next page)

(continued from previous page)

```

namespace llvm {
class Cpu0Subtarget;
class TargetInstrInfo;
class Type;

class Cpu0RegisterInfo : public Cpu0GenRegisterInfo {
protected:
    const Cpu0Subtarget &Subtarget;

public:
    Cpu0RegisterInfo(const Cpu0Subtarget &Subtarget);

    const MCPhysReg *getCalleeSavedRegs(const MachineFunction *MF) const override;

    const uint32_t *getCallPreservedMask(const MachineFunction &MF,
                                         CallingConv::ID) const override;

    BitVector getReservedRegs(const MachineFunction &MF) const override;

    bool requiresRegisterScavenging(const MachineFunction &MF) const override;

    bool trackLivenessAfterRegAlloc(const MachineFunction &MF) const override;

    /// Stack Frame Processing Methods
    void eliminateFrameIndex(MachineBasicBlock::iterator II,
                             int SPAdj, unsigned FIOperandNum,
                             RegScavenger *RS = nullptr) const override;

    /// Debug information queries.
    Register getFrameRegister(const MachineFunction &MF) const override;

    /// \brief Return GPR register class.
    virtual const TargetRegisterClass *intRegClass(unsigned Size) const = 0;
};

} // end namespace llvm

#endif

```

Ibdex/chapters/Chapter3_1/Cpu0RegisterInfo.cpp

```

//===== Cpu0RegisterInfo.cpp - CPU0 Register Information --- =====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

```

(continues on next page)

(continued from previous page)

```
//  
// This file contains the CPU0 implementation of the TargetRegisterInfo class.  
//  
//=====//  
  
#define DEBUG_TYPE "cpu0-reg-info"  
  
#include "Cpu0RegisterInfo.h"  
  
#include "Cpu0.h"  
#include "Cpu0Subtarget.h"  
#include "Cpu0MachineFunction.h"  
#include "llvm/IR/Function.h"  
#include "llvm/IR/Type.h"  
#include "llvm/Support/CommandLine.h"  
#include "llvm/Support/Debug.h"  
#include "llvm/Support/ErrorHandling.h"  
#include "llvm/Support/raw_ostream.h"  
  
#define GET_REGINFO_TARGET_DESC  
#include "Cpu0GenRegisterInfo.inc"  
  
using namespace llvm;  
  
Cpu0RegisterInfo::Cpu0RegisterInfo(const Cpu0Subtarget &ST)  
    : Cpu0GenRegisterInfo(Cpu0::LR), Subtarget(ST) {}  
  
//=====//  
// Callee Saved Registers methods  
//=====//  
/// Cpu0 Callee Saved Registers  
// In Cpu0CallConv.td,  
// def CSR_O32 : CalleeSavedRegs<(add LR, FP,  
//                                (sequence "S%u", 2, 0))>;  
// llc create CSR_O32_SaveList and CSR_O32_RegMask from above defined.  
const MCPhysReg *  
Cpu0RegisterInfo::getCalleeSavedRegs(const MachineFunction *MF) const {  
    return CSR_O32_SaveList;  
}  
  
const uint32_t *  
Cpu0RegisterInfo::getCallPreservedMask(const MachineFunction &MF,  
                                      CallingConv::ID) const {  
    return CSR_O32_RegMask;  
}  
  
// pure virtual method  
//@getReservedRegs {  
BitVector Cpu0RegisterInfo::  
getReservedRegs(const MachineFunction &MF) const {  
    //@getReservedRegs body {  
        static const uint16_t ReservedCPURegs[] = {
```

(continues on next page)

(continued from previous page)

```

        Cpu0::ZERO, Cpu0::AT, Cpu0::SP, Cpu0::LR, /*Cpu0::SW, */Cpu0::PC
    };
    BitVector Reserved(getNumRegs());

    for (unsigned I = 0; I < array_lengthof(ReservedCPURegs); ++I)
        Reserved.set(ReservedCPURegs[I]);

    return Reserved;
}

//@eliminateFrameIndex {
//- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    unsigned FIOperandNum, RegScavenger *RS) const {
}

bool
Cpu0RegisterInfo::requiresRegisterScavenging(const MachineFunction &MF) const {
    return true;
}

bool
Cpu0RegisterInfo::trackLivenessAfterRegAlloc(const MachineFunction &MF) const {
    return true;
}

// pure virtual method
Register Cpu0RegisterInfo::
getFrameRegister(const MachineFunction &MF) const {
    const TargetFrameLowering *TFI = MF.getSubtarget().getFrameLowering();
    return TFI->hasFP(MF) ? (Cpu0::FP) :
                               (Cpu0::SP);
}

```

[Index/chapters/Chapter3_1/Cpu0SERegisterInfo.h](#)

```

//===== Cpu0SERegisterInfo.h - Cpu032 Register Information -----*- C++ -*-==//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====-----//
```

(continues on next page)

(continued from previous page)

```
//  
// This file contains the Cpu032/64 implementation of the TargetRegisterInfo  
// class.  
//  
//=====//  
  
#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEREGISTERINFO_H  
#define LLVM_LIB_TARGET_CPU0_CPU0SEREGISTERINFO_H  
  
#include "Cpu0Config.h"  
  
#include "Cpu0RegisterInfo.h"  
  
namespace llvm {  
class Cpu0SEInstrInfo;  
  
class Cpu0SERegisterInfo : public Cpu0RegisterInfo {  
public:  
    Cpu0SERegisterInfo(const Cpu0Subtarget &Subtarget);  
  
    const TargetRegisterClass *intRegClass(unsigned Size) const override;  
};  
  
} // end namespace llvm  
  
#endif
```

Ibdex/chapters/Chapter3_1/Cpu0SERegisterInfo.cpp

```
//===== Cpu0SERegisterInfo.cpp - CPU0 Register Information ====== ======//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// This file contains the CPU0 implementation of the TargetRegisterInfo  
// class.  
//  
//=====//  
  
#include "Cpu0SERegisterInfo.h"  
  
using namespace llvm;  
  
#define DEBUG_TYPE "cpu0-reg-info"  
  
Cpu0SERegisterInfo::Cpu0SERegisterInfo(const Cpu0Subtarget &ST)
```

(continues on next page)

(continued from previous page)

```

: Cpu0RegisterInfo(ST) {}

const TargetRegisterClass *
Cpu0SERegisterInfo::intRegClass(unsigned Size) const {
    return &Cpu0::CPURegsRegClass;
}

```

build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

```

//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifndef GET_INSTRINFO_HEADER
#define GET_INSTRINFO_HEADER
namespace llvm {
    struct Cpu0GenInstrInfo : public TargetInstrInfoImpl {
        explicit Cpu0GenInstrInfo(int SO = -1, int DO = -1);
    };
} // End llvm namespace
#endif // GET_INSTRINFO_HEADER

#define GET_INSTRINFO_HEADER
#include "Cpu0GenInstrInfo.inc"
//- Cpu0InstInfo.h
class Cpu0InstrInfo : public Cpu0GenInstrInfo {
    Cpu0TargetMachine &TM;
public:
    explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);
};

```

Chapter3_1 adds most of the Cpu0 backend classes. The code in Chapter3_1 can be summarized in Fig. 3.5.

The **Cpu0Subtarget** class provides interfaces such as `getInstrInfo()`, `getFrameLowering()`, etc., to access other Cpu0 classes. Most classes (like **Cpu0InstrInfo**, **Cpu0RegisterInfo**, etc.) contain a **Subtarget** reference member, allowing them to access other classes through the **Cpu0Subtarget** interface.

If a backend module does not have a **Subtarget** reference, these classes can still access the **Subtarget** class through **Cpu0TargetMachine** (typically referred to as *TM*) using:

```
static_cast<Cpu0TargetMachine &>(TM).getSubtargetImpl()
```

Once the **Subtarget** class is obtained, the backend code can access other classes through it.

For classes named **Cpu0SExx**, they represent the standard 32-bit class. This naming convention follows the **LLVM 3.5 Mips backend** style. The Mips backend uses *Mips16*, *MipsSE*, and *Mips64* file/class names to define classes for 16-bit, 32-bit, and 64-bit architectures, respectively.

Since **Cpu0Subtarget** creates **Cpu0InstrInfo**, **Cpu0RegisterInfo**, etc., in its constructor, it can provide class references through the interfaces shown in Fig. 3.5.

Fig. 3.6 below illustrates the **Cpu0 TableGen** inheritance relationship.

In the previous chapter, we mentioned that backend classes can include TableGen-generated classes and inherit from them. All TableGen-generated classes for the Cpu0 backend are located in:

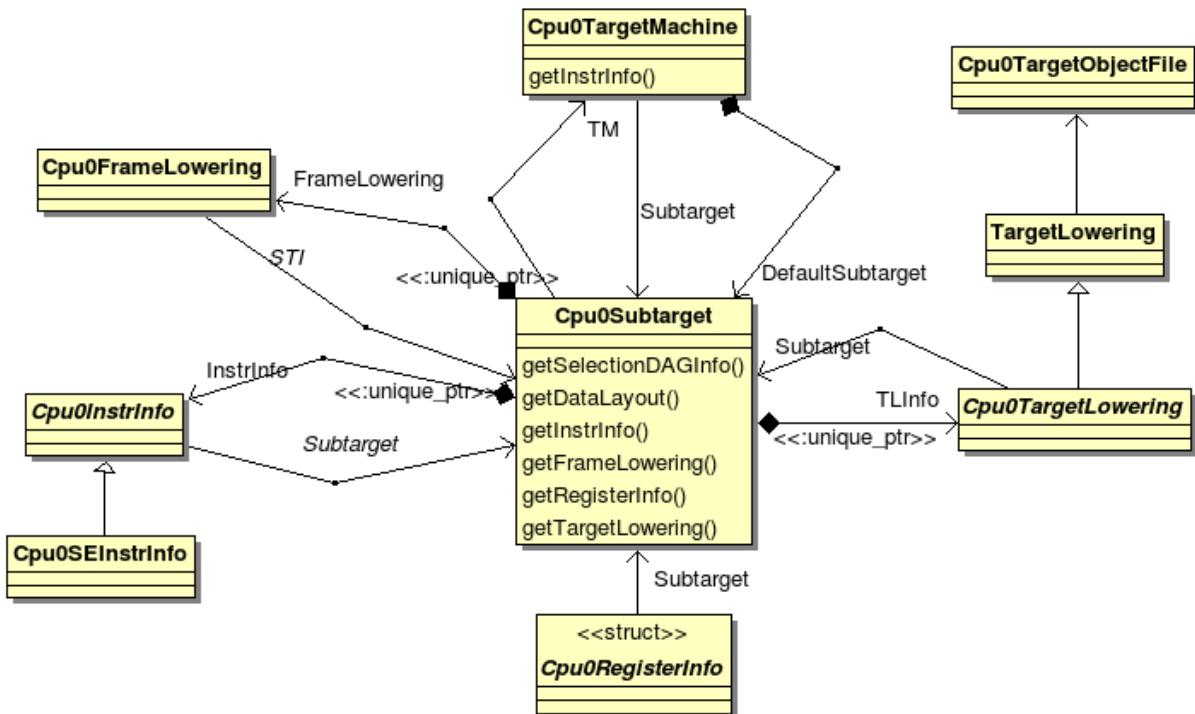


Fig. 3.5: Cpu0 backend class access link

build/lib/Target/Cpu0/*.inc

Through **C++ inheritance**, TableGen provides backend developers with a flexible way to utilize its generated code. Developers also have the opportunity to override functions if needed.

Since LLVM has a deep inheritance tree, it is not fully explored here. Benefiting from the inheritance tree structure, minimal code needs to be implemented in instruction, frame/stack, and DAG selection classes, as much of the functionality is already provided by their parent classes.

The `llvm-tblgen` tool generates **Cpu0GenInstrInfo.inc** based on the information in **Cpu0InstrInfo.td**.

Cpu0InstrInfo.h extracts the necessary code from **Cpu0GenInstrInfo.inc** by defining:

```
#define GET_INSTRINFO_HEADER
```

With TableGen, the backend code size is further reduced through the pattern matching theory of compiler development. This is explained in the “DAG” and “Instruction Selection” sections of the previous chapter.

The following is a code fragment from **Cpu0GenInstrInfo.inc**. Code between `#ifdef GET_INSTRINFO_HEADER` and `#endif // GET_INSTRINFO_HEADER` is extracted into **Cpu0InstrInfo.h**.

build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

```
//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifndef GET_INSTRINFO_HEADER
#define GET_INSTRINFO_HEADER
#endif // GET_INSTRINFO_HEADER
namespace llvm {
    struct Cpu0GenInstrInfo : public TargetInstrInfoImpl {
```

(continues on next page)

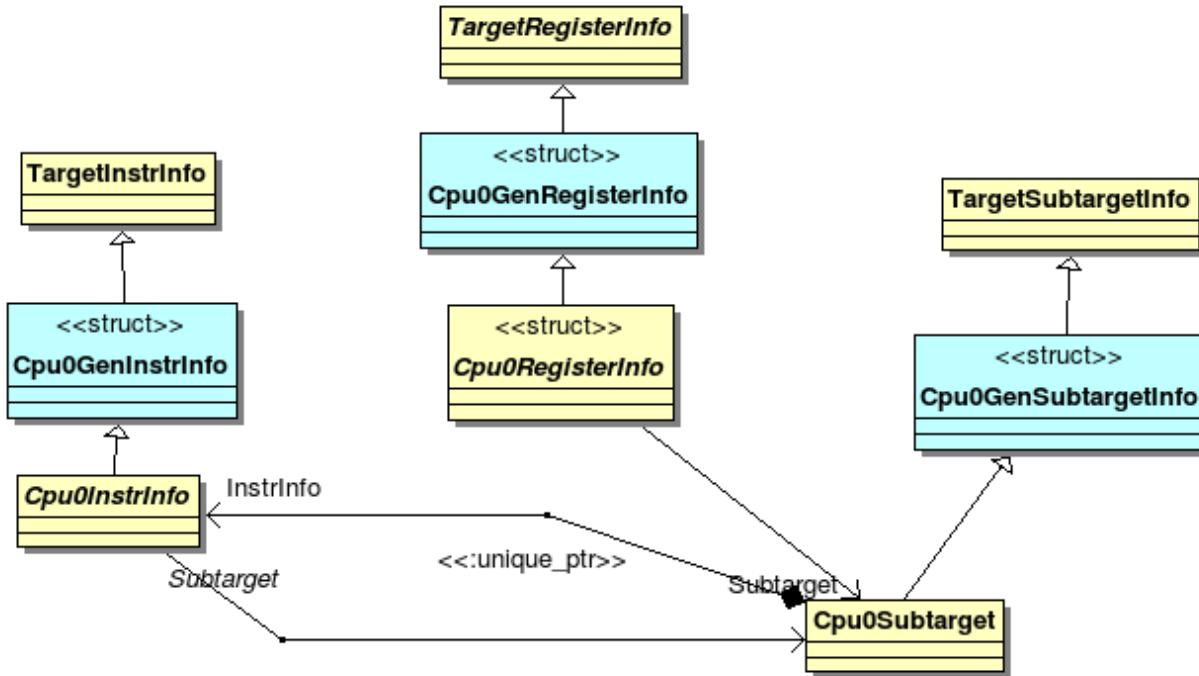


Fig. 3.6: Cpu0 classes inherited from TableGen-generated files

(continued from previous page)

```

explicit Cpu0GenInstrInfo(int SO = -1, int DO = -1);
};

} // End llvm namespace
#endif // GET_INSTRINFO_HEADER
  
```

Reference web sites are here¹².

Chapter3_1/CMakeLists.txt is modified with these new added *.cpp as follows,

Ibdex/chapters/Chapter3_1/CMakeLists.txt

```

tablegen(LLVM Cpu0GenDAGISel.inc -gen-dag-isel)
tablegen(LLVM Cpu0GenCallingConv.inc -gen-callingconv)
  
```

```

Cpu0FrameLowering.cpp
  
```

Please take a look for Chapter3_1 code. After that, building Chapter3_1 by “#define CH CH3_1” in Cpu0Config.h as follows, and do building with cmake and make again.

¹ <http://llvm.org/docs/WritingAnLLVMBBackend.html#target-machine>

² <http://llvm.org/docs/LangRef.html#data-layout>

~/llvm/test/llvm/lib/Target/Cpu0SetChapter.h

```
#define CH      CH3_1
```

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
... Assertion failed: (MRI && "Unable to create reg info"), function initAsmInfo
...
```

With Chapter3_1 implementation, the Chapter2 error message “Could not allocate target machine!” has gone. The new error say that we have not Target AsmPrinter and asm register info. We will add it in next section.

With the implementation of **Chapter3_1**, the **Chapter2** error message “*Could not allocate target machine!*” has been resolved. The new error indicates that we lack **Target AsmPrinter** and **ASM register info**. We will add these in the next section.

Chapter3_1 creates **FeatureCpu032I** and **FeatureCpu032II** for CPUs **cpu032I** and **cpu032II**, respectively. Additionally, it defines two more features: **FeatureCmp** and **FeatureSlt**.

To demonstrate **instruction set design choices**, this book introduces two CPUs. Readers will understand why **MIPS CPUs** use the **SLT** instruction instead of **CMP** after reading the later chapter, “*Control Flow Statement*”.

With the added support for **cpu032I** and **cpu032II** in *Cpu0.td* and *Cpu0InstrInfo.td* from **Chapter3_1**, running the command:

```
llc -march=cpu0 -mcpu=help
```

will display messages as follows:

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=help
Available CPUs for this target:

cpu032I - Select the cpu032I processor.
cpu032II - Select the cpu032II processor.

Available features for this target:

ch10_1 - Enable Chapter instructions..
ch11_1 - Enable Chapter instructions..
ch11_2 - Enable Chapter instructions..
ch14_1 - Enable Chapter instructions..
ch3_1 - Enable Chapter instructions..
ch3_2 - Enable Chapter instructions..
ch3_3 - Enable Chapter instructions..
ch3_4 - Enable Chapter instructions..
ch3_5 - Enable Chapter instructions..
ch4_1 - Enable Chapter instructions..
ch4_2 - Enable Chapter instructions..
ch5_1 - Enable Chapter instructions..
ch6_1 - Enable Chapter instructions..
ch7_1 - Enable Chapter instructions..
ch8_1 - Enable Chapter instructions..
ch8_2 - Enable Chapter instructions..
```

(continues on next page)

(continued from previous page)

```

ch9_1      - Enable Chapter instructions..
ch9_2      - Enable Chapter instructions..
ch9_3      - Enable Chapter instructions..
chall      - Enable Chapter instructions..
cmp        - Enable 'cmp' instructions..
cpu032I    - Cpu032I ISA Support.
cpu032II   - Cpu032II ISA Support (slt).
o32        - Enable o32 ABI.
s32        - Enable s32 ABI.
slt        - Enable 'slt' instructions..

```

Use `+feature` to enable a feature, or `-feature` to disable it.

For example, `llc -mcpu=mycpu -mattr=+feature1,-feature2`

...

When the user inputs `-mcpu(cpu032I)`, the variable `IsCpu032I` from `Cpu0InstrInfo.td` will be **true** since the function `isCpu032I()` defined in `Cpu0Subtarget.h` returns **true**. This happens because `Cpu0ArchVersion` is set to `cpu032I` in `initializeSubtargetDependencies()`, which is called in the constructor. The variable `CPU` in the constructor is “`cpu032I`” when the user inputs `-mcpu(cpu032I)`.

Please note that the variable `Cpu0ArchVersion` must be initialized in `Cpu0Subtarget.cpp`. Otherwise, `Cpu0ArchVersion` may hold an undefined value, causing issues with `isCpu032I()` and `isCpu032II()`, which support `llc -mcpu(cpu032I)` and `llc -mcpu(cpu032II)`, respectively.

The values of the variables `HasCmp` and `HasSlt` depend on `Cpu0ArchVersion`. The instructions `slt`, `beq`, etc., are supported only if `HasSlt` is true. Furthermore, `HasSlt` is **true** only when `Cpu0ArchVersion` is `Cpu032II`.

Similarly, `Ch4_1`, `Ch4_2`, etc., control the enabling or disabling of instruction definitions. Through `Subtarget->hasChapter4_1()`, which exists in both `Cpu0.td` and `Cpu0Subtarget.h`, predicates such as `Ch4_1`, defined in `Cpu0InstrInfo.td`, can be enabled or disabled.

For example, the `shift-rotate` instructions can be enabled by defining `CH` to be greater than or equal to `CH4_1`, as shown below:

Ibdex/Cpu0/Cpu0InstrInfo.td

```

let Predicates = [Ch4_1] in {
class shift_rotate_reg<bits<8> op, bits<4> isRotate, string instr_asm,
                     SDNode OpNode, RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
  !strconcat(instr_asm, "\t$ra, $rb, $rc"),
  [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], IIAlu> {
  let shamt = 0;
}
}

```

`~/llvm/test/llvm/lib/Target/Cpu0SetChapter.h`

```
#define CH        CH4_1
```

On the contrary, it can be disabled by define it to less than `CH4_1`, for instance `CH3_5`, as follows,

~/llvm/test/llvm/lib/Target/Cpu0SetChapter.h

```
#define CH      CH3_5
```

3.2 Add AsmPrinter



Fig. 3.7: When “llc -filetype=asm”, Cpu0AsmPrinter extract MCInst from MachineInstr for asm encoding

As Fig. 3.7, because MachineInstr is a big class for optimization and conversion in many passes. LLVM creates MCInst for encoding purpose in assembly and binary object.

Chapter3_2/ contains the Cpu0AsmPrinter definition.

Ibdex/chapters/Chapter2/Cpu0.td

```
def Cpu0InstrInfo : InstrInfo;

// Will generate Cpu0GenAsmWrite.inc included by Cpu0InstPrinter.cpp, contents
// as follows,
// void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {...}
// const char *Cpu0InstPrinter::getRegisterName(unsigned RegNo) {...}
def Cpu0 : Target {
// def Cpu0InstrInfo : InstrInfo as before.
let InstructionSet = Cpu0InstrInfo;
}
```

As mentioned in the comments of **Chapter2/Cpu0.td**, it will generate **Cpu0GenAsmWriter.inc**, which is included by **Cpu0InstPrinter.cpp** as follows:

Ibdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.h

```
===== Cpu0InstPrinter.h - Convert Cpu0 MCInst to assembly syntax -*- C++ -*-=====
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This class prints a Cpu0 MCInst to a .s file.
//
//=====
```

(continues on next page)

(continued from previous page)

```
#ifndef LLVM_LIB_TARGET_CPU0_INSPRINTER_CPU0INSPRINTER_H
#define LLVM_LIB_TARGET_CPU0_INSPRINTER_CPU0INSPRINTER_H

#include "Cpu0Config.h"

#include "llvm/MC/MCInstPrinter.h"

namespace llvm {
// These enumeration declarations were originally in Cpu0InstrInfo.h but
// had to be moved here to avoid circular dependencies between
// LLVMCpu0CodeGen and LLVMCpu0AsmPrinter.

class TargetMachine;

class Cpu0InstPrinter : public MCInstPrinter {
public:
    Cpu0InstPrinter(const MCAsmInfo &MAI, const MCInstrInfo &MII,
                    const MCRegisterInfo &MRI)
        : MCInstPrinter(MAI, MII, MRI) {}

    // Autogenerated by tblgen.
    std::pair<const char *, uint64_t> getMnemonic(const MCInst *MI) override;
    void printInstruction(const MCInst *MI, uint64_t Address, raw_ostream &O);
    static const char *getRegisterName(unsigned RegNo);

    void printRegName(raw_ostream &OS, unsigned RegNo) const override;
    void printInst(const MCInst *MI, uint64_t Address, StringRef Annot,
                  const MCSubtargetInfo &STI, raw_ostream &O) override;

    bool printAliasInstr(const MCInst *MI, uint64_t Address, raw_ostream &OS);
    void printCustomAliasOperand(const MCInst *MI, uint64_t Address,
                                unsigned OpIdx, unsigned PrintMethodIdx,
                                raw_ostream &O);

private:
    void printOperand(const MCInst *MI, unsigned OpNo, raw_ostream &O);
    void printOperand(const MCInst *MI, uint64_t /*Address*/, unsigned OpNum,
                     raw_ostream &O) {
        printOperand(MI, OpNum, O);
    }
    void printUnsignedImm(const MCInst *MI, int opNum, raw_ostream &O);
    void printMemOperand(const MCInst *MI, int opNum, raw_ostream &O);
    //#if CH >= CH7_1
    void printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O);
    //#endif
};

} // end namespace llvm

#endif
```

Ibdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
//===== Cpu0InstPrinter.cpp - Convert Cpu0 MCInst to assembly syntax =====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----//  
//  
// This class prints an Cpu0 MCInst to a .s file.  
//  
//=====-----//  
  
#include "Cpu0InstPrinter.h"  
  
#include "Cpu0InstrInfo.h"  
#include "llvm/ADT/StringExtras.h"  
#include "llvm/MC/MCE Expr.h"  
#include "llvm/MC/MCInst.h"  
#include "llvm/MC/MCInstrInfo.h"  
#include "llvm/MC/MCSymbol.h"  
#include "llvm/Support/ErrorHandling.h"  
#include "llvm/Support/raw_ostream.h"  
using namespace llvm;  
  
#define DEBUG_TYPE "asm-printer"  
  
#define PRINT_ALIAS_INSTR  
#include "Cpu0GenAsmWriter.inc"  
  
void Cpu0InstPrinter::printRegName(raw_ostream &OS, unsigned RegNo) const {  
    // getRegisterName(RegNo) defined in Cpu0GenAsmWriter.inc which indicate in  
    // Cpu0.td.  
    OS << '$' << StringRef(getRegisterName(RegNo)).lower();  
}  
  
//@1 {  
void Cpu0InstPrinter::printInst(const MCInst *MI, uint64_t Address,  
                               StringRef Annot, const MCSubtargetInfo &STI,  
                               raw_ostream &O) {  
    // Try to print any aliases first.  
    if (!printAliasInstr(MI, Address, O))  
    //@1 }  
    // printInstruction(MI, O) defined in Cpu0GenAsmWriter.inc which came from  
    // Cpu0.td indicate.  
    printInstruction(MI, Address, O);  
    printAnnotation(O, Annot);  
}  
  
void Cpu0InstPrinter::printOperand(const MCInst *MI, unsigned OpNo,  
                                   raw_ostream &O) {
```

(continues on next page)

(continued from previous page)

```

const MCOperand &Op = MI->getOperand(OpNo);
if (Op.isReg()) {
    printRegName(O, Op.getReg());
    return;
}

if (Op.isImm()) {
    O << Op.getImm();
    return;
}

assert(Op.isExpr() && "unknown operand kind in printOperand");
Op.getExpr()->print(O, &MAI, true);
}

void Cpu0InstPrinter::printUnsignedImm(const MCInst *MI, int opNum,
                                      raw_ostream &O) {
    const MCOperand &MO = MI->getOperand(opNum);
    if (MO.isImm())
        O << (unsigned short int)MO.getImm();
    else
        printOperand(MI, opNum, O);
}

void Cpu0InstPrinter::
printMemOperand(const MCInst *MI, int opNum, raw_ostream &O) {
    // Load/Store memory operands -- imm($reg)
    // If PIC target the target is loaded as the
    // pattern ld $t9,%call16($gp)
    printOperand(MI, opNum+1, O);
    O << "(";
    printOperand(MI, opNum, O);
    O << ")";
}

//#if CH >= CH7_1
// The DAG data node, mem_ea of Cpu0InstrInfo.td, cannot be disabled by
// ch7_1, only opcode node can be disabled.
void Cpu0InstPrinter::
printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O) {
    // when using stack locations for not load/store instructions
    // print the same way as all normal 3 operand instructions.
    printOperand(MI, opNum, O);
    O << ", ";
    printOperand(MI, opNum+1, O);
    return;
}
//#endif

```

Ibdex/chapters/Chapter3_2/InstPrinter/CMakeLists.txt

```
add_llvm_component_library(LLVMCpu0AsmPrinter
    Cpu0InstPrinter.cpp

    LINK_COMPONENTS
        Support

    ADD_TO_COMPONENT
        Cpu0
    )
```

Cpu0GenAsmWriter.inc contains the implementations of **Cpu0InstPrinter::printInstruction()** and **Cpu0InstPrinter::getRegisterName()**. Both of these functions are auto-generated based on the information defined in **Cpu0InstrInfo.td** and **Cpu0RegisterInfo.td**.

To enable these functions in our code, we only need to add a class **Cpu0InstPrinter** and include them, as demonstrated in [Chapter3_2](#).

The file **Chapter3_2/Cpu0/InstPrinter/Cpu0InstPrinter.cpp** includes **Cpu0GenAsmWriter.inc** and invokes the auto-generated functions from TableGen.

The process of printing assembly and the interaction between **Cpu0InstPrinter.cpp** and **Cpu0GenAsmWriter.inc** is illustrated in [Fig. 3.8](#).

Cpu0AsmPrinter::emitInstruction() calls **Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI)** to extract **MCInst** from **MachineInstr**.

- AsmPrinter::OutStreamer is nMCAsmStreamer if llc -filetype=asm; AsmPrinter::OutStreamer is nMCObjectStreamer if llc -filetype=obj as [Fig. 5.2](#).
- **Bits** is the format of instruction for **Opcode**, used to print “,” between operands.

The function **Cpu0InstPrinter::printMemOperand()** is defined in **Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp**, as shown above. This function is triggered because **Cpu0InstrInfo.td** defines ‘**let PrintMethod = “printMemOperand”**;’, as shown below.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Address operand
def mem : Operand<i32> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops CPURegs, simm16);
    let EncoderMethod = "getMemEncoding";
}

...
// 32-bit load.
multiclass LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
    bit Pseudo = 0> {
    def #NAME# : LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo>;
}

// 32-bit store.
multiclass StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
    bit Pseudo = 0> {
    def #NAME# : StoreM<op, instr_asm, OpNode, CPURegs, mem, Pseudo>;
}
```

(continues on next page)

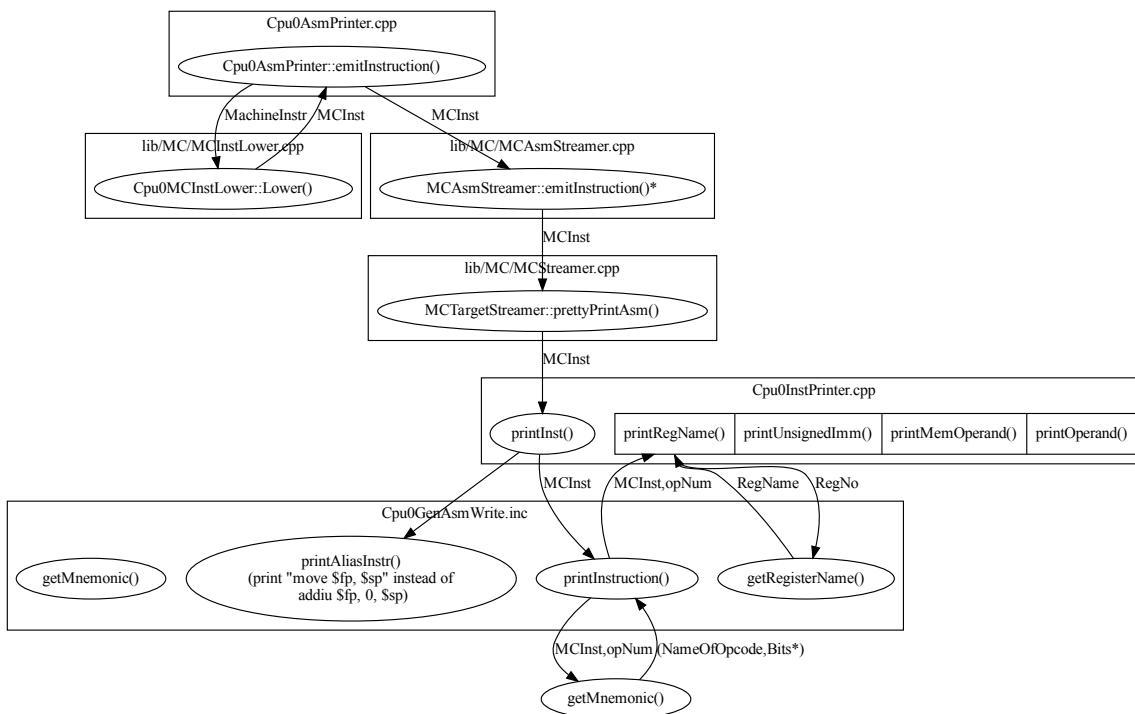


Fig. 3.8: The flow of printing assembly and calling between `Cpu0InstPrinter.cpp` and `Cpu0GenAsmWrite.inc`

(continued from previous page)

```

}

defm LD      : LoadM32<0x01, "ld", load_a>;
defm ST      : StoreM32<0x02, "st", store_a>;

```

Cpu0InstPrinter::printMemOperand() will print backend operands for “local variable access”, which is like the following,

```

ld    $2, 16($fp)
st    $2, 8($fp)

```

Next, add **Cpu0MCInstLower** (**Cpu0MCInstLower.h**, **Cpu0MCInstLower.cpp**) as well as **Cpu0BaseInfo.h**, **Cpu0FixupKinds.h**, and **Cpu0MCAsmInfo** (**Cpu0MCAsmInfo.h**, **Cpu0MCAsmInfo.cpp**) in the sub-directory **MC-TargetDesc**, as shown below.

[Index/chapters/Chapter3_2/Cpu0MCInstLower.h](#)

```

===== Cpu0MCInstLower.h - Lower MachineInstr to MCInst -----* C++ -*===== //
//
//                               The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== //=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0MCINSTLOWER_H
#define LLVM_LIB_TARGET_CPU0_CPU0MCINSTLOWER_H

#include "Cpu0Config.h"

#include "llvm/ADT/SmallVector.h"
#include "llvm/CodeGen/MachineOperand.h"
#include "llvm/Support/Compiler.h"

namespace llvm {
    class MCContext;
    class MCInst;
    class MCOperand;
    class MachineInstr;
    class MachineFunction;
    class Cpu0AsmPrinter;

    // @1 {
    /// This class is used to lower an MachineInstr into an MCInst.
    class LLVM_LIBRARY_VISIBILITY Cpu0MCInstLower {
        // @2
        typedef MachineOperand::MachineOperandType MachineOperandType;
        MCContext *Ctx;
        Cpu0AsmPrinter &AsmPrinter;
    public:
        Cpu0MCInstLower(Cpu0AsmPrinter &asmprinter);
        void Initialize(MCContext* C);

```

(continues on next page)

(continued from previous page)

```

void Lower(const MachineInstr *MI, MCInst &OutMI) const;
MCOperand LowerOperand(const MachineOperand& MO, unsigned offset = 0) const;
};

}

#endif

```

Ibdex/chapters/Chapter3_2/Cpu0MCInstLower.cpp

```

===== Cpu0MCInstLower.cpp - Convert Cpu0 MachineInstr to MCInst =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file contains code to lower Cpu0 MachineInstrs to their corresponding
// MCInst records.
//
//=====

#include "Cpu0MCInstLower.h"

#include "Cpu0AsmPrinter.h"
#include "Cpu0InstrInfo.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstr.h"
#include "llvm/CodeGen/MachineOperand.h"
#include "llvm/IR/Mangler.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCE Expr.h"
#include "llvm/MC/MCInst.h"

using namespace llvm;

Cpu0MCInstLower::Cpu0MCInstLower(Cpu0AsmPrinter &asmprinter)
: AsmPrinter(asmprinter) {}

void Cpu0MCInstLower::Initialize(MCContext* C) {
    Ctx = C;
}

static void CreateMCInst(MCInst& Inst, unsigned Opc, const MCOperand& Opnd0,
                        const MCOperand& Opnd1,
                        const MCOperand& Opnd2 = MCOperand()) {
    Inst.setOpcode(Opc);
    Inst.addOperand(Opnd0);

```

(continues on next page)

(continued from previous page)

```

Inst.addOperand(Opnd1);
if (Opnd2.isValid())
    Inst.addOperand(Opnd2);
}

//@LowerOperand {
MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                         unsigned offset) const {
    MachineOperandType MOTy = MO.getType();

    switch (MOTy) {
        //@2
        default: llvm_unreachable("unknown operand type");
        case MachineOperand::MO_Register:
            // Ignore all implicit register operands.
            if (MO.isImplicit()) break;
            return MCOperand::createReg(MO.getReg());
        case MachineOperand::MO_Immediate:
            return MCOperand::createImm(MO.getImm() + offset);
        case MachineOperand::MO_RegisterMask:
            break;
    }

    return MCOperand();
}

void Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) const {
    OutMI.setOpcode(MI->getOpcode());

    for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
        const MachineOperand &MO = MI->getOperand(i);
        MCOperand MCOp = LowerOperand(MO);

        if (MCOp.isValid())
            OutMI.addOperand(MCOp);
    }
}

```

[Index/chapters/Chapter3_2/MCTargetDesc/Cpu0BaseInfo.h](#)

```

===== Cpu0BaseInfo.h - Top level definitions for CPU0 MC -----*- C++ -*===== //
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== //
//
// This file contains small standalone helper functions and enum definitions for

```

(continues on next page)

(continued from previous page)

```
// the Cpu0 target useful for the compiler back-end and the MC libraries.
//
//=====
#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0BASEINFO_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0BASEINFO_H

#include "Cpu0Config.h"

#include "Cpu0MCTargetDesc.h"
#include "llvm/MC/MCE Expr.h"
#include "llvm/Support/DataTypes.h"
#include "llvm/Support/ErrorHandling.h"

namespace llvm {

/// Cpu0II - This namespace holds all of the target specific flags that
/// instruction info tracks.
// @Cpu0II
namespace Cpu0II {
    /// Target Operand Flag enum.
    enum TOF {
        //=====
        // Cpu0 Specific MachineOperand flags.

        MO_NO_FLAG,

        /// MO_GOT_CALL - Represents the offset into the global offset table at
        /// which the address of a call site relocation entry symbol resides
        /// during execution. This is different from the above since this flag
        /// can only be present in call instructions.
        MO_GOT_CALL,

        /// MO_GPREL - Represents the offset from the current gp value to be used
        /// for the relocatable object file being produced.
        MO_GPREL,

        /// MO_ABS_HI/LO - Represents the hi or low part of an absolute symbol
        /// address.
        MO_ABS_HI,
        MO_ABS_LO,

        /// MO_GOT_HI16/LO16 - Relocations used for large GOTs.
        MO_GOT_HI16,
        MO_GOT_LO16
    }; // enum TOF {

    enum {
        //=====
        // Instruction encodings. These are the standard/most common forms for
        // Cpu0 instructions.
        //

```

(continues on next page)

(continued from previous page)

```
// Pseudo - This represents an instruction that is a pseudo instruction
// or one that has not been implemented yet. It is illegal to code generate
// it, but tolerated for intermediate implementation stages.
Pseudo = 0,

/// FrmR - This form is for instructions of the format R.
FrmR = 1,
/// FrmI - This form is for instructions of the format I.
FrmI = 2,
/// FrmJ - This form is for instructions of the format J.
FrmJ = 3,
/// FrmOther - This form is for instructions that have no specific format.
FrmOther = 4,

FormMask = 15
};

}

}

#endif
```

Ibdex/chapters/Chapter3_2/Cpu0MCAsmInfo.h

```
===== Cpu0MCAsmInfo.h - Cpu0 Asm Info -----*-- C++ -*-----//  

//  

// The LLVM Compiler Infrastructure  

//  

// This file is distributed under the University of Illinois Open Source  

// License. See LICENSE.TXT for details.  

//  

//=====-----//  

//  

// This file contains the declaration of the Cpu0MCAsmInfo class.  

//  

//=====-----//  

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCASMINFO_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCASMINFO_H  

#include "Cpu0Config.h"
#if CH >= CH3_2  

#include "llvm/MC/MCAsmInfoELF.h"  

namespace llvm {
    class Triple;  

    class Cpu0MCAsmInfo : public MCAsmInfoELF {
        void anchor() override;
```

(continues on next page)

(continued from previous page)

```

public:
    explicit Cpu0MCAsmInfo(const Triple &TheTriple);
};

} // namespace llvm

#endif // #if CH >= CH3_2

#endif

```

Ibdex/chapters/Chapter3_2/Cpu0MCAsmInfo.cpp

```

//===== Cpu0MCAsmInfo.cpp - Cpu0 Asm Properties =====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
//
// This file contains the declarations of the Cpu0MCAsmInfo properties.
//
//=====//

#include "Cpu0MCAsmInfo.h"
#if CH >= CH3_2

#include "llvm/ADT/Triple.h"

using namespace llvm;

void Cpu0MCAsmInfo::anchor() { }

Cpu0MCAsmInfo::Cpu0MCAsmInfo(const Triple &TheTriple) {
    if ((TheTriple.getArch() == Triple::cpu0))
        IsLittleEndian = false; // the default of IsLittleEndian is true

    AlignmentIsInBytes      = false;
    Data16bitsDirective     = "\t.2byte\t";
    Data32bitsDirective     = "\t.4byte\t";
    Data64bitsDirective     = "\t.8byte\t";
    PrivateGlobalPrefix      = "$";
    // PrivateLabelPrefix: display $BB for the labels of basic block
    PrivateLabelPrefix       = "$";
    CommentString            = "#";
    ZeroDirective            = "\t.space\t";
    GPRel32Directive         = "\t.gpword\t";
    GPRel64Directive         = "\t.gpdword\t";
    WeakRefDirective          = "\t.weak\t";
    UseAssignmentForEHBegin = true;
}

```

(continues on next page)

(continued from previous page)

```
SupportsDebugInformation = true;
ExceptionsType = ExceptionHandling::DwarfCFI;
DwarfRegNumForCFI = true;
}

#endif // #if CH >= CH3_2
```

Finally, add code in **Cpu0MCTargetDesc.cpp** to register **Cpu0InstPrinter** as shown below. It also registers other classes (register, instruction, and subtarget) that were defined in **Chapter3_1** at this point.

Ibdex/chapters/Chapter3_2/MCTargetDesc/Cpu0MCTargetDesc.h

```
namespace llvm {
class MCAsmBackend;
class MCCodeEmitter;
class MCContext;
class MCInstrInfo;
class MCObjectWriter;
class MCRegisterInfo;
class MCSubtargetInfo;
class StringRef;
...
class raw_ostream;
...
}
```

Ibdex/chapters/Chapter3_2/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
#include "InstPrinter/Cpu0InstPrinter.h"
#include "Cpu0MCAsmInfo.h"

/// Select the Cpu0 Architecture Feature for the given triple and cpu name.
/// The function will be called at command 'llvm-objdump -d' for Cpu0 elf input.
static std::string selectCpu0ArchFeature(const Triple &TT, StringRef CPU) {
    std::string Cpu0ArchFeature;
    if (CPU.empty() || CPU == "generic") {
        if (TT.getArch() == Triple::cpu0 || TT.getArch() == Triple::cpu0el) {
            if (CPU.empty() || CPU == "cpu032II") {
                Cpu0ArchFeature = "+cpu032II";
            }
            else {
                if (CPU == "cpu032I") {
                    Cpu0ArchFeature = "+cpu032I";
                }
            }
        }
    }
    return Cpu0ArchFeature;
}
//@1 }
```

(continues on next page)

(continued from previous page)

```

static MCInstrInfo *createCpu0MCInstrInfo() {
    MCInstrInfo *X = new MCInstrInfo();
    InitCpu0MCInstrInfo(X); // defined in Cpu0GenInstrInfo.inc
    return X;
}

static MCRegisterInfo *createCpu0MCRegisterInfo(const Triple &TT) {
    MCRegisterInfo *X = new MCRegisterInfo();
    InitCpu0MCRegisterInfo(X, Cpu0::SW); // defined in Cpu0GenRegisterInfo.inc
    return X;
}

static MCSubtargetInfo *createCpu0MCSubtargetInfo(const Triple &TT,
                                                 StringRef CPU, StringRef FS) {
    std::string ArchFS = selectCpu0ArchFeature(TT,CPU);
    if (!FS.empty()) {
        if (!ArchFS.empty())
            ArchFS = ArchFS + "," + FS.str();
        else
            ArchFS = FS.str();
    }
    return createCpu0MCSubtargetInfoImpl(TT, CPU, /*TuneCPU*/ CPU, ArchFS);
// createCpu0MCSubtargetInfoImpl defined in Cpu0GenSubtargetInfo.inc
}

static MCAsmInfo *createCpu0MCAsmInfo(const MCRegisterInfo &MRI,
                                       const Triple &TT,
                                       const MCTargetOptions &Options) {
    MCAsmInfo *MAI = new Cpu0MCAsmInfo(TT);

    unsigned SP = MRI.getDwarfRegNum(Cpu0::SP, true);
    MCCFIInstruction Inst = MCCFIInstruction::createDefCfaRegister(nullptr, SP);
    MAI->addInitialFrameState(Inst);

    return MAI;
}

static MCInstPrinter *createCpu0MCInstPrinter(const Triple &T,
                                              unsigned SyntaxVariant,
                                              const MCAsmInfo &MAI,
                                              const MCInstrInfo &MII,
                                              const MCRegisterInfo &MRI) {
    return new Cpu0InstPrinter(MAI, MII, MRI);
}

namespace {

class Cpu0MCInstrAnalysis : public MCInstrAnalysis {
public:
    Cpu0MCInstrAnalysis(const MCInstrInfo *Info) : MCInstrAnalysis(Info) {}
};

```

(continues on next page)

(continued from previous page)

```

}

static MCInstrAnalysis *createCpu0MCInstrAnalysis(const MCInstrInfo *Info) {
    return new Cpu0MCInstrAnalysis(Info);
}

//@2 {
extern "C" void LLVMInitializeCpu0TargetMC() {
    for (Target *T : {&TheCpu0Target, &TheCpu0elTarget}) {
        // Register the MC asm info.
        RegisterMCAsmInfoFn X(*T, createCpu0MCAsmInfo);

        // Register the MC instruction info.
        TargetRegistry::RegisterMCInstrInfo(*T, createCpu0MCInstrInfo);

        // Register the MC register info.
        TargetRegistry::RegisterMCRegInfo(*T, createCpu0MCRegisterInfo);

        // Register the MC subtarget info.
        TargetRegistry::RegisterMCSubtargetInfo(*T,
                                               createCpu0MCSubtargetInfo);
        // Register the MC instruction analyzer.
        TargetRegistry::RegisterMCInstrAnalysis(*T, createCpu0MCInstrAnalysis);
        // Register the MCInstPrinter.
        TargetRegistry::RegisterMCInstPrinter(*T,
                                              createCpu0MCInstPrinter);
    }
}
//@2 }

```

[Index/chapters/Chapter3_2/MCTargetDesc/CMakeLists.txt](#)

Cpu0MCAsmInfo.cpp

To make the registration clearly, summary as the following diagram, Fig. 3.9.

Above `createCpu0MCAsmInfo()` registering the object of class `Cpu0MCAsmInfo` for target `TheCpu0Target` and `TheCpu0elTarget`. `TheCpu0Target` is for big endian and `TheCpu0elTarget` is for little endian. `Cpu0MCAsmInfo` is derived from `MCAsmInfo` which is an LLVM built-in class. Most code is implemented in its parent, backend reuses those code by inheritance.

Above `createCpu0MCAsmInfo()` registers the `Cpu0MCAsmInfo` object for the targets `TheCpu0Target` (big-endian) and `TheCpu0elTarget` (little-endian). `Cpu0MCAsmInfo` is derived from `MCAsmInfo`, an LLVM built-in class. Most of its functionality is inherited from its parent class.

Above `createCpu0MCInstrInfo()` instantiates an `MCInstrInfo` object `X` and initializes it using `InitCpu0MCInstrInfo(X)`. Since `InitCpu0MCInstrInfo(X)` is defined in `Cpu0GenInstrInfo.inc`, this function incorporates the instruction details specified in `Cpu0InstrInfo.td`.

Above `createCpu0MCInstPrinter()` instantiates `Cpu0InstPrinter`, which handles instruction printing.

Above `createCpu0MCRegisterInfo()` follows a similar approach to “Register function of MC instruction info” but initializes register information from `Cpu0RegisterInfo.td`. It reuses values from the instruction/register TableGen descriptions,

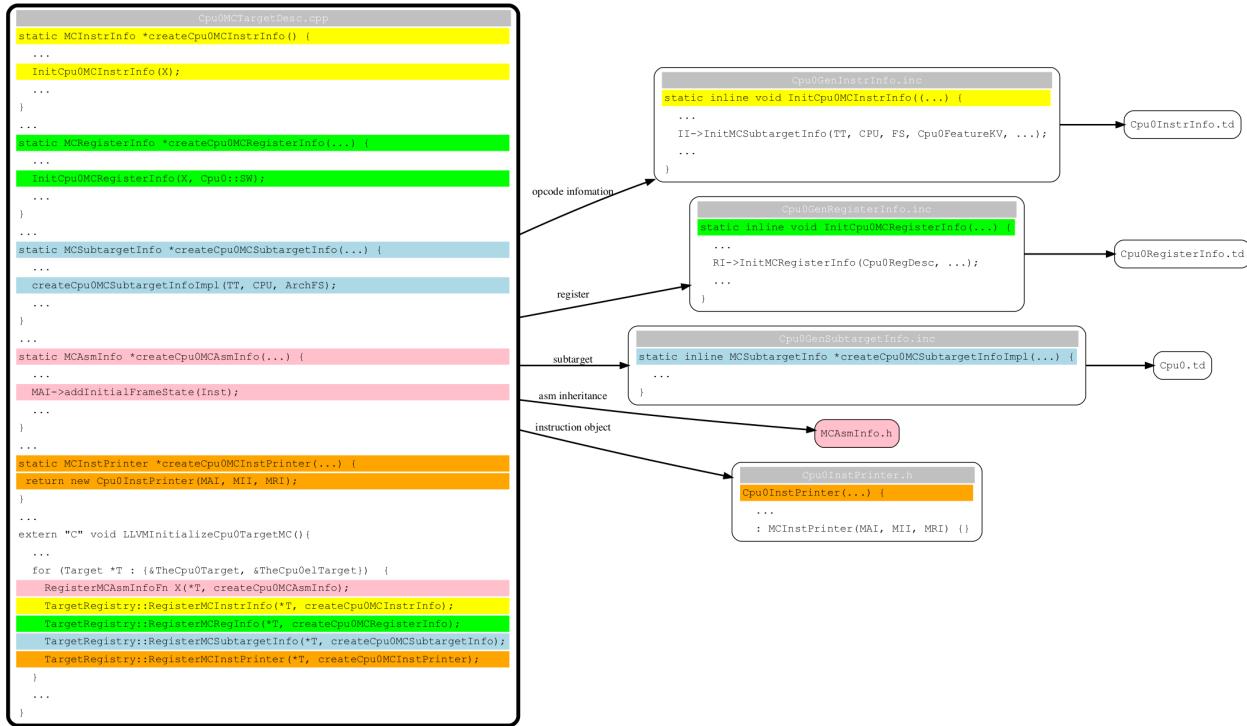


Fig. 3.9: Tblgen generate files for Cpu0 backend

eliminating redundancy in the initialization routine if they remain consistent.

Above `createCpu0MCSubtargetInfo()` instantiates an `MCSubtargetInfo` object and initializes it with details from `Cpu0.td`.

According to “section Target Registration”³, we can register **Cpu0 backend** classes in `LLVMInitializeCpu0TargetMC()` using LLVM’s dynamic registration mechanism.

Now, it’s time to work with **AsmPrinter**, as shown below.

Ibdex/chapters/Chapter3_2/Cpu0AsmPrinter.h

```

//===== Cpu0AsmPrinter.h - Cpu0 LLVM Assembly Printer -----*-- C++ -*--=/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====-----=====
//
// Cpu0 Assembly printer class.
//
//=====-----=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0ASMPRINTER_H
#define LLVM_LIB_TARGET_CPU0_CPU0ASMPRINTER_H

```

(continues on next page)

³ <http://jonathan2251.github.io/lbd/llvmsstructure.html#target-registration>

(continued from previous page)

```
#include "Cpu0Config.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0MCInstLower.h"
#include "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/Support/Compiler.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
class MCStreamer;
class MachineInstr;
class MachineBasicBlock;
class Module;
class raw_ostream;

class LLVM_LIBRARY_VISIBILITY Cpu0AsmPrinter : public AsmPrinter {

    void EmitInstrWithMacroNoAT(const MachineInstr *MI);

private:

    // lowerOperand - Convert a MachineOperand into the equivalent MCOperand.
    bool lowerOperand(const MachineOperand &MO, MCOperand &MCO);

public:

    const Cpu0Subtarget *Subtarget;
    const Cpu0FunctionInfo *Cpu0FI;
    Cpu0MCInstLower MCInstLowering;

    explicit Cpu0AsmPrinter(TargetMachine &TM,
                           std::unique_ptr<MCStreamer> Streamer)
        : AsmPrinter(TM, std::move(Streamer)),
          MCInstLowering(*this) {
        Subtarget = static_cast<Cpu0TargetMachine &>(TM).getSubtargetImpl();
    }

    StringRef getPassName() const override {
        return "Cpu0 Assembly Printer";
    }

    virtual bool runOnMachineFunction(MachineFunction &MF) override;

    // emitInstruction() must exists or will have run time error.
    void emitInstruction(const MachineInstr *MI) override;
    void printSavedRegsBitmask(raw_ostream &O);
    void printHex32(unsigned int Value, raw_ostream &O);
    void emitFrameDirective();
    const char *getCurrentABIString() const;
}
```

(continues on next page)

(continued from previous page)

```
void emitFunctionEntryLabel() override;
void emitFunctionBodyStart() override;
void emitFunctionBodyEnd() override;
void emitStartOfAsmFile(Module &M) override;
void PrintDebugValueComment(const MachineInstr *MI, raw_ostream &OS);
};

}

#endif
```

Index/chapters/Chapter3_2/Cpu0AsmPrinter.cpp

```
//===== Cpu0AsmPrinter.cpp - Cpu0 LLVM Assembly Printer =====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file contains a printer that converts from our internal representation
// of machine-dependent LLVM code to GAS-format CPU0 assembly language.
//
//=====

#include "Cpu0AsmPrinter.h"

#include "InstPrinter/Cpu0InstPrinter.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "Cpu0.h"
#include "Cpu0InstrInfo.h"
#include "llvm/ADT/SmallString.h"
#include "llvm/ADT/StringExtras.h"
#include "llvm/ADT/Twine.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineInstr.h"
#include "llvm/CodeGen/MachineMemOperand.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Mangler.h"
#include "llvm/MC/MCAsmInfo.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetLoweringObjectFile.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-asm-printer"

bool Cpu0AsmPrinter::runOnMachineFunction(MachineFunction &MF) {
    Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    AsmPrinter::runOnMachineFunction(MF);
    return true;
}

//@EmitInstruction {
//- emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
//@EmitInstruction body {
    if (MI->isDebugValue()) {
        SmallString<128> Str;
        raw_svector_ostream OS(Str);

        PrintDebugValueComment(MI, OS);
        return;
    }

    //@print out instruction:
    // Print out both ordinary instruction and bouble instruction
    MachineBasicBlock::const_instr_iterator I = MI->getIterator();
    MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

    do {
        if (I->isPseudo())
            llvm_unreachable("Pseudo opcode found in emitInstruction()");

        MCInst TmpInst0;
        // Call Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) to
        // extracts MCInst from MachineInstr.
        MCInstLowering.Lower(&*I, TmpInst0);
        OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
    } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}
//@EmitInstruction }

//=====
// Cpu0 Asm Directives
//
// -- Frame directive "frame Stackpointer, Stacksize, RARegister"
// Describe the stack frame.
//
// -- Mask directives "(f)mask bitmask, offset"
// Tells the assembler which registers are saved and where.
```

(continues on next page)

(continued from previous page)

```

// bitmask - contain a little endian bitset indicating which registers are
//           saved on function prologue (e.g. with a 0x80000000 mask, the
//           assembler knows the register 31 (RA) is saved at prologue.
// offset   - the position before stack pointer subtraction indicating where
//           the first saved register on prologue is located. (e.g. with a
//
// Consider the following function prologue:
//
//     .frame $fp,48,$ra
//     .mask 0xc0000000,-8
//     addiu $sp, $sp, -48
//     st $ra, 40($sp)
//     st $fp, 36($sp)
//
// With a 0xc0000000 mask, the assembler knows the register 31 (RA) and
// 30 (FP) are saved at prologue. As the save order on prologue is from
// left to right, RA is saved first. A -8 offset means that after the
// stack pointer subtraction, the first register in the mask (RA) will be
// saved at address 48-8=40.
//
//=====//=====
//=====//=====
// Mask directives
//=====//=====
//     .frame $sp,8,$lr
//->    .mask 0x00000000,0
//     .set noreorder
//     .set nomacro

// Create a bitmask with all callee saved registers for CPU or Floating Point
// registers. For CPU registers consider LR, GP and FP for saving if necessary.
void Cpu0AsmPrinter::printSavedRegsBitmask(raw_ostream &O) {
    // CPU and FPU Saved Registers Bitmasks
    unsigned CPUBitmask = 0;
    int CPUTopSavedRegOff;

    // Set the CPU and FPU Bitmasks
    const MachineFrameInfo &MFI = MF->getFrameInfo();
    const TargetRegisterInfo *TRI = MF->getSubtarget().getRegisterInfo();
    const std::vector<CalleeSavedInfo> &CSI = MFI.getCalleeSavedInfo();
    // size of stack area to which FP callee-saved regs are saved.
    unsigned CPUREgSize = TRI->getRegSizeInBits(Cpu0::CPUREgsRegClass) / 8;
    unsigned i = 0, e = CSI.size();

    // Set CPU Bitmask.
    for ( ; i != e; ++i) {
        unsigned Reg = CSI[i].getReg();
        unsigned RegNum = TRI->getEncodingValue(Reg);
        CPUBitmask |= (1 << RegNum);
    }
}

```

(continues on next page)

(continued from previous page)

```
CPUTopSavedRegOff = CPUBitmask ? -CPURegSize : 0;

// Print CPUBitmask
O << "\t.mask \t"; printHex32(CPUBitmask, O);
O << ',' << CPUTopSavedRegOff << '\n';
}

// Print a 32 bit hex number with all numbers.
void Cpu0AsmPrinter::printHex32(unsigned Value, raw_ostream &O) {
    O << "0x";
    for (int i = 7; i >= 0; i--)
        O.write_hex((Value & (0xF << (i*4))) >> (i*4));
}

//=====/
// Frame and Set directives
//=====
//-> .frame $sp,8,$lr
//     .mask 0x00000000,0
//     .set   noreorder
//     .set   nomacro
/// Frame Directive
void Cpu0AsmPrinter::emitFrameDirective() {
    const TargetRegisterInfo &RI = *MF->getSubtarget().getRegisterInfo();

    unsigned stackReg = RI.getFrameRegister(*MF);
    unsigned returnReg = RI.getRARegister();
    unsigned stackSize = MF->getFrameInfo().getStackSize();

    if (OutStreamer->hasRawTextSupport())
        OutStreamer->emitRawText("\t.frame\t$" +
           StringRef(Cpu0InstPrinter::getRegisterName(stackReg)).lower() +
            "," + Twine(stackSize) + ",$" +
           StringRef(Cpu0InstPrinter::getRegisterName(returnReg)).lower());
}

/// Emit Set directives.
const char *Cpu0AsmPrinter::getCurrentABIString() const {
    switch (static_cast<Cpu0TargetMachine &>(TM).getABI().GetEnumValue()) {
    case Cpu0ABIInfo::ABI::O32: return "abiO32";
    case Cpu0ABIInfo::ABI::S32: return "abis32";
    default: llvm_unreachable("Unknown Cpu0 ABI");
    }
}

//          .type main,@function
//->         .ent   main                      # @main
//     main:
void Cpu0AsmPrinter::emitFunctionEntryLabel() {
    if (OutStreamer->hasRawTextSupport())
        OutStreamer->emitRawText("\t.ent\t" + Twine(CurrentFnSym->getName()));
    OutStreamer->emitLabel(CurrentFnSym);
```

(continues on next page)

(continued from previous page)

```

}

//  .frame  $sp,8,$pc
//  .mask   0x00000000,0
//-> .set  noreorder
//@-> .set  nomacro
/// EmitFunctionBodyStart - Targets can override this to emit stuff before
/// the first basic block in the function.
void Cpu0AsmPrinter::emitFunctionBodyStart() {
    MCInstLowering.Initialize(&MF->getContext());
    emitFrameDirective();

    if (OutStreamer->hasRawTextSupport()) {
        SmallString<128> Str;
        raw_svector_ostream OS(Str);
        printSavedRegsBitmask(OS);
        OutStreamer->emitRawText(OS.str());
        OutStreamer->emitRawText(StringRef("\t.set\tnoreorder"));
        OutStreamer->emitRawText(StringRef("\t.set\tnomacro"));
        if (Cpu0FI->get.EmitNOAT())
            OutStreamer->emitRawText(StringRef("\t.set\tnoat"));
    }
}

//-> .set  macro
//-> .set  reorder
//-> .end  main
/// EmitFunctionBodyEnd - Targets can override this to emit stuff after
/// the last basic block in the function.
void Cpu0AsmPrinter::emitFunctionBodyEnd() {
    // There are instruction for this macros, but they must
    // always be at the function end, and we can't emit and
    // break with BB logic.
    if (OutStreamer->hasRawTextSupport()) {
        if (Cpu0FI->get.EmitNOAT())
            OutStreamer->emitRawText(StringRef("\t.set\tat"));
        OutStreamer->emitRawText(StringRef("\t.set\tmacro"));
        OutStreamer->emitRawText(StringRef("\t.set\treorder"));
        OutStreamer->emitRawText("\t.end\t" + Twine(CurrentFnSym->getName()));
    }
}

//      .section .mdebug.abi32
//      .previous
void Cpu0AsmPrinter::emitStartOfAsmFile(Module &M) {
    // FIXME: Use SwitchSection.

    // Tell the assembler which ABI we are using
    if (OutStreamer->hasRawTextSupport())
        OutStreamer->emitRawText("\t.section .mdebug." +
                                Twine(getCurrentABIString()));
}

```

(continues on next page)

(continued from previous page)

```
// return to previous section
if (OutStreamer->hasRawTextSupport())
    OutStreamer->emitRawText(StringRef("\t.previous"));
}

void Cpu0AsmPrinter::PrintDebugValueComment(const MachineInstr *MI,
                                             raw_ostream &OS) {
    // TODO: implement
    OS << "PrintDebugValueComment ()";
}

// Force static initialization.
extern "C" void LLVMInitializeCpu0AsmPrinter() {
    RegisterAsmPrinter<Cpu0AsmPrinter> X(TheCpu0Target);
    RegisterAsmPrinter<Cpu0AsmPrinter> Y(TheCpu0elTarget);
}
```

When an instruction is ready to be printed, the function *Cpu0AsmPrinter::EmitInstruction()* is triggered first. It then calls *OutStreamer.EmitInstruction()* to print the opcode and register names based on the information from *Cpu0GenInstrInfo.inc* and *Cpu0GenRegisterInfo.inc*. Both files are registered dynamically in *LLVMInitializeCpu0TargetMC()*.

Note that *Cpu0InstPrinter.cpp* only prints the operands, while the opcode information comes from *Cpu0InstrInfo.td*.

Add the following code to *Cpu0ISelLowering.cpp*.

Ibdex/chapters/Chapter3_2/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
: TargetLowering(TM, new Cpu0TargetObjectFile()),
  Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {

    // Set .align 2
    // It will emit .align 2 later
    setMinFunctionAlignment(2);

    // must, computeRegisterProperties - Once all of the register classes are
    // added, this allows us to compute derived properties we expose.
    computeRegisterProperties();
}
```

Add the following code to *Cpu0MachineFunction.h* since *Cpu0AsmPrinter.cpp* will call *getEmitNOAT()*.

Ibdex/chapters/Chapter3_2/Cpu0MachineFunction.h

```
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
    Cpu0FunctionInfo(MachineFunction& MF)
    : ...
        , EmitNOAT(false)
    {}
```

(continues on next page)

(continued from previous page)

```

...
bool getEmitNOAT() const { return EmitNOAT; }
void setEmitNOAT() { EmitNOAT = true; }

private:
...

bool EmitNOAT;
};

```

Ibdex/chapters/Chapter3_2/CMakeLists.txt

```

tablegen(LLVM Cpu0GenCodeEmitter.inc -gen-emitter)
tablegen(LLVM Cpu0GenMCCodeEmitter.inc -gen-emitter)

tablegen(LLVM Cpu0GenAsmWriter.inc -gen-asm-writer)

```

```

...
add_llvm_target(Cpu0CodeGen

```

```

Cpu0AsmPrinter.cpp
Cpu0MCInstLower.cpp

```

```

LINK_COMPONENTS

```

```

...

```

```

Cpu0AsmPrinter

```

```

...
)
...

```

```

add_subdirectory(InstPrinter)

```

Now, run *Chapter3_2/Cpu0* for AsmPrinter support, and you will get a new error message as follows,

```

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
/Users/Jonathan/llvm/test/build/bin/llc: target does not
support generation of this file type!

```

The `llc` fails to compile IR code into machine code since we haven't implemented the class `Cpu0DAGToDAGISel`.

3.3 Add Cpu0DAGToDAGISel class

The IR DAG to machine instruction DAG transformation was introduced in the previous chapter.

Now, let's check what IR DAG nodes the file `ch3.ll` contains. List `ch3.ll` as follows:

```
// ch3.ll
define i32 @main() nounwind uwtable {
%1 = alloca i32, align 4
store i32 0, i32* %1
ret i32 0
}
```

As above, *ch3.ll* uses the IR DAG node **store**, **ret**. So, the definitions in *Cpu0InstrInfo.td* as below are enough. The **ADDiu** is used for stack adjustment, which will be needed in the later section “*Add Prologue/Epilogue functions*” of this chapter.

IR DAG is defined in the file *include/llvm/Target/TargetSelectionDAG.td*.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//=====

/// Load and Store Instructions
/// aligned
defm LD      : LoadM32<0x01, "ld", load_a>;
defm ST      : StoreM32<0x02, "st", store_a>;

/// Arithmetic Instructions (ALU Immediate)
// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;

def RET     : RetBase<GPROut>;
```

Add class *Cpu0DAGToDAGISel* (*Cpu0ISelDAGToDAG.cpp*) to *CMakeLists.txt*, and add the following fragment to *Cpu0TargetMachine.cpp*,

Ibdex/chapters/Chapter3_3/CMakeLists.txt

```
add_llvm_target(
  ...
)
```

```
  Cpu0ISelDAGToDAG.cpp
  Cpu0SEISelDAGToDAG.cpp
```

```
  ...
)
```

The following code in *Cpu0TargetMachine.cpp* will create a pass in the instruction selection stage.

Ibdex/chapters/Chapter3_3/Cpu0TargetMachine.cpp

```
#include "Cpu0SEISelDAGToDAG.h"
```

```
...
class Cpu0PassConfig : public TargetPassConfig {
public:
    ...
    bool addInstSelector() override;
};

// Install an instruction selector pass using
// the ISelDag to gen Cpu0 code.
bool Cpu0PassConfig::addInstSelector() {
    addPass(createCpu0SEISelDag(getCpu0TargetMachine(), getOptLevel()));
    return false;
}
```

Index/chapters/Chapter3_3/Cpu0ISelDAGToDAG.h

```
===== Cpu0ISelDAGToDAG.h - A Dag to Dag Inst Selector for Cpu0 =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file defines an instruction selector for the CPU0 target.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0ISELDAGTODAG_H
#define LLVM_LIB_TARGET_CPU0_CPU0ISELDAGTODAG_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/IR/Type.h"
#include "llvm/Support/Debug.h"

//=====
// Instruction Selector Implementation
//=====

//=====
// Cpu0DAGToDAGISel - CPU0 specific code to select CPU0 machine
// instructions for SelectionDAG operations.
```

(continues on next page)

(continued from previous page)

```
//===== /  
namespace llvm {  
  
class Cpu0DAGToDAGISel : public SelectionDAGISel {  
public:  
    explicit Cpu0DAGToDAGISel(Cpu0TargetMachine &TM, CodeGenOpt::Level OL)  
        : SelectionDAGISel(TM, OL), Subtarget(nullptr) {}  
  
    // Pass Name  
    StringRef getPassName() const override {  
        return "CPU0 DAG->DAG Pattern Instruction Selection";  
    }  
  
    bool runOnMachineFunction(MachineFunction &MF) override;  
  
protected:  
  
    /// Keep a pointer to the Cpu0Subtarget around so that we can make the right  
    /// decision when generating code for different targets.  
    const Cpu0Subtarget *Subtarget;  
  
private:  
    // Include the pieces autogenerated from the target description.  
    #include "Cpu0GenDAGISel.inc"  
  
    /// getTargetMachine - Return a reference to the TargetMachine, casted  
    /// to the target-specific type.  
    const Cpu0TargetMachine &getTargetMachine() {  
        return static_cast<const Cpu0TargetMachine &>(TM);  
    }  
  
    void Select(SDNode *N) override;  
  
    virtual bool trySelect(SDNode *Node) = 0;  
  
    // Complex Pattern.  
    bool SelectAddr(SDNode *Parent, SDValue N, SDValue &Base, SDValue &Offset);  
  
    // getImm - Return a target constant with the specified value.  
    inline SDValue getImm(const SDNode *Node, unsigned Imm) {  
        return CurDAG->getTargetConstant(Imm, SDLoc(Node), Node->getValueType(0));  
    }  
  
    virtual void processFunctionAfterISel(MachineFunction &MF) = 0;  
};  
}  
  
#endif
```

Ibdex/chapters/Chapter3_3/Cpu0ISelDAGToDAG.cpp

```

//===== Cpu0ISelDAGToDAG.cpp - A Dag to Dag Inst Selector for Cpu0 =====//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file defines an instruction selector for the CPU0 target.
//
//=====
//

#include "Cpu0ISelDAGToDAG.h"
#include "Cpu0.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0RegisterInfo.h"
#include "Cpu0SEISelDAGToDAG.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/SelectionDAGNodes.h"
#include "llvm/IR/CFG.h"
#include "llvm/IR/GlobalValue.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/IR/Type.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

//=====
// Instruction Selector Implementation
//=====

//=====
// Cpu0DAGToDAGISel - CPU0 specific code to select CPU0 machine
// instructions for SelectionDAG operations.
//=====

bool Cpu0DAGToDAGISel::runOnMachineFunction(MachineFunction &MF) {
    bool Ret = SelectionDAGISel::runOnMachineFunction(MF);
}

```

(continues on next page)

(continued from previous page)

```
    return Ret;
}

//@SelectAddr {
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
//@SelectAddr
    EVT ValTy = Addr.getValueType();
    SDLoc DL(Addr);

    // If Parent is an unaligned f32 load or store, select a (base + index)
    // floating point load/store instruction (luxc1 or suxc1).
    const LSBaseSDNode* LS = 0;

    if (Parent && (LS = dyn_cast<LSBaseSDNode>(Parent))) {
        EVT VT = LS->getMemoryVT();

        if (VT.getSizeInBits() / 8 > LS->getAlignment()) {
            assert(0 && "Unaligned loads/stores not supported for this type.");
            if (VT == MVT::f32)
                return false;
        }
    }

    // if Address is FI, get the TargetFrameIndex.
    if (FrameIndexSDNode *FIN = dyn_cast<FrameIndexSDNode>(Addr)) {
        Base = CurDAG->getTargetFrameIndex(FIN->getIndex(), ValTy);
        Offset = CurDAG->getTargetConstant(0, DL, ValTy);
        return true;
    }

    Base = Addr;
    Offset = CurDAG->getTargetConstant(0, DL, ValTy);
    return true;
}

//@Select {
/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
void Cpu0DAGToDAGISel::Select(SDNode *Node) {
//@Select
    unsigned Opcode = Node->getOpcode();

    // If we have a custom node, we already have selected!
    if (Node->isMachineOpcode()) {
        LLVM_DEBUG(errs() << "==" ; Node->dump(CurDAG); errs() << "\n");
        Node->setNodeId(-1);
        return;
    }
}
```

(continues on next page)

(continued from previous page)

```
// See if subclasses can handle this node.
if (trySelect(Node))
    return;

switch(Opcode) {
default: break;

}

// Select the default instruction
SelectCode(Node);
}
```

Ibdex/chapters/Chapter3_3/Cpu0SEISelDAGToDAG.h

```
===== Cpu0SEISelDAGToDAG.h - A Dag to Dag Inst Selector for Cpu0SE =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//-----
// Subclass of Cpu0DAGToDAGISel specialized for cpu032.
//
//=====

#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEISELDAGTODAG_H
#define LLVM_LIB_TARGET_CPU0_CPU0SEISELDAGTODAG_H

#include "Cpu0Config.h"

#include "Cpu0ISelDAGToDAG.h"

namespace llvm {

class Cpu0SEISelDAGToDAGISel : public Cpu0DAGToDAGISel {

public:
    explicit Cpu0SEISelDAGToDAGISel(Cpu0TargetMachine &TM, CodeGenOpt::Level OL)
        : Cpu0DAGToDAGISel(TM, OL) {}

private:
    bool runOnMachineFunction(MachineFunction &MF) override;

    bool trySelect(SDNode *Node) override;
```

(continues on next page)

(continued from previous page)

```
void processFunctionAfterISel(MachineFunction &MF) override;  
  
    // Insert instructions to initialize the global base register in the  
    // first MBB of the function.  
    // void initGlobalBaseReg(MachineFunction &MF);  
  
};  
  
FunctionPass *createCpu0SEISelDag(Cpu0TargetMachine &TM,  
                                    CodeGenOpt::Level OptLevel);  
  
}  
  
#endif
```

Ibdex/chapters/Chapter3_3/Cpu0ISelDAGToDAG.cpp

```
===== Cpu0SEISelDAGToDAG.cpp - A Dag to Dag Inst Selector for Cpu0SE =====//  
//  
//          The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----//  
//  
// Subclass of Cpu0DAGToDAGISel specialized for cpu032.  
//  
//=====-----//  
  
#include "Cpu0SEISelDAGToDAG.h"  
  
#include "MCTargetDesc/Cpu0BaseInfo.h"  
#include "Cpu0.h"  
#include "Cpu0MachineFunction.h"  
#include "Cpu0RegisterInfo.h"  
#include "llvm/CodeGen/MachineConstantPool.h"  
#include "llvm/CodeGen/MachineFrameInfo.h"  
#include "llvm/CodeGen/MachineFunction.h"  
#include "llvm/CodeGen/MachineInstrBuilder.h"  
#include "llvm/CodeGen/MachineRegisterInfo.h"  
#include "llvm/CodeGen/SelectionDAGNodes.h"  
#include "llvm/IR/CFG.h"  
#include "llvm/IR/GlobalValue.h"  
#include "llvm/IR/Instructions.h"  
#include "llvm/IR/Intrinsics.h"  
#include "llvm/IR/Type.h"  
#include "llvm/Support/Debug.h"  
#include "llvm/Support/ErrorHandling.h"  
#include "llvm/Support/raw_ostream.h"
```

(continues on next page)

(continued from previous page)

```

#include "llvm/Target/TargetMachine.h"
using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

bool Cpu0SEDAGToDAGISel::runOnMachineFunction(MachineFunction &MF) {
    Subtarget = &static_cast<const Cpu0Subtarget &>(MF.getSubtarget());
    return Cpu0DAGToDAGISel::runOnMachineFunction(MF);
}

void Cpu0SEDAGToDAGISel::processFunctionAfterISel(MachineFunction &MF) {
}

//@selectNode
bool Cpu0SEDAGToDAGISel::trySelect(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();
    SDLoc DL(Node);

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
     **/

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
     **/

    EVT NodeTy = Node->getValueType(0);
    unsigned MultOpc;

    switch(Opcode) {
    default: break;
    }

    return false;
}

FunctionPass *llvm::createCpu0SEISelDag(Cpu0TargetMachine &TM,
                                         CodeGenOpt::Level OptLevel) {
    return new Cpu0SEDAGToDAGISel(TM, OptLevel);
}

```

Function *Cpu0DAGToDAGISel::Select()* of *Cpu0ISelDAGToDAG.cpp* is for the selection of “OP code DAG node,” while *Cpu0DAGToDAGISel::SelectAddr()* is for the selection of “DATA DAG node with **addr** type,” which is defined in *Chapter2/Cpu0InstrInfo.td*. This method’s name corresponds to *Chapter2/Cpu0InstrInfo.td* as follows:

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
def addr : ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;
```

The *iPTR*, *ComplexPattern*, *frameindex*, and *SDNPWantParent* are defined as follows:

Ilvm/include/Ilvm/Target/TargetSelection.td

```
def SDNPWantParent : SDNodeProperty; // ComplexPattern gets the parent
...
def frameindex : SDNode<"ISD::FrameIndex", SDTPtrLeaf, [], "FrameIndexSDNode">;
...
// Complex patterns, e.g. X86 addressing mode, requires pattern matching code
// in C++. NumOperands is the number of operands returned by the select function;
// SelectFunc is the name of the function used to pattern match the max. pattern;
// RootNodes are the list of possible root nodes of the sub-dags to match.
// e.g. X86 addressing mode - def addr : ComplexPattern<4, "SelectAddr", [add]>;
//
class ComplexPattern<ValueType ty, int numops, string fn,
    list<SDNode> roots = [], list<SDNodeProperty> props = []> {
    ValueType Ty = ty;
    int NumOperands = numops;
    string SelectFunc = fn;
    list<SDNode> RootNodes = roots;
    list<SDNodeProperty> Properties = props;
}
```

Ilvm/include/Ilvm/CodeGen/ValueTypes.td

```
// Pseudo valuetype mapped to the current pointer size.
def iPTR : Valuetype<0, 255>;
```

Build Chapter3_3 and run it. The error message from Chapter3_2 is gone. The new error message for Chapter3_3 is as follows:

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
...
LLVM ERROR: Cannot select: t6: ch = Cpu0ISD::Ret t4, Register:i32 $lr
    t5: i32 = Register $lr
...
```

The above can display the error message for the DAG node “Cpu0ISD::Ret” because the following code was added in Chapter3_1/Cpu0ISelLowering.cpp.

Ibdex/chapters/Chapter3_1/Cpu0ISelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
    switch (Opcode) {
        case Cpu0ISD::JmpLink:           return "Cpu0ISD::JmpLink";
        case Cpu0ISD::TailCall:          return "Cpu0ISD::TailCall";
```

(continues on next page)

(continued from previous page)

```

case Cpu0ISD::Hi:           return "Cpu0ISD::Hi";
case Cpu0ISD::Lo:           return "Cpu0ISD::Lo";
case Cpu0ISD::GPRel:        return "Cpu0ISD::GPRel";
case Cpu0ISD::Ret:          return "Cpu0ISD::Ret";
case Cpu0ISD::EH_RETURN:    return "Cpu0ISD::EH_RETURN";
case Cpu0ISD::DivRem:       return "Cpu0ISD::DivRem";
case Cpu0ISD::DivRemU:      return "Cpu0ISD::DivRemU";
case Cpu0ISD::Wrapper:      return "Cpu0ISD::Wrapper";

default:                  return NULL;
}

```

3.4 Handle return register \$lr

The following code is the result of running the Mips backend with ch3.cpp.

```

Jonathan@tekiiMac:~/input$ Jonathan$ ~/llvm/debug/build/bin/llc
-march=mips -relocation-model=pic -filetype=asm ch3.bc -o -
.text
.abicalls
.section .mdebug.abi32,"",@progbits
.nan legacy
.file "ch3.bc"
.text
.globl main
.align 2
.type main,@function
.set nomicromips
.set nomips16
.ent main
main:                      # @main
.frame $fp,8,$ra
.mask 0x40000000,-4
.fmask 0x00000000,0
.set noreorder
.set nomacro
.set noat
# BB#0:
addiu $sp, $sp, -8
sw $fp, 4($sp)             # 4-byte Folded Spill
move $fp, $sp
sw $zero, 0($fp)
addiu $2, $zero, 0
move $sp, $fp
lw $fp, 4($sp)             # 4-byte Folded Reload
jr $ra
addiu $sp, $sp, 8
.set at
.set macro
.set reorder

```

(continues on next page)

(continued from previous page)

```
.end main
$func_end0:
.size main, ($func_end0)-main
```

As you can see, Mips returns to the caller by using “jr \$ra”, where \$ra is a specific register that holds the caller’s next instruction address. It also stores the return value in register \$2.

If we only create DAGs directly, we encounter the following two problems:

1. LLVM can allocate any register for the return value, such as \$3, rather than keeping it in \$2.
2. LLVM may randomly allocate a register for “jr” since “jr” requires one operand. For example, it might generate “jr \$8” instead of “jr \$ra”.

If the backend strictly uses the “jal sub-routine” and “jr” while always storing the return address in the specific register \$ra, the second problem does not occur. However, in Mips, programmers are allowed to use “jal \$rx, sub-routine” and “jr \$rx”, where \$rx is not necessarily \$ra.

Allowing programmers to use registers other than \$ra provides more flexibility for high-level languages such as C when integrating assembly.

The following file, *ch8_2_longbranch.cpp*, demonstrates this concept. It uses “jr \$1” without spilling the \$ra register. This optimization can significantly improve performance, especially in hot functions.

Ibdex/input/ch8_2_longbranch.cpp

```
int test_longbranch()
{
    volatile int a = 2;
    volatile int b = 1;
    int result = 0;

    if (a < b)
        result = 1;
    return result;
}
```

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch8_2_longbranch.cpp -emit-llvm -o ch8_2_longbranch.bc
JonathantekiiMac:input Jonathan$ ~/llvm/debug/build/bin/llc
-march=mips -relocation-model=pic -filetype=asm -force-mips-long-branch
ch8_2_longbranch.bc -o -
...
.ent _Z15test_longbranchv
_Z15test_longbranchv:                                # @_Z15test_longbranchv
.frame $fp,16,$ra
.mask 0x40000000,-4
.fmask 0x00000000,0
.set noreorder
.set nomacro
.set noat
# BB#0:
addiu $sp, $sp, -16
sw $fp, 12($sp)          # 4-byte Folded Spill
```

(continues on next page)

(continued from previous page)

```

move    $fp, $sp
addiu $1, $zero, 2
sw    $1, 8($fp)
addiu $2, $zero, 1
sw    $2, 4($fp)
sw    $zero, 0($fp)
lw    $1, 8($fp)
lw    $3, 4($fp)
slt   $1, $1, $3
bnez  $1, $BB0_3
nop
# BB#1:
addiu $sp, $sp, -8
sw    $ra, 0($sp)
lui   $1, %hi(($BB0_4)-($BB0_2))
bal   $BB0_2
addiu $1, $1, %lo(($BB0_4)-($BB0_2))
$BB0_2:
addu  $1, $ra, $1
lw    $ra, 0($sp)
jr   $1
addiu $sp, $sp, 8
$BB0_3:
sw    $2, 0($fp)
$BB0_4:
lw    $2, 0($fp)
move  $sp, $fp
lw    $fp, 12($sp)           # 4-byte Folded Reload
jr   $ra
addiu $sp, $sp, 16
.set  at
.set  macro
.set  reorder
.end  _Z15test_longbranchv
$func_end0:
.size _Z15test_longbranchv, ($func_end0)-_Z15test_longbranchv

```

The following code handles the return register \$lr.

Ibdex/chapters/Chapter3_4/Cpu0CallingConv.td

```

def RetCC_Cpu0EABI : CallingConv<[
    // i32 are returned in registers V0, V1, A0, A1
    CCIfType<i32>, CCAssignToReg<[V0, V1, A0, A1]>>
]>;

```

```

def RetCC_Cpu0 : CallingConv<[
    CCDelegateTo<RetCC_Cpu0EABI>
]>;

```

Ibdex/chapters/Chapter3_4/Cpu0InstrFormats.td

```
// Cpu0 Pseudo Instructions Format
class Cpu0Pseudo<dag outs, dag ins, string asmstr, list<dag> pattern>:
    Cpu0Inst<outs, ins, asmstr, pattern, IIPseudo, Pseudo> {
let isCodeGenOnly = 1;
let isPseudo = 1;
}
```

Ibdex/chapters/Chapter3_4/Cpu0InstrInfo.td

```
let Predicates = [Ch3_4] in {
let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
    def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
}
```

Ibdex/chapters/Chapter3_4/Cpu0ISelLowering.h

```
/// Cpu0CC - This class provides methods used to analyze formal and call
/// arguments and inquire about calling convention information.
class Cpu0CC {
public:
    enum SpecialCallingConvType {
        NoSpecialCallingConv
    };

    Cpu0CC(CallingConv::ID CallConv, bool IsO32, CCState &Info,
           SpecialCallingConvType SpecialCallingConv = NoSpecialCallingConv);

    void analyzeCallResult(const SmallVectorImpl<ISD::InputArg> &Ins,
                          bool IsSoftFloat, const SDNode *CallNode,
                          const Type *RetTy) const;

    void analyzeReturn(const SmallVectorImpl<ISD::OutputArg> &Outs,
                      bool IsSoftFloat, const Type *RetTy) const;

    const CCState &getCCInfo() const { return CCInfo; }

    /// hasByValArg - Returns true if function has byval arguments.
    bool hasByValArg() const { return !ByValArgs.empty(); }

    /// reservedArgArea - The size of the area the caller reserves for
    /// register arguments. This is 16-byte if ABI is O32.
    unsigned reservedArgArea() const;

    typedef SmallVectorImpl<ByValArgInfo>::const_iterator byval_iterator;
    byval_iterator byval_begin() const { return ByValArgs.begin(); }
    byval_iterator byval_end() const { return ByValArgs.end(); }

private:
    /// Return the type of the register which is used to pass an argument or
```

(continues on next page)

(continued from previous page)

```

/// return a value. This function returns f64 if the argument is an i64
/// value which has been generated as a result of softening an f128 value.
/// Otherwise, it just returns VT.
MVT getRegVT(MVT VT, bool IsSoftFloat) const;

template<typename Ty>
void analyzeReturn(const SmallVectorImpl<Ty> &RetVals, bool IsSoftFloat,
                   const SDNode *CallNode, const Type *RetTy) const;

CCState &CCInfo;
CallingConv::ID CallConv;
bool IsO32;
SmallVector<ByValArgInfo, 2> ByValArgs;
};

```

Index/chapters/Chapter3_4/Cpu0ISelLowering.cpp

```

SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                                 CallingConv::ID CallConv, bool IsVarArg,
                                 const SmallVectorImpl<ISD::OutputArg> &Outs,
                                 const SmallVectorImpl<SDValue> &OutVals,
                                 const SDLoc &DL, SelectionDAG &DAG) const {

```

```

// CCValAssign - represent the assignment of
// the return value to a location
SmallVector<CCValAssign, 16> RVLocs;
MachineFunction &MF = DAG.getMachineFunction();

// CCState - Info about the registers and stack slot.
CCState CCInfo(CallConv, IsVarArg, MF, RVLocs,
               *DAG.getContext());
Cpu0CC Cpu0CCInfo(CallConv, ABI.IsO32(),
                  CCInfo);

// Analyze return values.
Cpu0CCInfo.analyzeReturn(Outs, Subtarget.abiUsesSoftFloat(),
                        MF.getFunction().getReturnType());

SDValue Flag;
SmallVector<SDValue, 4> RetOps(1, Chain);

// Copy the result values into the output registers.
for (unsigned i = 0; i != RVLocs.size(); ++i) {
    SDValue Val = OutVals[i];
    CCValAssign &VA = RVLocs[i];
    assert(VA.isRegLoc() && "Can only return in registers!");

    if (RVLocs[i].getValVT() != RVLocs[i].getLocVT())
        Val = DAG.getNode(ISD::BITCAST, DL, RVLocs[i].getLocVT(), Val);
}

```

(continues on next page)

(continued from previous page)

```

Chain = DAG.getCopyToReg(Chain, DL, VA.getLocReg(), Val, Flag);

// Guarantee that all emitted copies are stuck together with flags.
Flag = Chain.getValue(1);
RetOps.push_back(DAG.getRegister(VA.getLocReg(), VA.getLocVT()));
}

//@Ordinary struct type: 2 {
// The cpu0 ABIs for returning structs by value requires that we copy
// the sret argument into $v0 for the return. We saved the argument into
// a virtual register in the entry block, so now we copy the value out
// and into $v0.
if (MF.getFunction().hasStructRetAttr()) {
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    unsigned Reg = Cpu0FI->getSRetReturnReg();

    if (!Reg)
        llvm_unreachable("sret virtual register not created in the entry block");
    SDValue Val =
        DAG.getCopyFromReg(Chain, DL, Reg, getPointerTy(DAG.getDataLayout()));
    unsigned V0 = Cpu0::V0;

    Chain = DAG.getCopyToReg(Chain, DL, V0, Val, Flag);
    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(V0, getPointerTy(DAG.getDataLayout())));
}
//@Ordinary struct type: 2 }

RetOps[0] = Chain; // Update chain.

// Add the flag if we have it.
if (Flag.getNode())
    RetOps.push_back(Flag);

// Return on Cpu0 is always a "ret $lr"
return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other, RetOps);
}

```

```

template<typename Ty>
void Cpu0TargetLowering::Cpu0CC::
analyzeReturn(const SmallVectorImpl<Ty> &RetVals, bool IsSoftFloat,
             const SDNode *CallNode, const Type *RetTy) const {
    CCAssignFn *Fn;

    Fn = RetCC_Cpu0;

    for (unsigned I = 0, E = RetVals.size(); I < E; ++I) {
        MVT VT = RetVals[I].VT;
        ISD::ArgFlagsTy Flags = RetVals[I].Flags;
        MVT RegVT = this->getRegVT(VT, IsSoftFloat);

```

(continues on next page)

(continued from previous page)

```

if (Fn(I, VT, RegVT, CCValAssign::Full, Flags, this->CCInfo)) {
#ifndef NDEBUG
    dbgs() << "Call result #" << I << " has unhandled type "
        << EVT(VT).getEVTString() << '\n';
#endif
    llvm_unreachable(nullptr);
}
}

void Cpu0TargetLowering::Cpu0CC::
analyzeCallResult(const SmallVectorImpl<ISD::InputArg> &Ins, bool IsSoftFloat,
                  const SDNode *CallNode, const Type *RetTy) const {
    analyzeReturn(Ins, IsSoftFloat, CallNode, RetTy);
}

void Cpu0TargetLowering::Cpu0CC::
analyzeReturn(const SmallVectorImpl<ISD::OutputArg> &Outs, bool IsSoftFloat,
             const Type *RetTy) const {
    analyzeReturn(Outs, IsSoftFloat, nullptr, RetTy);
}

```

```

unsigned Cpu0TargetLowering::Cpu0CC::reservedArgArea() const {
    return (IsO32 && (CallConv != CallingConv::Fast)) ? 8 : 0;
}

```

```

MVT Cpu0TargetLowering::Cpu0CC::getRegVT(MVT VT,
                                         bool IsSoftFloat) const {
if (IsSoftFloat || IsO32)
    return VT;

return VT;
}

```

Ibdex/chapters/Chapter3_4/Cpu0MachineFunction.h

```

/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {

```

```

SRetReturnReg(0), CallsEhReturn(false), CallsEhDwarf(false),

```

```

unsigned getSRetReturnReg() const { return SRetReturnReg; }
void setSRetReturnReg(unsigned Reg) { SRetReturnReg = Reg; }

```

```

bool hasByvalArg() const { return HasByvalArg; }
void setFormalArgInfo(unsigned Size, bool HasByval) {
    IncomingArgSize = Size;
    HasByvalArg = HasByval;
}

```

(continues on next page)

(continued from previous page)

```
}
```

```
/// SRetReturnReg - Some subtargets require that sret lowering includes
/// returning the value of the returned struct in a register. This field
/// holds the virtual register into which the sret argument is passed.
unsigned SRetReturnReg;
```

```
/// True if function has a byval argument.
bool HasByvalArg;

/// Size of incoming argument area.
unsigned IncomingArgSize;

/// CallsEhReturn - Whether the function calls llvm.eh.return.
bool CallsEhReturn;

/// CallsEhDwarf - Whether the function calls llvm.eh.dwarf.
bool CallsEhDwarf;

/// Frame objects for spilling eh data registers.
int EhDataRegFI[2];
```

```
}
```

Ibdex/chapters/Chapter3_4/Cpu0SEInstrInfo.h

```
//@expandPostRAPpseudo
bool expandPostRAPpseudo(MachineInstr &MI) const override;

private:
    void expandRetLR(MachineBasicBlock &MBB, MachineBasicBlock::iterator I) const;
```

Ibdex/chapters/Chapter3_4/Cpu0SEInstrInfo.cpp

```
//@expandPostRAPpseudo
/// Expand Pseudo instructions into real backend instructions
bool Cpu0SEInstrInfo::expandPostRAPpseudo(MachineInstr &MI) const {
    // @expandPostRAPpseudo-body
    MachineBasicBlock &MBB = *MI.getParent();

    switch (MI.getDesc().getOpcode()) {
        default:
            return false;
        case Cpu0::RetLR:
            expandRetLR(MBB, MI);
            break;
    }
}
```

(continues on next page)

(continued from previous page)

```
    MBB.erase(MI);
    return true;
}
```

```
void Cpu0SEInstrInfo::expandRetLR(MachineBasicBlock &MBB,
                                   MachineBasicBlock::iterator I) const {
    BuildMI(MBB, I, I->getDebugLoc(), get(Cpu0::RET)).addReg(Cpu0::LR);
}
```

Build Chapter3_4 and run with it, finding the error message in Chapter3_3 is gone. The compilation result will hang, and please press “Ctrl+C” to abort as follows,

```
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-230:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch3.bc -o -
...
define i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    ret i32 0
}

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
...
.text
.section .mdebug.abi032
.previous
.file "ch3.bc"
^C
```

It hangs because the Cpu0 backend has not handled stack slots for local variables. The instruction “store i32 0, i32* %1” in the above IR requires Cpu0 to allocate a stack slot and save to it.

However, ch3.cpp can be run with the option `clang -O2` as follows,

```
118-165-78-230:input Jonathan$ clang -O2 -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-230:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch3.bc -o -
...
define i32 @main() #0 {
    ret i32 0
}

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
.text
.section .mdebug.abi032
.previous
.file "ch3.bc"
.globl main
```

(continues on next page)

(continued from previous page)

```
.align 2
.type main,@function
.ent main                      # @main
main:
.frame $sp,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $2, $zero, 0
ret $lr
.set macro
.set reorder
.end main
$func_end0:
.size main, ($func_end0)-main
```

To see how the ‘**DAG->DAG Pattern Instruction Selection**’ works in llc, let’s compile with the option `llc -print-before-all -print-after-all` and get the following result.

The DAGs before and after the instruction selection stage are shown below,

```
118-165-78-230:input Jonathan$ clang -O2 -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
-print-before-all -print-after-all ch3.bc -o -
...
*** IR Dump After Module Verifier ***
; Function Attrs: nounwind readnone
define i32 @main() #0 {
    ret i32 0
}
...
Initial selection DAG: BB#0 'main:'
SelectionDAG has 5 nodes:
    t0: ch = EntryToken
    t3: ch,glue = CopyToReg t0, Register:i32 %V0, Constant:i32<0>
    t4: ch = Cpu0ISD::Ret t3, Register:i32 %V0, t3:1
...
===== Instruction selection begins: BB#0 ''
Selecting: t4: ch = Cpu0ISD::Ret t3, Register:i32 %V0, t3:1

ISEL: Starting pattern match on root node: t4: ch = Cpu0ISD::Ret t3, Register:i32 %V0,
→ t3:1

Morphed node: t4: ch = RetLR Register:i32 %V0, t3, t3:1

ISEL: Match complete!
Selecting: t3: ch,glue = CopyToReg t0, Register:i32 %V0, Constant:i32<0>

Selecting: t2: i32 = Register %V0
```

(continues on next page)

(continued from previous page)

```
Selecting: t1: i32 = Constant<0>

ISEL: Starting pattern match on root node: t1: i32 = Constant<0>

    Initial Opcode index to 3158
    Morphed node: t1: i32 = ADDiu Register:i32 %ZERO, TargetConstant:i32<0>

ISEL: Match complete!
Selecting: t0: ch = EntryToken

===== Instruction selection ends:
Selected selection DAG: BB#0 'main:'
SelectionDAG has 7 nodes:
    t0: ch = EntryToken
    t1: i32 = ADDiu Register:i32 %ZERO, TargetConstant:i32<0>
    t3: ch,glue = CopyToReg t0, Register:i32 %V0, t1
    t4: ch = RetLR Register:i32 %V0, t3, t3:1
    ...
***** REWRITE VIRTUAL REGISTERS *****
***** Function: main
***** REGISTER MAP *****
[%vreg0 -> %V0] GPROut

0B BB#0: derived from LLVM BB %0
16B  %vreg0<def> = ADDiu %ZERO, 0; GPROut:%vreg0
32B  %V0<def> = COPY %vreg0<kill>; GPROut:%vreg0
48B  RetLR %V0<imp-use>
> %V0<def> = ADDiu %ZERO, 0
> %V0<def> = COPY %V0<kill>
Identity copy: %V0<def> = COPY %V0<kill>
    deleted.
> RetLR %V0<imp-use>
# *** IR Dump After Virtual Register Rewriter ***:
# Machine code for function main: Properties: <Post SSA, tracking liveness, ↵
    AllVRegsAllocated>

0B BB#0: derived from LLVM BB %0
16B  %V0<def> = ADDiu %ZERO, 0
48B  RetLR %V0<imp-use>
...
***** EXPANDING POST-RA PSEUDO INSTRS *****
***** Function: main
# *** IR Dump After Post-RA pseudo instruction expansion pass ***:
# Machine code for function main: Properties: <Post SSA, tracking liveness, ↵
    AllVRegsAllocated>

BB#0: derived from LLVM BB %0
%V0<def> = ADDiu %ZERO, 0
RET %LR
...
.globl main
```

(continues on next page)

(continued from previous page)

```
.p2align 2
.type main,@function
.ent main          # @main
main:
.frame $sp,0,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $2, $zero, 0
ret $lr
.set macro
.set reorder
.end main
$func_end0:
.size main, ($func_end0)-main

.ident "Apple LLVM version 7.0.0 (clang-700.1.76)"
.section ".note.GNU-stack","",@progbits
```

Summary above translation into Table: Chapter 3 .bc IR instructions.

Table 3.1: Chapter 3 .bc IR instructions

.bc	Lower	ISel	RVR	Post-RA	AsmP
constant 0	constant 0	ADDiu	ADDiu	ADDiu	addiu
ret	Cpu0ISD::Ret	CopyToReg,RetLR	RetLR	RET	ret

- Lower: Initial selection DAG (Cpu0ISelLowering.cpp, LowerReturn(...))
- ISel: Instruction selection
- RVR: REWRITE VIRTUAL REGISTERS, remove CopyToReg
- AsmP: Cpu0 Asm Printer
- Post-RA: Post-RA pseudo instruction expansion pass

From the above llc -print-before-all -print-after-all display, we observe that **ret** is translated into **Cpu0ISD::Ret** in the stage of Optimized Legalized Selection DAG, and then finally translated into the Cpu0 instruction **ret**.

Since **ret** uses **constant 0** (**ret i32 0** in this example), the constant 0 is translated into “**addiu \$2, \$zero, 0**” via the following pattern defined in **Cpu0InstrInfo.td**.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
      (ADDiu ZERO, imm:$in)>;
```

In order to handle the IR **ret**, the following codes in **Cpu0InstrInfo.td** perform the following tasks:

1. Declare a pseudo node **Cpu0::RetLR** to handle the IR **Cpu0ISD::Ret** with the following code:

Ibdex/chapters/Chapter3_4/Cpu0InstrInfo.td

```
// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,
[SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

let Predicates = [Ch3_4] in {
let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
  def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
}
```

2. Create the **Cpu0ISD::Ret** node in **LowerReturn()** of **Cpu0ISelLowering.cpp**, which is called when encountering the **return** keyword in C. Reminder: In **LowerReturn()**, the return value is placed in register **\$2 (\$v0)**.

If we use the following code in Cpu0, then V0 won't **live out**.

Ibdex/chapters/Chapter3_4/Cpu0ISelLowering.cpp

```
SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                                CallingConv::ID CallConv, bool IsVarArg,
                                const SmallVectorImpl<ISD::OutputArg> &Outs,
                                const SmallVectorImpl<SDValue> &OutVals,
                                const SDLoc &DL, SelectionDAG &DAG) const {

  return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other,
                     Chain, DAG.getRegister(Cpu0::LR, MVT::i32));
}

}
```

3. After instruction selection, the **Cpu0ISD::Ret** node is replaced by **Cpu0::RetLR** as shown below. This effect comes from the "**def RetLR**" declaration in step 1.

```
===== Instruction selection begins: BB#0 'entry'
Selecting: 0x1ea4050: ch = Cpu0ISD::Ret 0x1ea3f50, 0x1ea3e50,
0x1ea3f50:1 [ID=27]

ISEL: Starting pattern match on root node: 0x1ea4050: ch = Cpu0ISD::Ret
0x1ea3f50, 0x1ea3e50, 0x1ea3f50:1 [ID=27]

Morphed node: 0x1ea4050: ch = RetLR 0x1ea3e50, 0x1ea3f50, 0x1ea3f50:1
...
ISEL: Match complete!
=> 0x1ea4050: ch = RetLR 0x1ea3e50, 0x1ea3f50, 0x1ea3f50:1
...
===== Instruction selection ends:
Selected selection DAG: BB#0 'main:entry'
SelectionDAG has 28 nodes:
...
  0x1ea3e50: <multiple use>
  0x1ea3f50: <multiple use>
  0x1ea3f50: <multiple use>
  0x1ea4050: ch = RetLR 0x1ea3e50, 0x1ea3f50, 0x1ea3f50:1
```

4. Expand the **Cpu0ISD::RetLR** into the instruction **Cpu0::RET \$lr** during the “Post-RA pseudo instruction expansion pass” stage using the code in **Chapter3_4/Cpu0SEInstrInfo.cpp** as mentioned above. This stage occurs after register allocation, so we can replace **V0 (\$r2)** with **LR (\$lr)** without any side effects.
5. Print assembly or object code based on the information from **.inc** files generated by TableGen from **.td** files during the “Cpu0 Assembly Print” stage.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//@JumpFR {
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
    FL<op, (outs), (ins RC:$ra),
        !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
let rb = 0;
let imm16 = 0;

}

def RET      : RetBase<GPROut>;
```

Table 3.2: Handle return register lr

Stage	Function
Write Code	Declare a pseudo node Cpu0::RetLR for IR Cpu0::Ret;
• Before CPU0 DAG->DAG Pattern Instruction Selection	Create Cpu0ISD::Ret DAG
Instruction selection	Cpu0::Ret is replaced by Cpu0::RetLR
Post-RA pseudo instruction expansion pass	Cpu0::RetLR -> Cpu0::RET \$lr
Cpu0 Assembly Printer	Print according “def RET”

The function **LowerReturn()** in *Cpu0ISelLowering.cpp* correctly handles the return variable.

In **Chapter3_4/Cpu0ISelLowering.cpp**, the function **LowerReturn()** creates the **Cpu0ISD::Ret** node, which is called by the LLVM system when it encounters the *return* keyword in C.

More specifically, it constructs the DAG as follows:

Cpu0ISD::Ret (CopyToReg %X, %V0, %Y, %V0, Flag)

3.5 Add Prologue/Epilogue functions

3.5.1 Concept

Following come from tricore_llvm.pdf section “4.4.2 Non-static Register Information”.

For some target architectures, some aspects of the target architecture’s register set are dependent upon variable factors and have to be determined at runtime. As a consequence, they cannot be generated statically from a TableGen description —although that would be possible for the bulk of them in the case of the TriCore backend. Among them are the following points:

- Callee-saved registers. Normally, the ABI specifies a set of registers that a function must save on entry and restore on return if their contents are possibly modified during execution.

- Reserved registers. Although the set of unavailable registers is already defined in the TableGen file, TriCoreRegisterInfo contains a method that marks all non-allocatable register numbers in a bit vector.

The following methods are implemented:

- emitPrologue() inserts prologue code at the beginning of a function. Thanks to TriCore's context model, this is a trivial task as it is not required to save any registers manually. The only thing that has to be done is reserving space for the function's stack frame by decrementing the stack pointer. In addition, if the function needs a frame pointer, the frame register %a14 is set to the old value of the stack pointer beforehand.
- emitEpilogue() is intended to emit instructions to destroy the stack frame and restore all previously saved registers before returning from a function. However, as %a10 (stack pointer), %a11 (return address), and %a14 (frame pointer, if any) are all part of the upper context, no epilogue code is needed at all. All cleanup operations are performed implicitly by the ret instruction.
- eliminateFrameIndex() is called for each instruction that references a word of data in a stack slot. All previous passes of the code generator have been addressing stack slots through an abstract frame index and an immediate offset. The purpose of this function is to translate such a reference into a register—offset pair. Depending on whether the machine function that contains the instruction has a fixed or a variable stack frame, either the stack pointer %a10 or the frame pointer %a14 is used as the base register. The offset is computed accordingly. Fig. 3.10 demonstrates for both cases how a stack slot is addressed.

If the addressing mode of the affected instruction cannot handle the address because the offset is too large (the offset field has 10 bits for the BO addressing mode and 16 bits for the BOL mode), a sequence of instructions is emitted that explicitly computes the effective address. Interim results are put into an unused address register. If none is available, an already occupied address register is scavenged. For this purpose, LLVM's framework offers a class named RegScavenger that takes care of all the details.

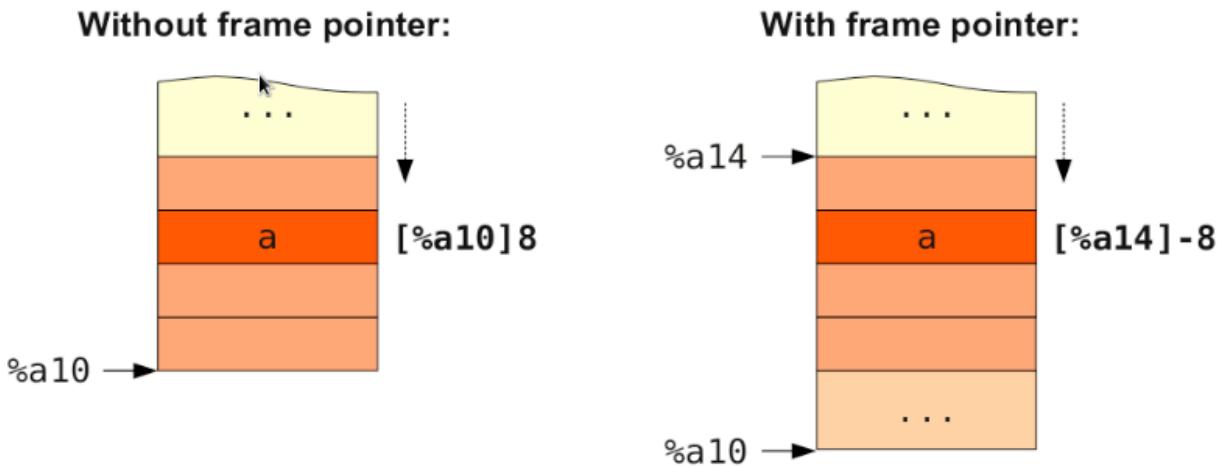


Fig. 3.10: Addressing of a variable *a* located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

3.5.2 Prologue and Epilogue functions

The Prologue and Epilogue functions as follows,

Ibdex/chapters/Chapter3_5/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
    MachineFrameInfo &MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    const Cpu0SEInstrInfo &TII =
        *static_cast<const Cpu0SEInstrInfo*>(STI.getInstrInfo());
    const Cpu0RegisterInfo &RegInfo =
        *static_cast<const Cpu0RegisterInfo *>(STI.getRegisterInfo());

    MachineBasicBlock::iterator MBBI = MBB.begin();
    DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() : DebugLoc();
    Cpu0ABIInfo ABI = STI.getABI();
    unsigned SP = Cpu0::SP;
    const TargetRegisterClass *RC = &Cpu0::GPROutRegClass;

    // First, compute final stack size.
    uint64_t StackSize = MFI.getStackSize();

    // No need to allocate space on the stack.
    if (StackSize == 0 && !MFI.adjustsStack()) return;

    MachineModuleInfo &MMI = MF.getMMI();
    const MCRegisterInfo *MRI = MMI.getContext().getRegisterInfo();

    // Adjust stack.
    TII.adjustStackPtr(SP, -StackSize, MBB, MBBI);

    // emit ".cfi_def_cfa_offset StackSize"
    unsigned CFIIndex =
        MF.addFrameInst(
            MCCFIIInstruction::cfiDefCfaOffset(nullptr, StackSize));
    BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
        .addCFIIndex(CFIIndex);

    const std::vector<CalleeSavedInfo> &CSI = MFI.getCalleeSavedInfo();

    if (!CSI.empty()) {
        // Find the instruction past the last instruction that saves a callee-saved
        // register to the stack.
        for (unsigned i = 0; i < CSI.size(); ++i)
            ++MBBI;

        // Iterate over list of callee-saved registers and emit .cfi_offset
        // directives.
        for (std::vector<CalleeSavedInfo>::const_iterator I = CSI.begin(),
            E = CSI.end(); I != E; ++I) {
            int64_t Offset = MFI.getObjectOffset(I->getFrameIdx());
            unsigned Reg = I->getReg();
            {
                // Reg is in CPURegs.
```

(continues on next page)

(continued from previous page)

```
        unsigned CFIIndex = MF.addFrameInst(MCCFIInstruction::createOffset(
            nullptr, MRI->getDwarfRegNum(Reg, true), Offset));
        BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
            .addCFIIndex(CFIIndex);
    }
}
}
```

```

void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
    MachineBasicBlock::iterator MBBI = MBB.getFirstTerminator();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    const Cpu0SEInstrInfo &TII =
        *static_cast<const Cpu0SEInstrInfo *>(STI.getInstrInfo());
    const Cpu0RegisterInfo &RegInfo =
        *static_cast<const Cpu0RegisterInfo *>(STI.getRegisterInfo());

    DebugLoc DL = MBBI != MBB.end() ? MBBI->getDebugLoc() : DebugLoc();
    Cpu0ABIInfo ABI = STI.getABI();
    unsigned SP = Cpu0::SP;

    // Get the number of bytes from FrameInfo
    uint64_t StackSize = MFI.getStackSize();

    if (!StackSize)
        return;

    // Adjust stack.
    TII.adjustStackPtr(SP, StackSize, MBB, MBBI);
}

```

```
bool
Cpu0SEFrameLowering::hasReservedCallFrame(const MachineFunction &MF) const {
    const MachineFrameInfo &MFI = MF.getFrameInfo();

    // Reserve call frame if the size of the maximum call frame fits into 16-bit
    // immediate field and there are no variable sized objects on the stack.
    // Make sure the second register scavenger spill slot can be accessed with one
    // instruction.
    return isInt<16>(MFI.getMaxCallFrameSize() + getStackAlignment()) &&
        !MFI.hasVarSizedObjects();
}
```

Ibdex/chapters/Chapter3_5/Cpu0MachineFunction.h

```

unsigned getIncomingArgSize() const { return IncomingArgSize; }

bool callsEhReturn() const { return CallsEhReturn; }
void setCallsEhReturn() { CallsEhReturn = true; }

bool callsEhDwarf() const { return CallsEhDwarf; }
void setCallsEhDwarf() { CallsEhDwarf = true; }

void createEhDataRegsFI();
int getEhDataRegFI(unsigned Reg) const { return EhDataRegFI[Reg]; }

unsigned getMaxCallFrameSize() const { return MaxCallFrameSize; }
void setMaxCallFrameSize(unsigned S) { MaxCallFrameSize = S; }

```

Now, we further explain the **Prologue** and **Epilogue** with an example.

For the following LLVM IR code of *ch3.cpp*, **Chapter3_5** of the **Cpu0** backend will emit the corresponding machine instructions as follows:

```

118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-230:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch3.bc -o -
...
define i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    ret i32 0
}

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
...
.section .mdebug.abi32
.previous
.file "ch3.bc"
.text
.globl main//static void expandLargeImm\n
.align 2
.type main,@function
.ent main                # @main
main:
.cfi_startproc
.frame $sp,8,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -8
$tmp1:
.cfi_def_cfa_offset 8
addiu $2, $zero, 0

```

(continues on next page)

(continued from previous page)

```

st    $2, 4($sp)
addiu $sp, $sp, 8
ret $lr
.set  macro
.set  reorder
.end  main
$tmp2:
.size main, ($tmp2)-main
.cfi_endproc

```

LLVM get the stack size by counting how many virtual registers is assigned to local variables. After that, it calls emit-Prologue().

```

virtual void adjustStackPtr(unsigned SP, int64_t Amount,
                           MachineBasicBlock &MBB,
                           MachineBasicBlock::iterator I) const = 0;

```

Ibdex/chapters/Chapter3_5/Cpu0SEInstrInfo.h

```

/// Adjust SP by Amount bytes.
void adjustStackPtr(unsigned SP, int64_t Amount, MachineBasicBlock &MBB,
                     MachineBasicBlock::iterator I) const override;

/// Emit a series of instructions to load an immediate. If NewImm is a
/// non-NULL parameter, the last instruction is not emitted, but instead
/// its immediate operand is returned in NewImm.
unsigned loadImmediate(int64_t Imm, MachineBasicBlock &MBB,
                      MachineBasicBlock::iterator II, const DebugLoc &DL,
                      unsigned *NewImm) const;

```

Ibdex/chapters/Chapter3_5/Cpu0SEInstrInfo.cpp

```

/// Adjust SP by Amount bytes.
void Cpu0SEInstrInfo::adjustStackPtr(unsigned SP, int64_t Amount,
                                      MachineBasicBlock &MBB,
                                      MachineBasicBlock::iterator I) const {
    DebugLoc DL = I != MBB.end() ? I->getDebugLoc() : DebugLoc();
    unsigned ADDu = Cpu0::ADDu;
    unsigned ADDiu = Cpu0::ADDiu;

    if (isInt<16>(Amount)) {
        // addiu sp, sp, amount
        BuildMI(MBB, I, DL, get(ADDiu), SP).addReg(SP).addImm(Amount);
    } else { // Expand immediate that doesn't fit in 16-bit.
        unsigned Reg = loadImmediate(Amount, MBB, I, DL, nullptr);
        BuildMI(MBB, I, DL, get(ADDu), SP).addReg(SP).addReg(Reg, RegState::Kill);
    }
}

/// This function generates the sequence of instructions needed to get the

```

(continues on next page)

(continued from previous page)

```

/// result of adding register REG and immediate IMM.
unsigned
Cpu0SEInstrInfo::loadImmediate(int64_t Imm, MachineBasicBlock &MBB,
                                MachineBasicBlock::iterator II,
                                const DebugLoc &DL,
                                unsigned *NewImm) const {
    Cpu0AnalyzeImmediate AnalyzeImm;
    unsigned Size = 32;
    unsigned LUi = Cpu0::LUi;
    unsigned ZEROReg = Cpu0::ZERO;
    unsigned ATReg = Cpu0::AT;
    bool LastInstrIsADDiu = NewImm;

    const Cpu0AnalyzeImmediate::InstSeq &Seq =
        AnalyzeImm.Analyze(Imm, Size, LastInstrIsADDiu);
    Cpu0AnalyzeImmediate::InstSeq::const_iterator Inst = Seq.begin();

    assert(Seq.size() && (!LastInstrIsADDiu || (Seq.size() > 1)));

    // The first instruction can be a LUi, which is different from other
    // instructions (ADDiu, ORI and SLL) in that it does not have a register
    // operand.
    if (Inst->Opc == LUi)
        BuildMI(MBB, II, DL, get(LUi), ATReg).addImm(SignExtend64<16>(Inst->ImmOpnd));
    else
        BuildMI(MBB, II, DL, get(Inst->Opc), ATReg).addReg(ZEROReg)
            .addImm(SignExtend64<16>(Inst->ImmOpnd));

    // Build the remaining instructions in Seq.
    for (++Inst; Inst != Seq.end() - LastInstrIsADDiu; ++Inst)
        BuildMI(MBB, II, DL, get(Inst->Opc), ATReg).addReg(ATReg)
            .addImm(SignExtend64<16>(Inst->ImmOpnd));

    if (LastInstrIsADDiu)
        *NewImm = Inst->ImmOpnd;

    return ATReg;
}

```

In `emitPrologue()`, it emits machine instructions to adjust the **sp** (stack pointer register) for local variables. For our example, it will emit the instruction:

```
addiu $sp, $sp, -8
```

In the above `ch3.cpp` assembly output, it generates:

```
addiu $2, $zero, 0
```

rather than:

```
ori $2, $zero, 0
```

because **ADDiu** is defined before **ORi** as shown below, so it takes priority. Of course, if **ORi** were defined first, it would

translate into the **ori** instruction.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
      (ADDiu ZERO, imm:$in)>;
```

Ibdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
let Predicates = [Ch3_5] in {
def : Pat<(i32 immZExt16:$in),
      (ORi ZERO, imm:$in)>;
def : Pat<(i32 immLow16Zero:$in),
      (LUI (HI16 imm:$in))>;

// Arbitrary immediates
def : Pat<(i32 imm:$imm),
      (ORi (LUI (HI16 imm:$imm)), (LO16 imm:$imm))>;
} // let Predicates = [Ch3_4]
```

3.5.3 Handle stack slot for local variables

The following code handle the stack slot for local variables.

Ibdex/chapters/Chapter3_5/Cpu0RegisterInfo.cpp

```
/*- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                     unsigned FIOperandNum, RegScavenger *RS) const {
    MachineInstr &MI = *II;
    MachineFunction &MF = *MI.getParent()->getParent();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    unsigned i = 0;
    while (!MI.getOperand(i).isFI()) {
        ++i;
        assert(i < MI.getNumOperands() &&
               "Instr doesn't have FrameIndex operand!");
    }

    LLVM_DEBUG(errs() << "\nFunction : " << MF.getFunction().getName() << "\n";
               errs() << "-----\n" << MI);

    int FrameIndex = MI.getOperand(i).getIndex();
    uint64_t stackSize = MF.getFrameInfo().getStackSize();
```

(continues on next page)

(continued from previous page)

```
int64_t spOffset = MF.getFrameInfo().getObjectOffset(FrameIndex);

LLVM_DEBUG(errs() << "FrameIndex : " << FrameIndex << "\n"
           << "spOffset : " << spOffset << "\n"
           << "stackSize : " << stackSize << "\n");

const std::vector<CalleeSavedInfo> &CSI = MFI.getCalleeSavedInfo();
int MinCSFI = 0;
int MaxCSFI = -1;

if (CSI.size()) {
    MinCSFI = CSI[0].getFrameIdx();
    MaxCSFI = CSI[CSI.size() - 1].getFrameIdx();
}

// The following stack frame objects are always referenced relative to $sp:
// 1. Outgoing arguments.
// 2. Pointer to dynamically allocated stack space.
// 3. Locations for callee-saved registers.
// Everything else is referenced relative to whatever register
// getFrameRegister() returns.
unsigned FrameReg;

FrameReg = Cpu0::SP;

// Calculate final offset.
// - There is no need to change the offset if the frame object is one of the
//   following: an outgoing argument, pointer to a dynamically allocated
//   stack space or a $gp restore location,
// - If the frame object is any of the following, its offset must be adjusted
//   by adding the size of the stack:
//   incoming argument, callee-saved register location or local variable.
int64_t Offset;
Offset = spOffset + (int64_t)stackSize;

Offset += MI.getOperand(i+1).getImm();

LLVM_DEBUG(errs() << "Offset : " << Offset << "\n" << "<----->\n");

// If MI is not a debug value, make sure Offset fits in the 16-bit immediate
// field.
if (!MI.isDebugValue() && !isInt<16>(Offset)) {
    errs() << "!!!!ERROR!!! Not support large frame over 16-bit at this point.\n"
        << "Though CH3_5 support it."
        << "Reference: "
        "http://jonathan2251.github.io/lbd/backendstructure.html#large-stack\n"
        << "However the CH9_3, dynamic-stack-allocation-support bring instruction "
        "move $fp, $sp that make it complicated in coding against the tutorial "
        "purpose of Cpu0.\n"
        << "Reference: "
        "http://jonathan2251.github.io/lbd/funccall.html#dynamic-stack-"
        "allocation-support\n";
}
```

(continues on next page)

(continued from previous page)

```

    assert(0 && "(!MI.isDebugValue() && !isInt<16>(Offset))");
}

MI.getOperand(i).ChangeToRegister(FrameReg, false);
MI.getOperand(i+1).ChangeToImmediate(Offset);
}

```

The `eliminateFrameIndex()` function in `Cpu0RegisterInfo.cpp` is called after the stages of **instruction selection** and **register allocation**. It translates the frame index to the correct offset of the stack pointer using:

```
spOffset = MF.getFrameInfo()->getObjectOffset(FrameIndex);
```

For instance, in `ch3.cpp`, the offset calculation is displayed as follows:

```

Spilling live registers at end of block.
BB#0: derived from LLVM BB %0
      %V0<def> = ADDiu %ZERO, 0
      ST %V0, <fi#0>, 0; mem:ST4[%1]
      RetLR %V0<imp-use,kill>
alloc FI(0) at SP[-4]

Function : main
<----->
ST %V0, <fi#0>, 0; mem:ST4[%1]
FrameIndex : 0
spOffset    : -4
stackSize   : 8
Offset      : 4
<----->
...
.file "ch3.bc"
...
.frame $sp,8,$lr
...
# BB#0:
addiu $sp, $sp, -8
$tmp1:
.cfi_def_cfa_offset 8
addiu $2, $zero, 0
st $2, 4($sp)
...

```

[Index/chapters/Chapter3_5/Cpu0SEFrameLowering.cpp](#)

```

// This method is called immediately before PrologEpilogInserter scans the
// physical registers used to determine what callee saved registers should be
// spilled. This method is optional.
void Cpu0SEFrameLowering::determineCalleeSaves(MachineFunction &MF,
                                                BitVector &SavedRegs,
                                                RegScavenger *RS) const {
// @determineCalleeSaves-body
    TargetFrameLowering::determineCalleeSaves(MF, SavedRegs, RS);
}

```

(continues on next page)

(continued from previous page)

```
Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

if (MF.getFrameInfo().hasCalls())
    setAliasRegs(MF, SavedRegs, Cpu0::LR);

return;
}
```

The `determineCalleeSaves()` function in `Cpu0SEFrameLowering.cpp` determines the spill registers. Once the spill registers are identified, the function `SpillCalleeSavedRegisters()` will save/restore registers to/from stack slots via the following code:

[Index/chapters/Chapter3_5/Cpu0InstrInfo.h](#)

```
void storeRegToStackSlot(MachineBasicBlock &MBB,
                         MachineBasicBlock::iterator MBBI,
                         Register SrcReg, bool isKill, int FrameIndex,
                         const TargetRegisterClass *RC,
                         const TargetRegisterInfo *TRI) const override {
    storeRegToStack(MBB, MBBI, SrcReg, isKill, FrameIndex, RC, TRI, 0);
}

void loadRegFromStackSlot(MachineBasicBlock &MBB,
                          MachineBasicBlock::iterator MBBI,
                          Register DestReg, int FrameIndex,
                          const TargetRegisterClass *RC,
                          const TargetRegisterInfo *TRI) const override {
    loadRegFromStack(MBB, MBBI, DestReg, FrameIndex, RC, TRI, 0);
}

virtual void storeRegToStack(MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator MI,
                               Register SrcReg, bool isKill, int FrameIndex,
                               const TargetRegisterClass *RC,
                               const TargetRegisterInfo *TRI,
                               int64_t Offset) const = 0;

virtual void loadRegFromStack(MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator MI,
                               Register DestReg, int FrameIndex,
                               const TargetRegisterClass *RC,
                               const TargetRegisterInfo *TRI,
                               int64_t Offset) const = 0;
```

```
MachineMemOperand *GetMemOperand(MachineBasicBlock &MBB, int FI,
                                 MachineMemOperand::Flags Flags) const;
```

[Index/chapters/Chapter3_5/Cpu0InstrInfo.cpp](#)

```
MachineMemOperand *
Cpu0InstrInfo::GetMemOperand(MachineBasicBlock &MBB, int FI,
                            MachineMemOperand::Flags Flags) const {
```

(continues on next page)

(continued from previous page)

```
MachineFunction &MF = *MBB.getParent();
MachineFrameInfo &MFI = MF.getFrameInfo();

return MF.getMachineMemOperand(MachinePointerInfo::getFixedStack(MF, FI),
                                Flags, MFI.getObjectSize(FI),
                                MFI.getObjectAlign(FI));
}
```

Ibdex/chapters/Chapter3_5/Cpu0SEInstrInfo.h

```
void storeRegToStack(MachineBasicBlock &MBB,
                     MachineBasicBlock::iterator MI,
                     Register SrcReg, bool isKill, int FrameIndex,
                     const TargetRegisterClass *RC,
                     const TargetRegisterInfo *TRI,
                     int64_t Offset) const override;

void loadRegFromStack(MachineBasicBlock &MBB,
                      MachineBasicBlock::iterator MI,
                      Register DestReg, int FrameIndex,
                      const TargetRegisterClass *RC,
                      const TargetRegisterInfo *TRI,
                      int64_t Offset) const override;
```

Ibdex/chapters/Chapter3_5/Cpu0SEInstrInfo.cpp

```
void Cpu0SEInstrInfo::
storeRegToStack(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                Register SrcReg, bool isKill, int FI,
                const TargetRegisterClass *RC, const TargetRegisterInfo *TRI,
                int64_t Offset) const {
    DebugLoc DL;
    MachineMemOperand *MMO = GetMemOperand(MBB, FI, MachineMemOperand::MOStore);

    unsigned Opc = 0;

    Opc = Cpu0::ST;
    assert(Opc && "Register class not handled!");
    BuildMI(MBB, I, DL, get(Opc)).addReg(SrcReg, getKillRegState(isKill))
        .addFrameIndex(FI).addImm(Offset).addMemOperand(MMO);
}

void Cpu0SEInstrInfo::
loadRegFromStack(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                 Register DestReg, int FI, const TargetRegisterClass *RC,
                 const TargetRegisterInfo *TRI, int64_t Offset) const {
    DebugLoc DL;
    if (I != MBB.end()) DL = I->getDebugLoc();
    MachineMemOperand *MMO = GetMemOperand(MBB, FI, MachineMemOperand::MOLoad);
    unsigned Opc = 0;
```

(continues on next page)

(continued from previous page)

```
Opc = Cpu0::LD;
assert (Opc && "Register class not handled!");
BuildMI(MBB, I, DL, get(Opc), DestReg).addFrameIndex(FI).addImm(Offset)
    .addMemOperand(MMO);
}
```

Functions `storeRegToStack()` in `Cpu0SEInstrInfo.cpp` and `storeRegToStackSlot()` in `Cpu0InstrInfo.cpp` handle register spilling during the register allocation process.

Each local variable is associated with a frame index. The code `.addFrameIndex(FI).addImm(Offset).addMemOperand(MMO);` in `storeRegToStack()` (where `Offset` is 0) is added for each virtual register.

The functions `loadRegFromStackSlot()` and `loadRegFromStack()` are used when registers need to be reloaded from stack slots.

If `V0` is added to `Cpu0CallingConv.td` as shown below and both `storeRegToStack()` and `storeRegToStackSlot()` are absent in `Cpu0SEInstrInfo.cpp`, `Cpu0SEInstrInfo.h`, and `Cpu0InstrInfo.h`, the following error will occur.

Ibdex/Cpu0/Cpu0CallingConv.td

```
def CSR_O32 : CalleeSavedRegs<(add LR, FP, V0,
                                (sequence "S%u", 1, 0))>;
```

```
114-43-191-19:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
.text
.section .mdebug.abi032
.previous
.file "ch3.bc"
Target didn't implement TargetInstrInfo::storeRegToStackSlot!
...
Stack dump:
...
Abort trap: 6
```

Table 3.3: Backend functions called in PrologEpilogInserter.cpp

Stage	Function
Prologue/Epilogue Insertion & Frame Finalization	
• Determine spill callee saved registers	• Cpu0SEFrameLowering::determineCalleeSaves
• Spill callee saved registers	• Cpu0SEFrameLowering::spillCalleeSavedRegisters
• Prolog	• Cpu0SEFrameLowering::emitPrologue
• Epilog	• Cpu0SEFrameLowering::emitEpilogue
• Handle stack slot for local variables	• Cpu0RegisterInfo::eliminateFrameIndex

File `PrologEpilogInserter.cpp` will call backend functions `spillCalleeSavedRegisters()`, `emitProlog()`, `emitEpilog()` and `eliminateFrameIndex()` as follows,

lib/CodeGen/PrologEpilogInserter.cpp

```

class PEI : public MachineFunctionPass {
public:
    static char ID;
    explicit PEI() : MachineFunctionPass(ID) {
        initializePEIPass(*PassRegistry::getPassRegistry());
    }

    if (TM && (!TM->usesPhysRegsForPEI())) {
        ...
    } else {
        SpillCalleeSavedRegisters = doSpillCalleeSavedRegs;
        ...
    }
}
...
}

/// insertCSRSpillsAndRestores - Insert spill and restore code for
/// callee saved registers used in the function.
///
static void insertCSRSpillsAndRestores(MachineFunction &Fn,
                                         const MBBVector &SaveBlocks,
                                         const MBBVector &RestoreBlocks) {
    ...
    // Spill using target interface.
    for (MachineBasicBlock *SaveBlock : SaveBlocks) {
        ...
        if (!TFI->spillCalleeSavedRegisters(*SaveBlock, I, CSI, TRI)) {
            for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
                // Insert the spill to the stack frame.
                ...
                TII.storeRegToStackSlot(*SaveBlock, I, Reg, true, CSI[i].getFrameIdx(),
                                       RC, TRI);
            }
        }
        ...
    }

    // Restore using target interface.
    for (MachineBasicBlock *MBB : RestoreBlocks) {
        ...
        // Restore all registers immediately before the return and any
        // terminators that precede it.
        if (!TFI->restoreCalleeSavedRegisters(*MBB, I, CSI, TRI)) {
            for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
                ...
                TII.loadRegFromStackSlot(*MBB, I, Reg, CSI[i].getFrameIdx(), RC, TRI);
                ...
            }
        }
        ...
    }
}
...
}

```

(continues on next page)

(continued from previous page)

```

}

static void doSpillCalleeSavedRegs(MachineFunction &Fn, RegScavenger *RS,
                                    unsigned &MinCSFrameIndex,
                                    unsigned &MaxCSFrameIndex,
                                    const MBBVector &SaveBlocks,
                                    const MBBVector &RestoreBlocks) {
    const Function *F = Fn.getFunction();
    const TargetFrameLowering *TFI = Fn.getSubtarget().getFrameLowering();
    MinCSFrameIndex = std::numeric_limits<unsigned>::max();
    MaxCSFrameIndex = 0;

    // Determine which of the registers in the callee save list should be saved.
    BitVector SavedRegs;
    TFI->determineCalleeSaves(Fn, SavedRegs, RS);

    // Assign stack slots for any callee-saved registers that must be spilled.
    assignCalleeSavedSpillSlots(Fn, SavedRegs, MinCSFrameIndex, MaxCSFrameIndex);

    // Add the code to save and restore the callee saved registers.
    if (!F->hasFnAttribute(Attribute::Naked))
        insertCSRSpillsAndRestores(Fn, SaveBlocks, RestoreBlocks);
}

void PEI::insertPrologEpilogCode(MachineFunction &Fn) {
    const TargetFrameLowering &TFI = *Fn.getSubtarget().getFrameLowering();

    // Add prologue to the function...
    for (MachineBasicBlock *SaveBlock : SaveBlocks)
        TFI.emitPrologue(Fn, *SaveBlock);

    // Add epilogue to restore the callee-save registers in each exiting block.
    for (MachineBasicBlock *RestoreBlock : RestoreBlocks)
        TFI.emitEpilogue(Fn, *RestoreBlock);
    ...
}

void PEI::replaceFrameIndices(MachineBasicBlock *BB, MachineFunction &Fn,
                               int &SPAdj) {
    ...
    for (unsigned i = 0, e = MI.getNumOperands(); i != e; ++i) {
        ...
        // If this instruction has a FrameIndex operand, we need to
        // use that target machine register info object to eliminate
        // it.
        TRI.eliminateFrameIndex(MI, SPAdj, i,
                               FrameIndexVirtualScavenging ? nullptr : RS);
        ...
    }
    ...
}

```

(continues on next page)

(continued from previous page)

```

/// replaceFrameIndices - Replace all MO_FrameIndex operands with physical
/// register references and actual offsets.
///
void PEI::replaceFrameIndices(MachineFunction &Fn) {
    ...
    // Iterate over the reachable blocks in DFS order.
    for (auto DFI = df_ext_begin(&Fn, Reachable), DFE = df_ext_end(&Fn, Reachable);
        DFI != DFE; ++DFI) {
        ...
        replaceFrameIndices(BB, Fn, SPAdj);
        ...
    }

    // Handle the unreachable blocks.
    for (auto &BB : Fn) {
        ...
        replaceFrameIndices(&BB, Fn, SPAdj);
    }
}

bool PEI::runOnMachineFunction(MachineFunction &Fn) {
    const TargetFrameLowering &TFI = *Fn.getSubtarget().getFrameLowering();
    ...
    FrameIndexVirtualScavenging = TRI->requiresFrameIndexScavenging(Fn);
    ...
    // Handle CSR spilling and restoring, for targets that need it.
    SpillCalleeSavedRegisters(Fn, RS, MinCSFrameIndex, MaxCSFrameIndex,
                               SaveBlocks, RestoreBlocks);
    ...
    // Calculate actual frame offsets for all abstract stack objects...
    calculateFrameObjectOffsets(Fn);

    // Add prolog and epilog code to the function. This function is required
    // to align the stack frame as necessary for any stack variables or
    // called functions. Because of this, calculateCalleeSavedRegisters()
    // must be called before this function in order to set the AdjustsStack
    // and MaxCallFrameSize variables.
    if (!F->hasFnAttribute(Attribute::Naked))
        insertPrologEpilogCode(Fn);

    // Replace all MO_FrameIndex operands with physical register references
    // and actual offsets.
    //
    replaceFrameIndices(Fn);

    // If register scavenging is needed, as we've enabled doing it as a
    // post-pass, scavenge the virtual registers that frame index elimination
    // inserted.
    if (TRI->requiresRegisterScavenging(Fn) && FrameIndexVirtualScavenging) {
        ScavengeFrameVirtualRegs(Fn, RS);

        // Clear any vregs created by virtual scavenging.
    }
}

```

(continues on next page)

(continued from previous page)

```
Fn.getRegInfo().clearVirtRegs();  
}  
...  
}
```

3.5.4 Large stack

At this stage, we have successfully translated a simple `main()` function containing only `return 0;`. However, the stack size adjustments for 32-bit values are handled by `Cpu0AnalyzeImmediate.cpp` and the instruction definitions in `Cpu0InstrInfo.td` from Chapter3_5.

Later, in **CH9_3**, dynamic stack allocation support introduces the instruction:

```
move $fp, $sp
```

This addition makes the implementation more complex, potentially diverging from the tutorial's primary focus on the **Cpu0** architecture.

Thus, for simplicity, we avoid handling large stack frames at this stage.⁴

Ibdex/chapters/Chapter3_5/CMakeLists.txt

```
add_llvm_target(  
    ...  
  
    Cpu0AnalyzeImmediate.cpp  
  
    ...  
)
```

Ibdex/chapters/Chapter3_5/Cpu0AnalyzeImmediate.h

```
===== Cpu0AnalyzeImmediate.h - Analyze Immediates -----*-- C++ -*-----//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----//  
#ifndef CPU0_ANALYZE_IMMEDIATE_H  
#define CPU0_ANALYZE_IMMEDIATE_H  
  
#include "Cpu0Config.h"  
#if CH >= CH3_5  
  
#include "llvm/ADT/SmallVector.h"  
#include "llvm/Support/DataTypes.h"  
  
namespace llvm {
```

(continues on next page)

⁴ <http://jonathan2251.github.io/lbd/funccall.html#dynamic-stack-allocation-support>

(continued from previous page)

```

class Cpu0AnalyzeImmediate {
public:
    struct Inst {
        unsigned Opc, ImmOpnd;
        Inst(unsigned Opc, unsigned ImmOpnd);
    };
    typedef SmallVector<Inst, 7> InstSeq;

    /// Analyze - Get an instruction sequence to load immediate Imm. The last
    /// instruction in the sequence must be an ADDiu if LastInstrIsADDiu is
    /// true;
    const InstSeq &Analyze(uint64_t Imm, unsigned Size, bool LastInstrIsADDiu);
private:
    typedef SmallVector<InstSeq, 5> InstSeqLs;

    /// AddInstr - Add I to all instruction sequences in SeqLs.
    void AddInstr(InstSeqLs &SeqLs, const Inst &I);

    /// GetInstSeqLsADDiu - Get instruction sequences which end with an ADDiu to
    /// load immediate Imm
    void GetInstSeqLsADDiu(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// GetInstSeqLsORi - Get instruction sequences which end with an ORi to
    /// load immediate Imm
    void GetInstSeqLsORi(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// GetInstSeqLsSHL - Get instruction sequences which end with a SHL to
    /// load immediate Imm
    void GetInstSeqLsSHL(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// GetInstSeqLs - Get instruction sequences to load immediate Imm.
    void GetInstSeqLs(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// ReplaceADDiuSHLWithLUi - Replace an ADDiu & SHL pair with a LUi.
    void ReplaceADDiuSHLWithLUi(InstSeq &Seq);

    /// GetShortestSeq - Find the shortest instruction sequence in SeqLs and
    /// return it in Insts.
    void GetShortestSeq(InstSeqLs &SeqLs, InstSeq &Insts);

    unsigned Size;
    unsigned ADDiu, ORi, SHL, LUi;
    InstSeq Insts;
};

}

#endif // #if CH >= CH3_5

#endif

```

Ibdex/chapters/Chapter3_5/Cpu0AnalyzeImmediate.cpp

```
//===== Cpu0AnalyzeImmediate.cpp - Analyze Immediates =====//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

#include "Cpu0AnalyzeImmediate.h"
#include "Cpu0.h"
#if CH >= CH3_5

#include "llvm/Support/MathExtras.h"

using namespace llvm;

Cpu0AnalyzeImmediate::Inst::Inst(unsigned O, unsigned I) : Opc(O), ImmOpnd(I) {}

// Add I to the instruction sequences.
void Cpu0AnalyzeImmediate::AddInstr(InstSeqLs &SeqLs, const Inst &I) {
    // Add an instruction sequence consisting of just I.
    if (SeqLs.empty()) {
        SeqLs.push_back(InstSeq(1, I));
        return;
    }

    for (InstSeqLs::iterator Iter = SeqLs.begin(); Iter != SeqLs.end(); ++Iter)
        Iter->push_back(I);
}

void Cpu0AnalyzeImmediate::GetInstSeqLsADDiu(uint64_t Imm, unsigned RemSize,
                                              InstSeqLs &SeqLs) {
    GetInstSeqLs((Imm + 0x8000ULL) & 0xffffffffffff0000ULL, RemSize, SeqLs);
    AddInstr(SeqLs, Inst(ADDiu, Imm & 0xffffFULL));
}

void Cpu0AnalyzeImmediate::GetInstSeqLsORi(uint64_t Imm, unsigned RemSize,
                                             InstSeqLs &SeqLs) {
    GetInstSeqLs(Imm & 0xffffffffffff0000ULL, RemSize, SeqLs);
    AddInstr(SeqLs, Inst(ORi, Imm & 0xffffFULL));
}

void Cpu0AnalyzeImmediate::GetInstSeqLsSHL(uint64_t Imm, unsigned RemSize,
                                             InstSeqLs &SeqLs) {
    unsigned Shamt = countTrailingZeros(Imm);
    GetInstSeqLs(Imm >> Shamt, RemSize - Shamt, SeqLs);
    AddInstr(SeqLs, Inst(SHL, Shamt));
}

void Cpu0AnalyzeImmediate::GetInstSeqLs(uint64_t Imm, unsigned RemSize,
                                         InstSeqLs &SeqLs) {
```

(continues on next page)

(continued from previous page)

```

uint64_t MaskedImm = Imm & (0xffffffffffffffffULL >> (64 - Size));

// Do nothing if Imm is 0.
if (!MaskedImm)
    return;

// A single ADDiu will do if RemSize <= 16.
if (RemSize <= 16) {
    AddInstr(SeqLs, Inst(ADDiu, MaskedImm));
    return;
}

// Shift if the lower 16-bit is cleared.
if (!(Imm & 0xffff)) {
    GetInstSeqLsSHL(Imm, RemSize, SeqLs);
    return;
}

GetInstSeqLsADDiu(Imm, RemSize, SeqLs);

// If bit 15 is cleared, it doesn't make a difference whether the last
// instruction is an ADDiu or ORi. In that case, do not call GetInstSeqLsORi.
if (Imm & 0x8000) {
    InstSeqLs SeqLsORi;
    GetInstSeqLsORi(Imm, RemSize, SeqLsORi);
    SeqLs.insert(SeqLs.end(), SeqLsORi.begin(), SeqLsORi.end());
}
}

// Replace a ADDiu & SHL pair with a LUi.
// e.g. the following two instructions
// ADDiu 0x0111
// SHL 18
// are replaced with
// LUi 0x444
void Cpu0AnalyzeImmediate::ReplaceADDiuSHLWithLUI(InstSeq &Seq) {
    // Check if the first two instructions are ADDiu and SHL and the shift amount
    // is at least 16.
    if ((Seq.size() < 2) || (Seq[0].Opc != ADDiu) ||
        (Seq[1].Opc != SHL) || (Seq[1].ImmOpnd < 16))
        return;

    // Sign-extend and shift operand of ADDiu and see if it still fits in 16-bit.
    int64_t Imm = SignExtend64<16>(Seq[0].ImmOpnd);
    int64_t ShiftedImm = (uint64_t)Imm << (Seq[1].ImmOpnd - 16);

    if (!isInt<16>(ShiftedImm))
        return;

    // Replace the first instruction and erase the second.
    Seq[0].Opc = LUi;
    Seq[0].ImmOpnd = (unsigned)(ShiftedImm & 0xffff);
}

```

(continues on next page)

(continued from previous page)

```
Seq.erase(Seq.begin() + 1);
}

void Cpu0AnalyzeImmediate::GetShortestSeq(InstSeqLs &SeqLs, InstSeq &Insts) {
    InstSeqLs::iterator ShortestSeq = SeqLs.end();
    // The length of an instruction sequence is at most 7.
    unsigned ShortestLength = 8;

    for (InstSeqLs::iterator S = SeqLs.begin(); S != SeqLs.end(); ++S) {
        ReplaceADDiuSHLWithLUI(*S);
        assert(S->size() <= 7);

        if (S->size() < ShortestLength) {
            ShortestSeq = S;
            ShortestLength = S->size();
        }
    }

    Insts.clear();
    Insts.append(ShortestSeq->begin(), ShortestSeq->end());
}

const Cpu0AnalyzeImmediate::InstSeq
&Cpu0AnalyzeImmediate::Analyze(uint64_t Imm, unsigned Size,
                                bool LastInstrIsADDiu) {
    this->Size = Size;

    ADDiu = Cpu0::ADDiu;
    ORi = Cpu0::ORi;
    SHL = Cpu0::SHL;
    LUI = Cpu0::LUI;

    InstSeqLs SeqLs;

    // Get the list of instruction sequences.
    if (LastInstrIsADDiu | !Imm)
        GetInstSeqLsADDiu(Imm, Size, SeqLs);
    else
        GetInstSeqLs(Imm, Size, SeqLs);

    // Set Insts to the shortest instruction sequence.
    GetShortestSeq(SeqLs, Insts);

    return Insts;
}

#endif
```

Ibdex/chapters/Chapter3_5/Cpu0InstrInfo.h

```
#include "Cpu0AnalyzeImmediate.h"
```

Ibdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
class Cpu0InstAlias<string Asm, dag Result, bit Emit = 0b1> :
    InstAlias<Asm, Result, Emit>;
```

```
def shamt      : Operand<i32>;
```

```
// Unsigned Operand
```

```
def uimm16     : Operand<i32> {
    let PrintMethod = "printUnsignedImm";
}
```

```
// Transformation Function - get the lower 16 bits.
```

```
def LO16 : SDNodeXForm<imm, [<
    return getImm(N, N->getZExtValue() & 0xffff);
}]>;
```

```
// Transformation Function - get the higher 16 bits.
```

```
def HI16 : SDNodeXForm<imm, [<
    return getImm(N, (N->getZExtValue() >> 16) & 0xffff);
}]>;
```

```
// Node immediate fits as 16-bit zero extended on target immediate.
```

```
// The LO16 param means that only the lower 16 bits of the node
// immediate are caught.
```

```
// e.g. addiu, sltiu
```

```
def immZExt16 : PatLeaf<(imm), [<
    if (N->getValueType(0) == MVT::i32)
        return (uint32_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
    else
        return (uint64_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
}]>, LO16>;
```

```
// Immediate can be loaded with LUi (32-bit int with lower 16-bit cleared).
```

```
def immLow16Zero : PatLeaf<(imm), [<
    int64_t Val = N->getSExtValue();
    return isInt<32>(Val) && !(Val & 0xffff);
}]>;
```

```
// shamt field must fit in 5 bits.
```

```
def immZExt5 : ImmLeaf<i32, [<{return Imm == (Imm & 0x1f);}>]>;
```

```
let Predicates = [Ch3_5] in {
// Arithmetic and logical instructions with 3 register operands.
class ArithLogicR<bits<8> op, string instr_asm, SDNode OpNode,
    InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
    FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
    !strconcat(instr_asm, "\t$ra, $rb, $rc"),
```

(continues on next page)

(continued from previous page)

```
[ (set GPROut:$ra, (OpNode RC:$rb, RC:$rc))), itin> {
let shamt = 0;
let isCommutable = isComm;      // e.g. add rb rc = add rc rb
let isReMaterializable = 1;
}
```

```
let Predicates = [Ch3_5] in {
// Shifts
class shift_rotate_imm<bits<8> op, bits<4> isRotate, string instr_asm,
    SDNode OpNode, PatFrag PF, Operand ImmOpnd,
    RegisterClass RC>:
FA<op, (outs GPROut:$ra), (ins RC:$rb, ImmOpnd:$shamt),
    !strconcat(instr_asm, "\t$ra, $rb, $shamt"),
    [(set GPROut:$ra, (OpNode RC:$rb, PF:$shamt))], IIAlu> {
let rc = 0;
}

// 32-bit shift instructions.
class shift_rotate_imm32<bits<8> op, bits<4> isRotate, string instr_asm,
    SDNode OpNode>:
shift_rotate_imm<op, isRotate, instr_asm, OpNode, immZExt5, shamt, CPURegs>;
}
```

```
let Predicates = [Ch3_5] in {
// Load Upper Immediate
class LoadUpper<bits<8> op, string instr_asm, RegisterClass RC, Operand Imm>:
FL<op, (outs RC:$ra), (ins Imm:$imm16),
    !strconcat(instr_asm, "\t$ra, $imm16"), [], IIAlu> {
let rb = 0;
let isReMaterializable = 1;
}
```

```
let Predicates = [Ch3_5] in {
def ORI      : ArithLogicI<0x0d, "ori", or, uimm16, immZExt16, CPURegs>;
}
```

```
let Predicates = [Ch3_5] in {
def LUI      : LoadUpper<0x0F, "lui", GPROut, uimm16>;
}
```

```
let Predicates = [Ch3_5] in {
let Predicates = [DisableOverflow] in {
def ADDu     : ArithLogicR<0x11, "addu", add, IIAlu, CPURegs, 1>;
}}
```

```
let Predicates = [Ch3_5] in {
def SHL      : shift_rotate_imm32<0x1e, 0x00, "shl", shl>;
}
```

```

let Predicates = [Ch3_5] in {
//=====
// Instruction aliases
//=====
def : Cpu0InstAlias<"move $dst, $src",
           (ADDu GPROut:$dst, GPROut:$src, ZERO), 1>;
}

```

```

let Predicates = [Ch3_5] in {
def : Pat<(i32 immZExt16:$in),
      (ORi ZERO, imm:$in)>;
def : Pat<(i32 immLow16Zero:$in),
      (LUI (HI16 imm:$in))>

// Arbitrary immediates
def : Pat<(i32 imm:$imm),
      (ORi (LUI (HI16 imm:$imm)), (LO16 imm:$imm))>;
} // let Predicates = [Ch3_4]

```

The `Cpu0AnalyzeImmediate.cpp` implementation is recursive and has a somewhat complex logic. However, recursive techniques are commonly covered in compiler frontend books, so you should already be familiar with them.

Instead of tracing the code directly, we list the stack size and the corresponding instructions generated in:

Table: Cpu0 stack adjustment instructions before replacing ``addiu`` and ``shl`` with ``lui`` instructions

as follows, and in:

Table: Cpu0 stack adjustment instructions after replacing ``addiu`` and ``shl`` with ``lui`` instructions

in the next section.

Table 3.4: Cpu0 stack adjustment instructions before replace addiu and shl with lui instruction

stack size range	ex. stack size	Cpu0 Prologue instructions	Cpu0 Epilogue instructions
0 ~ 0x7ff8	• 0x7ff8	• addiu \$sp, \$sp, -32760;	• addiu \$sp, \$sp, 32760;
0x8000 ~ 0xffff8	• 0x8000	• addiu \$sp, \$sp, -32768;	• addiu \$1, \$zero, 1; • shl \$1, \$1, 16; • addiu \$1, \$1, -32768; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff8	• 0x7fffffff8	• addiu \$1, \$zero, 8; • shl \$1, \$1, 28; • addiu \$1, \$1, 8; • addu \$sp, \$sp, \$1;	• addiu \$1, \$zero, 8; • shl \$1, \$1, 28; • addiu \$1, \$1, -8; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff8	• 0x90008000	• addiu \$1, \$zero, -9; • shl \$1, \$1, 28; • addiu \$1, \$1, -32768; • addu \$sp, \$sp, \$1;	• addiu \$1, \$zero, -28671; • shl \$1, \$1, 16 • addiu \$1, \$1, -32768; • addu \$sp, \$sp, \$1;

Since the Cpu0 stack is 8-byte aligned, addresses from 0x7FF9 to 0x7FFF cannot exist.

Assume `sp = 0xA0008000` and `stack size = 0x90008000`, then `(0xA0008000 - 0x90008000) => 0x10000000`. Verify with the Cpu0 prologue instructions as follows,

1. “addiu \$1, \$zero, -9”=>(\$1 = 0 + 0xffffffff7) => \$1 = 0xffffffff7.

2. “shl \$1, \$1, 28;”=> \$1 = 0x70000000.
3. “addiu \$1, \$1, -32768”=> \$1 = (0x70000000 + 0xffff8000) => \$1 = 0x6fff8000.
4. “addu \$sp, \$sp, \$1”=> \$sp = (0xa0008000 + 0x6fff8000) => \$sp = 0x10000000.

Verify with the Cpu0 epilogue instructions with `sp = 0x10000000` and `stack size = 0x90008000` as follows,

1. “addiu \$1, \$zero, -28671”=> (\$1 = 0 + 0xffff9001) => \$1 = 0xffff9001.
2. “shl \$1, \$1, 16;”=> \$1 = 0x90010000.
3. “addiu \$1, \$1, -32768”=> \$1 = (0x90010000 + 0xffff8000) => \$1 = 0x90008000.
4. “addu \$sp, \$sp, \$1”=> \$sp = (0x10000000 + 0x90008000) => \$sp = 0xa0008000.

The `Cpu0AnalyzeImmediate::GetShortestSeq()` will call `Cpu0AnalyzeImmediate::ReplaceADDiuSHLWithLUI()` to replace `addiu` and `shl` with a single instruction `lui` only. The effect is shown in the following table.

Table 3.5: Cpu0 stack adjustment instructions after replace addiu and shl with lui instruction

stack size range	ex. stack size	Cpu0 Prologue instructions	Cpu0 Epilogue instructions
0x8000 ~ 0xffff8	• 0x8000	• addiu \$sp, \$sp, -32768;	• ori \$1, \$zero, 32768; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff8	• 0x7fffffff8	• lui \$1, 32768; • addiu \$1, \$1, 8; • addu \$sp, \$sp, \$1;	• lui \$1, 32767; • ori \$1, \$1, 65528 • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff8	• 0x90008000	• lui \$1, 28671; • ori \$1, \$1, 32768; • addu \$sp, \$sp, \$1;	• lui \$1, 36865; • addiu \$1, \$1, -32768; • addu \$sp, \$sp, \$1;

Assume `sp = 0xA0008000` and `stack size = 0x90008000`, then `(0xA0008000 - 0x90008000) => 0x10000000`. Verify with the Cpu0 prologue instructions as follows,

1. “lui \$1, 28671”=> \$1 = 0x6fff0000.
2. “ori \$1, \$1, 32768”=> \$1 = (0x6fff0000 + 0x00008000) => \$1 = 0x6fff8000.
3. “addu \$sp, \$sp, \$1”=> \$sp = (0xa0008000 + 0x6fff8000) => \$sp = 0x10000000.

Verify with the Cpu0 epilogue instructions with `sp = 0x10000000` and `stack size = 0x90008000` as follows,

1. “lui \$1, 36865”=> \$1 = 0x90010000.
2. “addiu \$1, \$1, -32768”=> \$1 = (0x90010000 + 0xffff8000) => \$1 = 0x90008000.
3. “addu \$sp, \$sp, \$1”=> \$sp = (0x10000000 + 0x90008000) => \$sp = 0xa0008000.

The file `ch3_largeframe.cpp` includes the large frame test.

Running Chapter3_5 with `ch3_largeframe.cpp` will produce the following result.

Ibdex/input/ch3_largeframe.cpp

```
int test_largeframe() {
    int a[469753856];

    return 0;
}
```

```

118-165-78-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3_largeframe.cpp -emit-llvm -o ch3_largeframe.bc
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch3_largeframe.bc.bc -o -
...
.section .mdebug.abi032
.previous
.file "ch3_largeframe.bc"
.globl _Z16test_largegframev
.align 2
.type _Z16test_largegframev,@function
.ent _Z16test_largegframev # @_Z16test_largegframev
_Z16test_largegframev:
.frame $fp,1879015424,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
.set noat
# BB#0:
    lui $1, 36865
    addiu $1, $1, -32768
    addu $sp, $sp, $1
    addiu $2, $zero, 0
    lui $1, 28672
    addiu $1, $1, -32768
    addu $sp, $sp, $1
    ret $lr
.set at
.set macro
.set reorder
.end _Z16test_largegframev
$func_end0:
.size _Z16test_largegframev, ($func_end0)-_Z16test_largegframev

```

3.6 Data operands DAGs

From the above or the compiler book, you can see that all the OP codes are the internal nodes in DAG graphs, and operands are the leaves of DAGs.

To develop your backend, you can copy the related data operand DAG nodes from other backends since the IR data nodes are handled by all backends.

Regarding data DAG nodes, you can understand some of them through `Cpu0InstrInfo.td` and find them using the following command:

```
grep -R "<datadag>"`find llvm/include/llvm`
```

By spending a little more time thinking or making educated guesses, you can identify them. Some data DAGs are well understood, some are partially understood, and some remain unknown—but that is acceptable.

Here is a list of some data DAGs we understand and have encountered so far:

include/llvm/Target/TargetSelectionDAG.td

```
// PatLeaf's are pattern fragments that have no operands. This is just a helper
// to define immediates and other common things concisely.
class PatLeaf<dag frag, code pred = [{}], SDNodeXForm xform = NOOP_SDNodeXForm>
: PatFrag<(ops), frag, pred, xform>;

// ImmLeaf is a pattern fragment with a constraint on the immediate. The
// constraint is a function that is run on the immediate (always with the value
// sign extended out to an int64_t) as Imm. For example:
//
// def immSExt8 : ImmLeaf<i16, [{ return (char)Imm == Imm; }]>;
//
// this is a more convenient form to match 'imm' nodes in than PatLeaf and also
// is preferred over using PatLeaf because it allows the code generator to
// reason more about the constraint.
//
// If FastIsel should ignore all instructions that have an operand of this type,
// the FastIselShouldIgnore flag can be set. This is an optimization to reduce
// the code size of the generated fast instruction selector.
class ImmLeaf<ValueType vt, code pred, SDNodeXForm xform = NOOP_SDNodeXForm>
: PatFrag<(ops), (vt imm), [{}], xform> {
let ImmediateCode = pred;
bit FastIselShouldIgnore = 0;
}
```

libdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
// Signed Operand
def simm16      : Operand<i32> {
let DecoderMethod= "DecodeSimm16";
}

def shamt       : Operand<i32>;

// Unsigned Operand
def uimm16      : Operand<i32> {
let PrintMethod = "printUnsignedImm";
}

// Address operand
def mem : Operand<iPTR> {
let PrintMethod = "printMemOperand";
let MIOperandInfo = (ops GPROut, simm16);
let EncoderMethod = "getMemEncoding";
}

// Transformation Function - get the lower 16 bits.
def LO16 : SDNodeXForm<imm, [{  
    return getImm(N, N->getZExtValue() & 0xffff);  
}]>;
```

(continues on next page)

(continued from previous page)

```
// Transformation Function - get the higher 16 bits.
def HI16 : SDNodeXForm<imm, [<br>
    return getImm(N, (N->getZExtValue() >> 16) & 0xffff);<br>
}]>;
```

```
// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addi, andi
def immSExt16 : PatLeaf<(imm), [<br>    { return isInt<16>(N->getSExtValue()); }]>;
```

```
// Node immediate fits as 16-bit zero extended on target immediate.
// The LO16 param means that only the lower 16 bits of the node
// immediate are caught.
// e.g. addiu, sltiu
def immZExt16 : PatLeaf<(imm), [<br>
    if (N->getValueType(0) == MVT::i32)
        return (uint32_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
    else
        return (uint64_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
], LO16>;
```

```
// Immediate can be loaded with LUi (32-bit int with lower 16-bit cleared).
def immLow16Zero : PatLeaf<(imm), [<br>
    int64_t Val = N->getSExtValue();
    return isInt<32>(Val) && !(Val & 0xffff);<br>
}]>;
```

```
// shamt field must fit in 5 bits.
def immZExt5 : ImmLeaf<i32, [<br>    {return Imm == (Imm & 0x1f);} ]>;
```

```
// Cpu0 Address Mode! SDNode frameindex could possibly be a match
// since load and store instructions from stack used it.
def addr :
    ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;
```

```
//=====
// Pattern fragment for load/store
//=====
```

```
class AlignedLoad<PatFrag Node> :
    PatFrag<(ops node:$ptr), (Node node:$ptr), [<br>
        LoadSDNode *LD = cast<LoadSDNode>(N);
        return LD->getMemoryVT().getSizeInBits()/8 <= LD->getAlignment();<br>
    }]>;
```

```
class AlignedStore<PatFrag Node> :
    PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr), [<br>
        StoreSDNode *SD = cast<StoreSDNode>(N);
        return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();<br>
    }]>;
```

```
def load_a      : AlignedLoad<load>;
```

```
def store_a     : AlignedStore<store>;
```

As mentioned in the subsection “Instruction Selection” of the last chapter, `immSExt16` is a data leaf DAG node that returns `true` if its value is within the range of a signed 16-bit integer. The `load_a`, `store_a`, and others are similar, but they check for alignment.

The `mem` is explained in Chapter3_2 for printing operands, and `addr` is explained in Chapter3_3 for data DAG selection.

The `simm16, ...,` are inherited from `Operand<i32>` because Cpu0 is a 32-bit architecture. It may exceed 16 bits, so the `immSExt16` pattern leaf is used to constrain it, as seen in the `ADDiu` example mentioned in the last chapter.

The `PatLeaf immZExt16`, `immLow16Zero`, and `ImmLeaf immZExt5` are similar to `immSExt16`. Summary of this Chapter

Summary the functions for llvm backend stages as the following table.

```
118-165-79-200:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc
-debug-pass=Structure -o -
...
Machine Branch Probability Analysis
ModulePass Manager
FunctionPass Manager
...
CPU0 DAG->DAG Pattern Instruction Selection
Initial selection DAG
Optimized lowered selection DAG
Type-legalized selection DAG
Optimized type-legalized selection DAG
Legalized selection DAG
Optimized legalized selection DAG
Instruction selection
Selected selection DAG
Scheduling
...
Greedy Register Allocator
...
Prologue/Epilogue Insertion & Frame Finalization
...
Post-RA pseudo instruction expansion pass
...
Cpu0 Assembly Printer
```

Table 3.6: Functions for llvm backend stages

Stage	Function
Before CPU0 DAG->DAG Pattern Instruction Selection	<ul style="list-style-type: none"> Cpu0TargetLowering::LowerFormalArguments Cpu0TargetLowering::LowerReturn Cpu0DAGToDAGISel::Select
Instruction selection	
Prologue/Epilogue Insertion & Frame Finalization	<ul style="list-style-type: none"> Determine spill callee saved registers Spill callee saved registers Prolog Epilog Handle stack slot for local variables
Post-RA pseudo instruction expansion pass	<ul style="list-style-type: none"> Cpu0SEFrameLowering::determineCalleeSaves Cpu0SEFrameLowering::spillCalleeSavedRegisters Cpu0SEFrameLowering::emitPrologue Cpu0SEFrameLowering::emitEpilogue Cpu0RegisterInfo::eliminateFrameIndex Cpu0SEInstrInfo::expandPostRAPseudo
Cpu0 Assembly Printer	<ul style="list-style-type: none"> Cpu0AsmPrinter.cpp, Cpu0MCInstLower.cpp Cpu0InstPrinter.cpp

We add a pass in Instruction Section stage in section “Add Cpu0DAGToDAGISel class”. You can embed your code into other passes like that. Please check `CodeGen/Passes.h` for the information. Remember the pass is called according the function unit as the `llc -debug-pass=Structure` indicated.

We have finished a simple compiler for cpu0 which only supports **ld, st, addiu, ori, lui, addu, shl** and **ret** 8 instructions.

We add a pass in the Instruction Selection stage in the section “Add Cpu0DAGToDAGISel class.” You can embed your code into other passes similarly. Please check `CodeGen/Passes.h` for more information. Remember that passes are called according to the function unit, as indicated by the command `llc -debug-pass=Structure`.

We have completed a simple compiler for Cpu0, which only supports eight instructions: **ld, st, addiu, ori, lui, addu, shl**, and **ret**.

We are satisfied with this result. However, you might think, “After writing so much code, we only implemented these eight instructions!” The key takeaway is that we have built a framework for the Cpu0 target machine. (Refer to the LLVM backend structure class inheritance tree earlier in this chapter.)

So far, we have written over 3,000 lines of source code, including comments, across multiple files: `*.cpp, *.h, *.td, and CMakeLists.txt`. You can count them using the command:

```
wc `find dir -name *.cpp`
```

for `*.cpp, *.h, *.td, and *.txt` files. In contrast, the LLVM front-end tutorial contains only about 700 lines of source code (excluding comments).

Don’t be discouraged by these results. In reality, writing a backend starts slowly but speeds up over time.

For comparison:

- Clang has over **500,000** lines of source code (including comments) in the `clang/lib` directory, supporting both C++ and Objective-C.
- The MIPS backend in LLVM 3.1 contains **15,000** lines (with comments).
- Even the complex x86 CPU backend, which is CISC externally but RISC internally (using micro-instructions), has only **45,000** lines in LLVM 3.1 (with comments).

In the next chapter, we will demonstrate how adding support for a new instruction is as easy as 1-2-3!

ARITHMETIC AND LOGIC INSTRUCTIONS

- *Arithmetic*
 - $+$, $-$, $*$, $<<$, and $>>$
 - *Display LLVM DAG Nodes With Graphviz*
 - *Operator % and /*
 - * *DAG Representation of %*
 - * *ARM Solution*
 - * *MIPS Solution*
 - * *Full Support for % and /*
 - * *Rotate Instructions*
- *Logical Instructions*
- *Summary*

This chapter first adds support for more Cpu0 arithmetic instructions. The section *Display LLVM DAG Nodes With Graphviz* will show you the steps of DAG optimization and their corresponding `llc` display options. These DAG translations exist at various optimization steps and can be displayed using the Graphviz tool, which provides useful graphical information.

Support for logic instructions will follow the arithmetic section. Although the LLVM backend only handles IR, we derive the IR from corresponding C operators using designed C example code. Instead of focusing on class relationships in the backend structure, as in the previous chapter, readers should now focus on mapping C operators to LLVM IR and defining the mapping relationship between IR and instructions in `.td` files.

The **HILo** and **C0** register classes are introduced in this chapter. Readers will learn how to handle additional register classes beyond general-purpose registers and understand why they are needed.

4.1 Arithmetic

The code added in `Chapter4_1/` to support arithmetic instructions is summarized as follows:

Ibdex/chapters/Chapter4_1/Cpu0Subtarget.cpp

```
static cl::opt<bool> EnableOverflowOpt
    ("cpu0-enable-overflow", cl::Hidden, cl::init(false),
     cl::desc("Use trigger overflow instructions add and sub \
instead of non-overflow instructions addu and subu"));

Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                           StringRef FS, bool little,
                           const Cpu0TargetMachine &_TM) :

...

EnableOverflow = EnableOverflowOpt;

...
}
```

Ibdex/chapters/Chapter4_1/Cpu0InstrInfo.td

```
// Only op DAG can be disabled by ch4_1, data DAG cannot.
def SDT_Cpu0DivRem      : SDTypeProfile<0, 2,
                           [SDTCisInt<0>,
                            SDTCisSameAs<0, 1>]>;

// DivRem(u) nodes
def Cpu0DivRem      : SDNode<"Cpu0ISD::DivRem", SDT_Cpu0DivRem,
                       [SDNPOutGlue]>;
def Cpu0DivRemU     : SDNode<"Cpu0ISD::DivRemU", SDT_Cpu0DivRem,
                       [SDNPOutGlue]>;

let Predicates = [Ch4_1] in {
    class shift_rotate_reg<bits<8> op, bits<4> isRotate, string instr_asm,
        SDNode OpNode, RegisterClass RC>:
        FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
        !strconcat(instr_asm, "\t$ra, $rb, $rc"),
        [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], IIAlu> {
            let shamrt = 0;
        }
}
```

```
let Predicates = [Ch4_1] in {
// Mul, Div
    class Mult<bits<8> op, string instr_asm, InstrItinClass itin,
        RegisterClass RC, list<Register> DefRegs>:
        FA<op, (outs), (ins RC:$ra, RC:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], itin> {
            let rc = 0;
            let shamrt = 0;
            let isCommutable = 1;
            let Defs = DefRegs;
```

(continues on next page)

(continued from previous page)

```

let hasSideEffects = 0;
}

class Mult32<bits<8> op, string instr_asm, InstrItinClass itin>:
    Mult<op, instr_asm, itin, CPURegs, [HI, LO]>

class Div<SDNode opNode, bits<8> op, string instr_asm, InstrItinClass itin,
    RegisterClass RC, list<Register> DefRegs>:
    FA<op, (outs), (ins RC:$ra, RC:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"),
        [(opNode RC:$ra, RC:$rb)], itin> {
    let rc = 0;
    let shamrt = 0;
    let Defs = DefRegs;
}

class Div32<SDNode opNode, bits<8> op, string instr_asm, InstrItinClass itin>:
    Div<opNode, op, instr_asm, itin, CPURegs, [HI, LO]>

// Move from Lo/Hi
class MoveFromLOHI<bits<8> op, string instr_asm, RegisterClass RC,
    list<Register> UseRegs>:
    FA<op, (outs RC:$ra), (ins),
        !strconcat(instr_asm, "\t$ra"), [], IIHiLo> {
    let rb = 0;
    let rc = 0;
    let shamrt = 0;
    let Uses = UseRegs;
    let hasSideEffects = 0;
}

// Move to Lo/Hi
class MoveToLOHI<bits<8> op, string instr_asm, RegisterClass RC,
    list<Register> DefRegs>:
    FA<op, (outs), (ins RC:$ra),
        !strconcat(instr_asm, "\t$ra"), [], IIHiLo> {
    let rb = 0;
    let rc = 0;
    let shamrt = 0;
    let Defs = DefRegs;
    let hasSideEffects = 0;
}

// Move from C0 (co-processor 0) Register
class MoveFromC0<bits<8> op, string instr_asm, RegisterClass RC>:
    FA<op, (outs), (ins RC:$ra, C0Regs:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
    let rc = 0;
    let shamrt = 0;
    let hasSideEffects = 0;
}

```

(continues on next page)

(continued from previous page)

```
// Move to C0 Register
class MoveToC0<bits<8> op, string instr_asm, RegisterClass RC>:
    FA<op, (outs C0Regs:$ra), (ins RC:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
    let rc = 0;
    let shamt = 0;
    let hasSideEffects = 0;
}

// Move from C0 register to C0 register
class C0Move<bits<8> op, string instr_asm>:
    FA<op, (outs C0Regs:$ra), (ins C0Regs:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
    let rc = 0;
    let shamt = 0;
    let hasSideEffects = 0;
}
} // let Predicates = [Ch4_1]
```

```
let Predicates = [Ch4_1] in {
let Predicates = [DisableOverflow] in {
def SUBu      : ArithLogicR<0x12, "subu", sub, IIAlu, CPURegs>;
}
let Predicates = [EnableOverflow] in {
def ADD       : ArithLogicR<0x13, "add", add, IIAlu, CPURegs, 1>;
def SUB       : ArithLogicR<0x14, "sub", sub, IIAlu, CPURegs>;
}
def MUL       : ArithLogicR<0x17, "mul", mul, IIImul, CPURegs, 1>;
}
```

```
let Predicates = [Ch4_1] in {
// sra is IR node for ash LLVM IR instruction of .bc
def ROL       : shift_rotate_imm32<0x1c, 0x01, "rol", rotl>;
def ROR       : shift_rotate_imm32<0x1d, 0x01, "ror", rotr>;
}
```

```
let Predicates = [Ch4_1] in {
// srl is IR node for lshr LLVM IR instruction of .bc
def SHR       : shift_rotate_imm32<0x1f, 0x00, "shr", srl>;
def SRA       : shift_rotate_imm32<0x20, 0x00, "sra", sra>;
def SRAV      : shift_rotate_reg<0x21, 0x00, "sra", sra, CPURegs>;
def SHLV      : shift_rotate_reg<0x22, 0x00, "shlv", shl, CPURegs>;
def SHRV      : shift_rotate_reg<0x23, 0x00, "shrv", srl, CPURegs>;
def ROLV      : shift_rotate_reg<0x24, 0x01, "rolv", rotl, CPURegs>;
def RORV      : shift_rotate_reg<0x25, 0x01, "rorv", rotr, CPURegs>;
}
```

```
let Predicates = [Ch4_1] in {
/// Multiply and Divide Instructions.
def MULT      : Mult32<0x41, "mult", IIImul>;
def MULTu     : Mult32<0x42, "multu", IIImul>;
```

(continues on next page)

(continued from previous page)

```

def SDIV      : Div32<Cpu0DivRem, 0x43, "div", IIIdiv>;
def UDIV      : Div32<Cpu0DivRemU, 0x44, "divu", IIIdiv>;

def MFHI      : MoveFromLOHI<0x46, "mfhi", CPURegs, [HI]>;
def MFLO      : MoveFromLOHI<0x47, "mflo", CPURegs, [LO]>;
def MTHI      : MoveToLOHI<0x48, "mthi", CPURegs, [HI]>;
def MTLO      : MoveToLOHI<0x49, "mtlo", CPURegs, [LO]>;

def MFC0      : MoveFromC0<0x50, "mfc0", CPURegs>;
def MTC0      : MoveToC0<0x51, "mtc0", CPURegs>;

def C0MOVE    : C0Move<0x52, "c0mov">;
}

```

Ibdex/chapters/Chapter4_1/Cpu0ISelLowering.h

```
SDValue PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI) const override;
```

Ibdex/chapters/Chapter4_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
  : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
setOperationAction(ISD::SDIV, MVT::i32, Expand);
setOperationAction(ISD::SREM, MVT::i32, Expand);
setOperationAction(ISD::UDIV, MVT::i32, Expand);
setOperationAction(ISD::UREM, MVT::i32, Expand);
```

```
setTargetDAGCombine(ISD::SDIVREM);
setTargetDAGCombine(ISD::UDIVREM);
```

```
...
}
```

```
static SDValue performDivRemCombine(SDNode *N, SelectionDAG& DAG,
                                    TargetLowering::DAGCombinerInfo &DCI,
                                    const Cpu0Subtarget &Subtarget) {
  if (DCI.isBeforeLegalizeOps())
    return SDValue();

  EVT Ty = N->getValueType(0);
  unsigned LO = Cpu0::LO;
  unsigned HI = Cpu0::HI;
  unsigned Opc = N->getOpcode() == ISD::SDIVREM ? Cpu0ISD::DivRem :
                                                       Cpu0ISD::DivRemU;
  SDLoc DL(N);
}
```

(continues on next page)

(continued from previous page)

```

SDValue DivRem = DAG getNode(Opc, DL, MVT::Glue,
                           N->getOperand(0), N->getOperand(1));
SDValue InChain = DAG.getEntryNode();
SDValue InGlue = DivRem;

// insert MFLO
if (N->hasAnyUseOfValue(0)) {
    SDValue CopyFromLo = DAG.getCopyFromReg(InChain, DL, LO, Ty,
                                             InGlue);
    DAG.ReplaceAllUsesOfValueWith(SDValue(N, 0), CopyFromLo);
    InChain = CopyFromLo.getValue(1);
    InGlue = CopyFromLo.getValue(2);
}

// insert MFHI
if (N->hasAnyUseOfValue(1)) {
    SDValue CopyFromHi = DAG.getCopyFromReg(InChain, DL,
                                             HI, Ty, InGlue);
    DAG.ReplaceAllUsesOfValueWith(SDValue(N, 1), CopyFromHi);
}

return SDValue();
}

SDValue Cpu0TargetLowering::PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI)
const {
SelectionDAG &DAG = DCI.DAG;
unsigned Opc = N->getOpcode();

switch (Opc) {
default: break;
case ISD::SDIVREM:
case ISD::UDIVREM:
    return performDivRemCombine(N, DAG, DCI, Subtarget);
}

return SDValue();
}

```

Index/chapters/Chapter4_1/Cpu0RegisterInfo.td

```

let Namespace = "Cpu0" in {

    // Hi/Lo registers number and name
    def HI    : Cpu0Reg<0, "ac0">, DwarfRegNum<[18]>;
    def LO    : Cpu0Reg<0, "ac0">, DwarfRegNum<[19]>;

}
...
```

```
// Hi/Lo Registers class
def HILO : RegisterClass<"Cpu0", [i32], 32, (add HI, LO);
```

Ibdex/chapters/Chapter4_1/Cpu0Schedule.td

```
def IIIHiLo : InstrItinClass;
def IIImul : InstrItinClass;
def IIIdiv : InstrItinClass;

def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
    InstrItinData<IIIHiLo>, [InstrStage<1, [IMULDIV]>],
    InstrItinData<IIImul>, [InstrStage<17, [IMULDIV]>],
    InstrItinData<IIIdiv>, [InstrStage<38, [IMULDIV]>],
] >;
```

Ibdex/chapters/Chapter4_1/Cpu0SEISelDAGToDAG.h

```
std::pair<SDNode *, SDNode *> selectMULT(SDNode *N, unsigned Opc,
                                              const SDLoc &DL, EVT Ty, bool HasLo,
                                              bool HasHi);
```

Ibdex/chapters/Chapter4_1/Cpu0SEISelDAGToDAG.cpp

```
/// Select multiply instructions.
std::pair<SDNode *, SDNode *>
Cpu0SEISelDAGToDAG::selectMULT(SDNode *N, unsigned Opc, const SDLoc &DL, EVT Ty,
                                bool HasLo, bool HasHi) {
    SDNode *Lo = 0, *Hi = 0;
    SDNode *Mul = CurDAG->getMachineNode(Opc, DL, MVT::Glue, N->getOperand(0),
                                            N->getOperand(1));
    SDValue InFlag = SDValue(Mul, 0);

    if (HasLo) {
        Lo = CurDAG->getMachineNode(Cpu0::MFLO, DL,
                                       Ty, MVT::Glue, InFlag);
        InFlag = SDValue(Lo, 1);
    }
    if (HasHi)
        Hi = CurDAG->getMachineNode(Cpu0::MFHI, DL,
                                       Ty, InFlag);

    return std::make_pair(Lo, Hi);
}
```

```
bool Cpu0SEISelDAGToDAG::trySelect(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();
    SDLoc DL(Node);
```

(continues on next page)

(continued from previous page)

```
///
// Instruction Selection not handled by the auto-generated
// tablegen selection should be handled here.
///

///
// Instruction Selection not handled by the auto-generated
// tablegen selection should be handled here.
///

EVT NodeTy = Node->getValueType(0);
unsigned MultOpc;

switch(Opcode) {
default: break;
```

```
case ISD::MULHS:
case ISD::MULHU: {
    MultOpc = (Opcode == ISD::MULHU ? Cpu0::MULTu : Cpu0::MULT);
    auto LoHi = selectMULT(Node, MultOpc, DL, NodeTy, false, true);
    ReplaceNode(Node, LoHi.second);
    return true;
}

case ISD::Constant: {
    const ConstantSDNode *CN = dyn_cast<ConstantSDNode>(Node);
    unsigned Size = CN->getValueSizeInBits(0);

    if (Size == 32)
        break;

    return true;
}
```

```
}
```

Ibdex/chapters/Chapter4_1/Cpu0SEInstrInfo.h

```
void copyPhysReg(MachineBasicBlock &MBB, MachineBasicBlock::iterator MI,
                  const DebugLoc &DL, MCRegister DestReg, MCRegister SrcReg,
                  bool KillSrc) const override;
```

Ibdex/chapters/Chapter4_1/Cpu0SEInstrInfo.cpp

```
void Cpu0SEInstrInfo::copyPhysReg(MachineBasicBlock &MBB,
                                   MachineBasicBlock::iterator I,
                                   const DebugLoc &DL, MCRegister DestReg,
                                   MCRegister SrcReg, bool KillSrc) const {
```

(continues on next page)

(continued from previous page)

```

unsigned Opc = 0, ZeroReg = 0;

if (Cpu0::CPUREgsRegClass.contains(DestReg)) { // Copy to CPU Reg.
    if (Cpu0::CPUREgsRegClass.contains(SrcReg))
        Opc = Cpu0::ADDu, ZeroReg = Cpu0::ZERO;
    else if (SrcReg == Cpu0::HI)
        Opc = Cpu0::MFHI, SrcReg = 0;
    else if (SrcReg == Cpu0::LO)
        Opc = Cpu0::MFLO, SrcReg = 0;
}
else if (Cpu0::CPUREgsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
    if (DestReg == Cpu0::HI)
        Opc = Cpu0::MTHI, DestReg = 0;
    else if (DestReg == Cpu0::LO)
        Opc = Cpu0::MTLO, DestReg = 0;
}

assert(Opc && "Cannot copy registers");

MachineInstrBuilder MIB = BuildMI(MBB, I, DL, get(Opc));

if (DestReg)
    MIB.addReg(DestReg, RegState::Define);

if (ZeroReg)
    MIB.addReg(ZeroReg);

if (SrcReg)
    MIB.addReg(SrcReg, getKillRegState(KillSrc));
}

```

4.1.1 +, -, *, <<, and >>

The **ADDu**, **ADD**, **SUBu**, **SUB**, and **MUL** instructions defined in `Chapter4_1/Cpu0InstrInfo.td` correspond to the **+**, **-**, ***** operators. **SHL** (defined earlier) and **SHLV** are used for **<<**, while **SRA**, **SRAV**, **SHR**, and **SHRV** handle **>>**.

In RISC CPUs like MIPS, the multiply/divide function unit and add/sub/logic unit are implemented using separate hardware circuits with distinct data paths. Cpu0 follows the same approach, allowing these function units to execute simultaneously (instruction-level parallelism). Refer to¹ for details on instruction itineraries.

`Chapter4_1/` supports the **+**, **-**, *****, **<<**, **and** **>>** operators in C. The corresponding LLVM IR instructions are **add**, **sub**, **mul**, **shl**, and **ashr**.

The **ashr** instruction (arithmetic shift right) shifts the first operand right by a specified number of bits with sign extension. In short, **ashr** performs “shift with sign extension fill.”

Note

ashr

¹ http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html

Example:

```
<result> = ashr i32 4, 1 ; yields {i32}:result = 2
<result> = ashr i8 -2, 1 ; yields {i8}:result = -1
<result> = ashr i32 1, 32 ; undefined
```

The behavior of the C `>>` operator for negative operands is implementation-dependent. Most compilers translate it as a “shift with sign extension fill,” which is equivalent to the MIPS `sra` instruction. The Microsoft website provides the following explanation:

Note

`>>`, Microsoft Specific

The result of a right shift of a signed negative quantity is implementation dependent. Although Microsoft C++ propagates the most-significant bit to fill vacated bit positions, there is no guarantee that other implementations will do likewise.

In addition to `ashr`, LLVM provides the `lshr` instruction (“logical shift right with zero fill”), which MIPS implements with the `srl` instruction.

Note

`lshr`

Example: `<result> = lshr i8 -2, 1 ; yields {i8}:result = 0x7FFFFFFF`

LLVM defines `sra` as the IR node for `ashr` and `srl` for `lshr`. (It’s unclear why LLVM does not directly use “`ashr`” and “`lshr`” as IR node names.) The following table summarizes C `>>` operator implementations:

Table 4.1: C operator `>>` implementation

Description	Shift with zero filled	Shift with signed extension filled
symbol in .bc	lshr	ashr
symbol in IR node	srl	sra
Mips instruction	srl	sra
Cpu0 instruction	shr	sra
signed example before $x >> 1$	0xffffffff i.e. -2	0xffffffff i.e. -2
signed example after $x >> 1$	0x7fffffff i.e. 2G-1	0xffffffff i.e. -1
unsigned example before $x >> 1$	0xffffffff i.e. 4G-2	0xffffffff i.e. 4G-2
unsigned example after $x >> 1$	0x7fffffff i.e. 2G-1	0xffffffff i.e. 4G-1

lshr: Logical SHift Right

ashr: Arithmetic SHift right

srl: Shift Right Logically

sra: Shift Right Arithmetically

shr: SHift Right

If we define $x >> 1$ as $x = x / 2$, then `lshr` fails for some signed values (e.g., -2). Similarly, `ashr` fails for some unsigned values (e.g., $4G - 2$). Thus, to correctly handle both signed and unsigned integers, we need both `lshr` and `ashr`.

Table 4.2: C operator << implementation

Description	Shift with zero filled
symbol in .bc	shl
symbol in IR node	shl
Mips instruction	sll
Cpu0 instruction	shl
signed example before $x \ll 1$	0x40000000 i.e. 1G
signed example after $x \ll 1$	0x80000000 i.e -2G
unsigned example before $x \ll 1$	0x40000000 i.e. 1G
unsigned example after $x \ll 1$	0x80000000 i.e 2G

Again, consider the definition of $x \ll 1$ as $x = x * 2$. From the table on C operator << implementation, we see that **lshr** satisfies the case for “unsigned $x = 1G$ ” but fails for “signed $x = 1G$ ”. This is acceptable since $2G$ exceeds the 32-bit signed integer range ($-2G$ to $2G - 1$).

In the case of overflow, there is no way to retain the correct result within a register. Thus, any value stored in the register is acceptable. You can verify that **lshr** satisfies $x = x * 2$ for all $x \ll 1$, as long as the result remains within range, regardless of whether x is signed or unsigned².

The reference for the **ashr** instruction is available here³, and for **lshr** here⁴.

The instructions **sraw**, **shlv**, and **shrv** operate on two virtual input registers, while **sra**, ..., and others operate on one virtual input register and one constant operand.

Now, let’s build Chapter4_1/ and run it using the input file ch4_math.ll as follows:

Ibdex/input/ch4_math.ll

```
; Function Attrs: nounwind
define i32 @_Z9test_mathv() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %1 = load i32, i32* %a, align 4
    %2 = load i32, i32* %b, align 4

    %3 = add nsw i32 %1, %2
    %4 = sub nsw i32 %1, %2
    %5 = mul nsw i32 %1, %2
    %6 = shl i32 %1, 2
    %7 = ash r i32 %1, 2
    %8 = lshr i32 %1, 30
    %9 = shl i32 %1, %2
    %10 = ash r i32 128, %2
    %11 = ash r i32 %1, %2

    %12 = add nsw i32 %3, %4
    %13 = add nsw i32 %12, %5
    %14 = add nsw i32 %13, %6
    %15 = add nsw i32 %14, %7
```

(continues on next page)

² <https://open4tech.com/logical-vs-arithmetic-shift>

³ <http://llvm.org/docs/LangRef.html#ashr-instruction>

⁴ <http://llvm.org/docs/LangRef.html#lshr-instruction>

(continued from previous page)

```
%16 = add nsw i32 %15, %8
%17 = add nsw i32 %16, %9
%18 = add nsw i32 %17, %10
%19 = add nsw i32 %18, %11

ret i32 %19
}
```

```
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_math.ll -o -
...
    ld      $2, 0($sp)
    ld      $3, 4($sp)
    subu   $4, $3, $2
    addu   $5, $3, $2
    addu   $4, $5, $4
    mul    $5, $3, $2
    addu   $4, $4, $5
    shl    $5, $3, 2
    addu   $4, $4, $5
    sra    $5, $3, 2
    addu   $4, $4, $5
    addiu  $5, $zero, 128
    shrv   $5, $5, $2
    addiu  $t9, $zero, 1
    shlv   $t9, $t9, $2
    srav   $2, $3, $2
    shr    $3, $3, 30
    addu   $3, $4, $3
    addu   $3, $3, $t9
    addu   $3, $3, $5
    addu   $2, $3, $2
    addiu $sp, $sp, 8
    ret    $lr
```

The example input `ch4_1_math.cpp` shown below is a C file that includes the operators `+`, `-`, `*`, `<<`, and `>>`.

Compiling this file with Clang will generate the corresponding LLVM IR instructions: **add**, **sub**, **mul**, **shl**, and **ashr**, as indicated in Chapter 3.

Ibdex/input/ch4_1_math.cpp

```
int test_math()
{
    int a = 5;
    int b = 2;
    unsigned int a1 = -5;
    int c, d, e, f, g, h, i;
    int j;
    unsigned int f1, g1, h1, i1;
```

(continues on next page)

(continued from previous page)

```

c = a + b;           // c = 7
d = a - b;           // d = 3
e = a * b;           // e = 10
f = (a << 2);      // f = 20
f1 = (a1 << 1);    // f1 = 0xffffffff6 = -10
g = (a >> 2);      // g = 1
g1 = (a1 >> 30);   // g1 = 0x03 = 3
h = (1 << a);      // h = 0x20 = 32
h1 = (1 << b);     // h1 = 0x04
i = (0x80 >> a);   // i = 0x04
i1 = (b >> a);     // i1 = 0x0
j = (-24 >> 2);   // j = -6

return (c+d+e+f+int(f1)+g+(int)g1+h+(int)h1+i+(int)i1+j);
// 7+3+10+20-10+1+3+32+4+4+0-6 = 68
}

```

Cpu0 instructions `add` and `sub` will trigger an overflow exception, whereas `addu` and `subu` truncate overflow values directly.

Compiling `ch4_1_addsuboverflow.cpp` with the following command:

```
l1c -cpu0-enable-overflow=true
```

will generate `add` and `sub` instructions as shown below:

Ibdex/input/ch4_1_addsuboverflow.cpp

```

#include "debug.h"

int test_add_overflow()
{
    int a = 0x70000000;
    int b = 0x20000000;
    int c = 0;

    c = a + b;

    return 0;
}

int test_sub_overflow()
{
    int a = -0x70000000;
    int b = 0x20000000;
    int c = 0;

    c = a - b;

    return 0;
}

```

```
118-165-78-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c ch4_1_addsuboverflow.cpp -emit-llvm -o ch4_1_addsuboverflow.bc
118-165-78-12:input Jonathan$ llvm-dis ch4_1_addsuboverflow.bc -o -
...
; Function Attrs: nounwind
define i32 @_Z13test_overflowv() #0 {
    ...
    %3 = add nsw i32 %1, %2
    ...
    %6 = sub nsw i32 %4, %5
    ...
}

118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
-cpu0-enable-overflow=true ch4_1_addsuboverflow.bc -o -
...
add      $3, $4, $3
...
sub      $3, $4, $3
...
```

In modern CPUs, programmers typically use truncate overflow instructions for C operators + and -.

However, by using the `-cpu0-enable-overflow=true` option, programmers can compile programs with overflow exception handling. This option is mainly used for debugging purposes. Compiling with this option can help identify bugs early and fix them efficiently.

4.1.2 Display LLVM DAG Nodes With Graphviz

The previous section displayed the DAG translation process in text format on the terminal using the `llc -debug` option.

The `llc` tool also supports graphical visualization. The [section Install other tools on iMac](#) explains how to download and install Graphviz, a tool for rendering DAGs.

This section introduces how to use `llc` with Graphviz for graphical display. Viewing DAGs graphically is often easier to interpret than reading raw text in the terminal. While not mandatory, this visualization can be very helpful, especially when debugging complex DAG transformations.

The following `llc` options allow graphical visualization of DAGs, as mentioned in the “SelectionDAG Instruction Selection Process” section of the LLVM Target-Independent Code Generator documentation⁵:

Note

The `llc` Graphviz DAG display options

- view-dag-combine1-dags displays the DAG after being built, before the first optimization pass.
- view-legalize-dags displays the DAG before Legalization.
- view-dag-combine2-dags displays the DAG before the second optimization pass.
- view-isel-dags displays the DAG before the Select phase.
- view-sched-dags displays the DAG before Scheduling.

⁵ <http://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process>

By tracking `llc -debug`, you can see the steps of DAG translation as follows,

```
Initial selection DAG
Optimized lowered selection DAG
Type-legalized selection DAG
Optimized type-legalized selection DAG
Legalized selection DAG
Optimized legalized selection DAG
Instruction selection
Selected selection DAG
Scheduling
...
```

Let's run `llc` with option `-view-dag-combine1-dags`, and open the output result with Graphviz as follows,

```
118-165-12-177:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -view-dag-combine1-dags -march=cpu0
-relocation-model=pic -filetype=asm ch4_1_mult.bc -o ch4_1_mult.cpu0.s
Writing '/tmp/llvm_84ibpm/dag.main.dot'... done.
118-165-12-177:input Jonathan$ Graphviz /tmp/llvm_84ibpm/dag.main.dot
```

It will show the `/tmp/llvm_84ibpm/dag.main.dot` as [Fig. 4.1](#).

[Fig. 4.1](#) is the stage of “Initial selection DAG”. List the other view options and their corresponding stages of DAG translation as follows,

Note

- `llc` Graphviz options and the corresponding stages of DAG translation
 - `-view-dag-combine1-dags`: Initial selection DAG
 - `-view-legalize-dags`: Optimized type-legalized selection DAG
 - `-view-dag-combine2-dags`: Legalized selection DAG
 - `-view-isel-dags`: Optimized legalized selection DAG
 - `-view-sched-dags`: Selected selection DAG

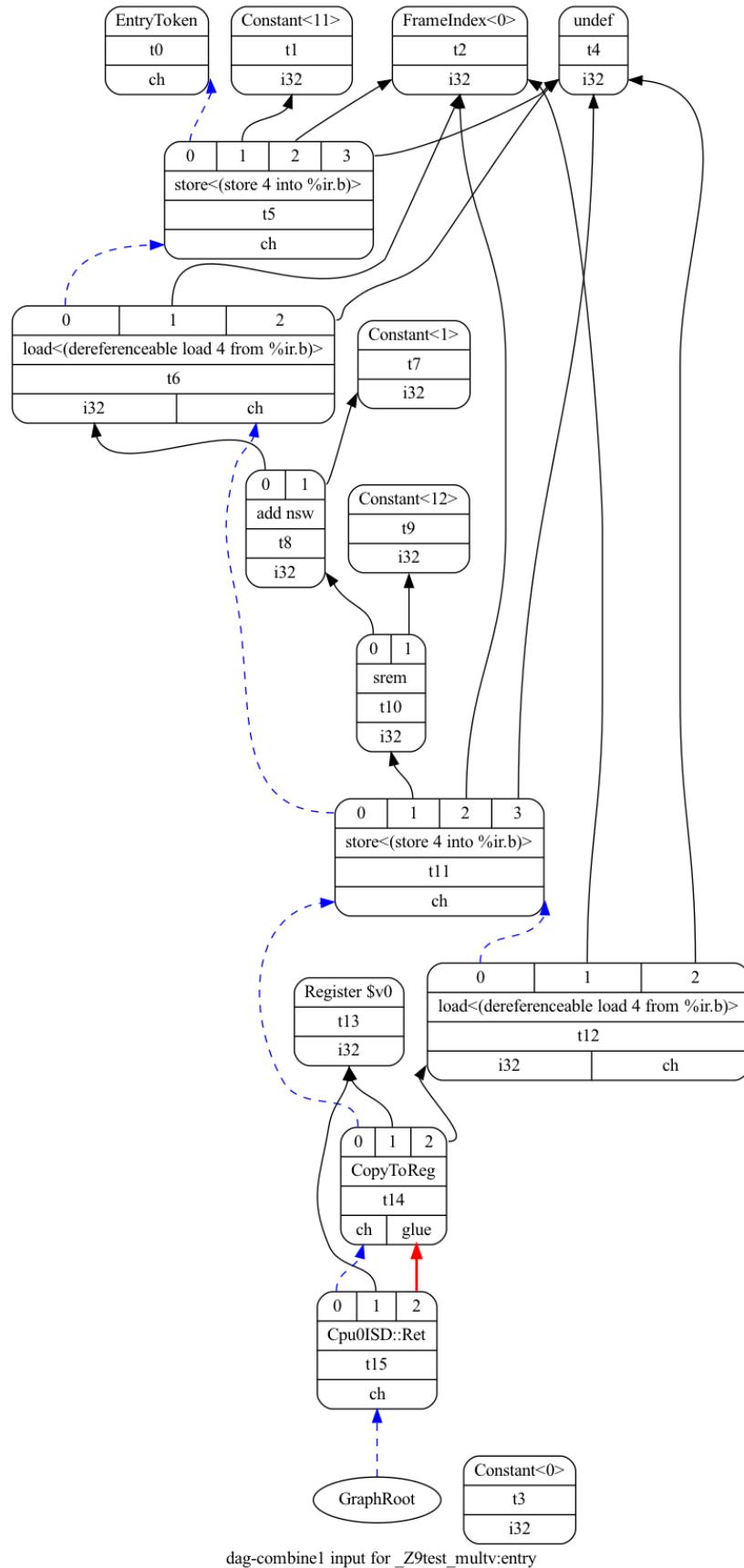
The `-view-isel-dags` option is particularly important and frequently used by LLVM backend developers. It displays the DAGs before instruction selection, providing crucial insight into how LLVM represents operations before they are mapped to target-specific instructions.

To write pattern-matching rules in the target description file (`.td`), backend developers need to understand the DAG nodes corresponding to specific C operators. This visualization helps in accurately defining these patterns.

4.1.3 Operator % and /

DAG Representation of %

The following example, `ch4_1_mult.cpp`, contains the C operator `%` (modulus). The corresponding LLVM IR is shown below:



Ibdex/input/ch4_1_mult.cpp

```

int test_mult()
{
    int b = 11;
//  unsigned int b = 11;

    b = (b+1)%12;

    return b;
}
...
define i32 @_Z8test_multv() #0 {
    %b = alloca i32, align 4
    store i32 11, i32* %b, align 4
    %1 = load i32* %b, align 4
    %2 = add nsw i32 %1, 1
    %3 = srem i32 %2, 12
    store i32 %3, i32* %b, align 4
    %4 = load i32* %b, align 4
    ret i32 %4
}

```

LLVM **srem** corresponds to the C operator “**%**”. Reference.⁶. The following note provides details on its syntax and behavior:

Note

‘srem’ Instruction

Syntax: <result> = srem <ty> <op1>, <op2> ; yields {ty}:result

Overview: The ‘**srem**’ instruction returns the remainder from the signed division of its two operands. This instruction also supports vector types, where the elements must be integers.

Arguments: The two arguments of the ‘**srem**’ instruction must be integers or vectors of integer values. Both operands must have identical types.

Semantics: This instruction returns the remainder of a signed division, meaning the result is either zero or has the same sign as the dividend (**op1**). It is not the modulo operator, where the result would have the same sign as the divisor (**op2**). For more details, see references such as The Math Forum or Wikipedia’s article on the modulo operation.

Note that signed integer remainder (**srem**) and unsigned integer remainder (**urem**) are distinct operations. For unsigned remainder, use ‘**urem**’ instead.

Taking the remainder of a division by zero results in undefined behavior. Overflow also causes undefined behavior, though it is a rare case. One such scenario is taking the remainder of a 32-bit division of **-2147483648** by **-1**, which cannot be directly represented. This rule allows **srem** to be implemented using division instructions that return both the quotient and the remainder.

Example: <result> = srem i32 4, %var ; yields {i32}: result = 4 % %var

To observe LLVM’s DAG representation of **srem**, run `llc -view-isel-dags --debug` on the input file `ch4_1_mult.bc` in `Chapter3_5/`. The **Optimized lowered selection DAG** statge is corresponding to DAG of —

⁶ <http://llvm.org/docs/LangRef.html#srem-instruction>

`view-isel-dags`. LLVM will display an error message along with the DAG output, as illustrated in the following **Optimized lowered selection DAG** and Fig. 4.2 below.

```
118-165-79-37:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -view-isel-dags -relocation-model=
pic -filetype=asm --debug ch4_1_mult.bc -o -
...
LLVM ERROR: Cannot select: t19: i32 = mulhs t8, Constant:i32<715827883>
```

```
Optimized lowered selection DAG: %bb.0 '_Z9test_multv:entry'
SelectionDAG has 22 nodes:
    t0: ch = EntryToken
    t5: ch = store<(store 4 into %ir.b)> t0, Constant:i32<11>, FrameIndex:i32<0>, ↵
    ↵undef:i32      // t5 = store 11 to FrameIndex(0)
    t6: i32, ch = load<(dereferenceable load 4 from %ir.b)> t5, FrameIndex:i32<0>, ↵
    ↵undef:i32      // t6 = load FrameIndex(0)
    t8: i32 = add nsw t6, Constant:i32<1>
    ↵           // t8 = add t6, 1 --> t8 = 12
    t22: i32 = sra t19, Constant:i32<1>
    ↵           // t22 = sra
    t28: i32 = srl t19, Constant:i32<31>
    t25: i32 = add t22, t28
    t26: i32 = mul t25, Constant:i32<12>
    t27: i32 = sub t8, t26
    t11: ch = store<(store 4 into %ir.b)> t6:1, t27, FrameIndex:i32<0>, undef:i32
    t12: i32, ch = load<(dereferenceable load 4 from %ir.b)> t11, FrameIndex:i32<0>, ↵
    ↵undef:i32
    t14: ch, glue = CopyToReg t11, Register:i32 $v0, t12
    t19: i32 = mulhs t8, Constant:i32<715827883>
    t15: ch = Cpu0ISD::Ret t14, Register:i32 $v0, t14:1
```

LLVM optimizes the **srem** operation by replacing division with multiplication in DAG optimization. This is because the **DIV** operation is more expensive in terms of execution time compared to **MUL**.

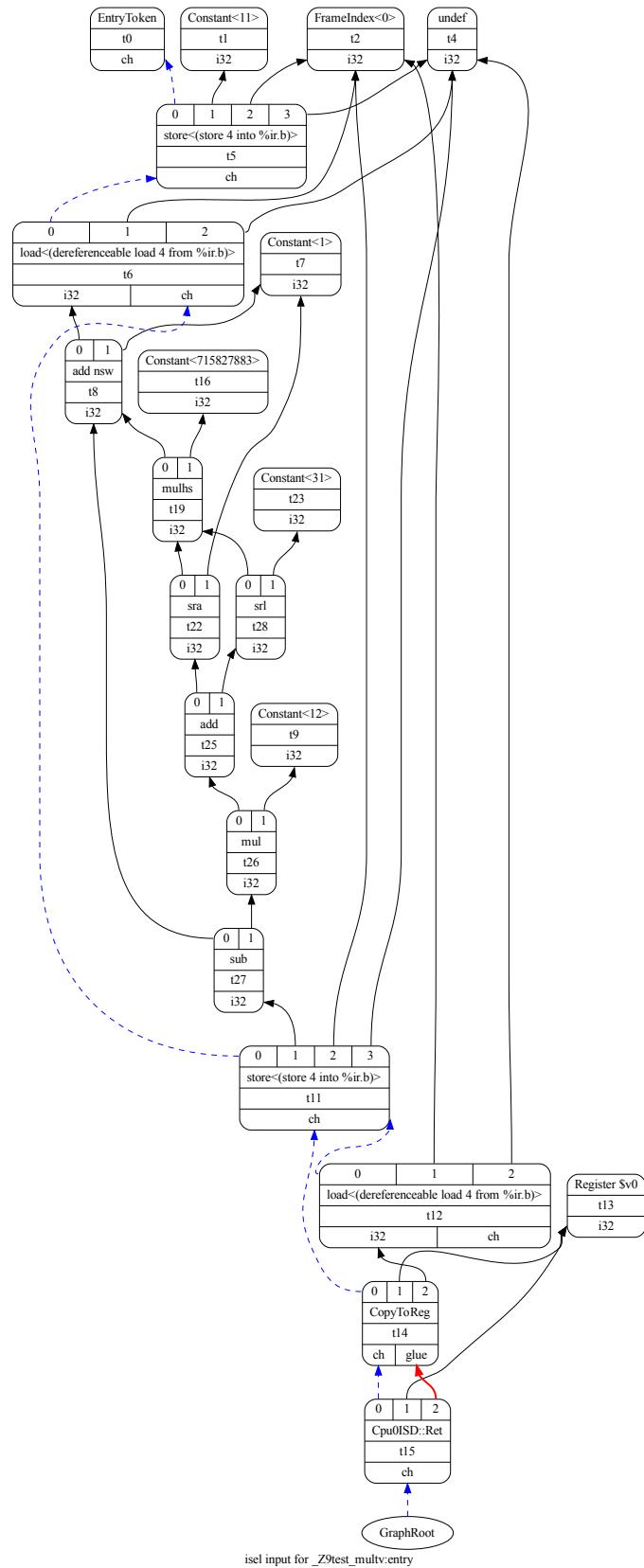
For example, the following C code:

```
int b = 11;
b = (b + 1) % 12;
```

is translated into DAGs, as shown in Fig. 4.2. The DAG representation is verified and explained by calculating the values at each node.

The computation follows these steps:

- t5 = store 11 to FrameIndex(0)
- t6 = load FrameIndex(0)
- t8 = add t6, 1 → t8 = 12
- 715827883 = 0x2AAAAAAAB
- **0xC * 0x2AAAAAAAB = 0x2,00000004**
- **mulhs(0xC, 0x2AAAAAAAB)** retrieves the signed multiplication's high word (upper 32 bits).
- A multiplication of two 32-bit operands typically produces a 64-bit result (e.g., **0x2, 0xAAAAAAAB**).
- In this case, the high word of the result is **0x2**.



- The final computation **sub(12, 12)** results in **0**, which correctly matches **(11 + 1) % 12**.

ARM Solution

To run this with an ARM-based solution, modify the following files in Chapter4_1/:

- Cpu0InstrInfo.td
- Cpu0ISelDAGToDAG.cpp

Make the necessary changes as follows:

Ibdex/chapters/Chapter4_1/Cpu0InstrInfo.td

```
/// Multiply and Divide Instructions.
def SMMUL : ArithLogicR<0x41, "smmul", mulhs, IIImul, CPUREgs, 1>;
def UMMUL : ArithLogicR<0x42, "ummul", mulhu, IIImul, CPUREgs, 1>;
//def MULT : Mult32<0x41, "mult", IIImul>;
//def MULTu : Mult32<0x42, "multu", IIImul>;
```

Ibdex/chapters/Chapter4_1/Cpu0ISelDAGToDAG.cpp

```
#if 0
/// Select multiply instructions.
std::pair<SDNode*, SDNode*>
Cpu0DAGToDAGISel::SelectMULT(SDNode *N, unsigned Opc, SDLoc DL, EVT Ty,
                               bool HasLo, bool HasHi) {
    SDNode *Lo = 0, *Hi = 0;
    SDNode *Mul = CurDAG->getMachineNode(Opc, DL, MVT::Glue, N->getOperand(0),
                                            N->getOperand(1));
    SDValue InFlag = SDValue(Mul, 0);

    if (HasLo) {
        Lo = CurDAG->getMachineNode(Cpu0::MFLO, DL,
                                       Ty, MVT::Glue, InFlag);
        InFlag = SDValue(Lo, 1);
    }
    if (HasHi)
        Hi = CurDAG->getMachineNode(Cpu0::MFHI, DL,
                                       Ty, InFlag);

    return std::make_pair(Lo, Hi);
}
#endif

/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {
...
switch(Opcode) {
default: break;
#endif
case ISD::MULHS:
case ISD::MULHU: {
```

(continues on next page)

(continued from previous page)

```

    MultOpc = (Opcode == ISD::MULHU ? Cpu0::MULTu : Cpu0::MULT);
    return SelectMULT(Node, MultOpc, DL, NodeTy, false, true).second;
}
#endif
...
}

```

Let's apply the above changes and run them with `ch4_1_mult.cpp` using the `llc -view-sched-dags` option to generate Fig. 4.3.

The **SMMUL** instruction is used to extract the high word of the multiplication result.

The following is the result of running the above changes with `ch4_1_mult.bc`.

```

118-165-66-82:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm
ch4_1_mult.bc -o -
...
# BB#0:                                # %entry
addiu $sp, $sp, -8
$tmp1:
.cfi_def_cfa_offset 8
addiu $2, $zero, 0
st $2, 4($fp)
addiu $2, $zero, 11
st $2, 0($fp)
lui $2, 10922
ori $3, $2, 43691
addiu $2, $zero, 12
smmul $3, $2, $3
shr $4, $3, 31
sra $3, $3, 1
addu $3, $3, $4
mul $3, $3, $2
subu $2, $2, $3
st $2, 0($fp)
addiu $sp, $sp, 8
ret $lr

```

MIPS Solution

MIPS uses the **MULT** instruction to perform multiplication, storing the high and low parts of the result in the **HI** and **LO** registers, respectively. After that, the **mfhi** and **mflo** instructions move the values from the HI/LO registers to general-purpose registers.

ARM's **SMMUL** instruction is optimized for cases where only the high part of the result is needed, as it ignores the low part. ARM also provides **SMULL** (signed multiply long) to obtain the full 64-bit result.

If only the low part of the result is needed, the **Cpu0 MUL** instruction can be used. The implementation in Chapter4_1/ follows the MIPS **MULT** style to minimize the number of added instructions. This approach makes **Cpu0** suitable as both a tutorial architecture for educational purposes and a learning resource for compiler design.

The following instructions are added in Chapter4_1/ for the MIPS-style implementation:

- **MULT, MULTu, MFHI, MFLO, MTHI, MTLO** in Chapter4_1/Cpu0InstrInfo.td

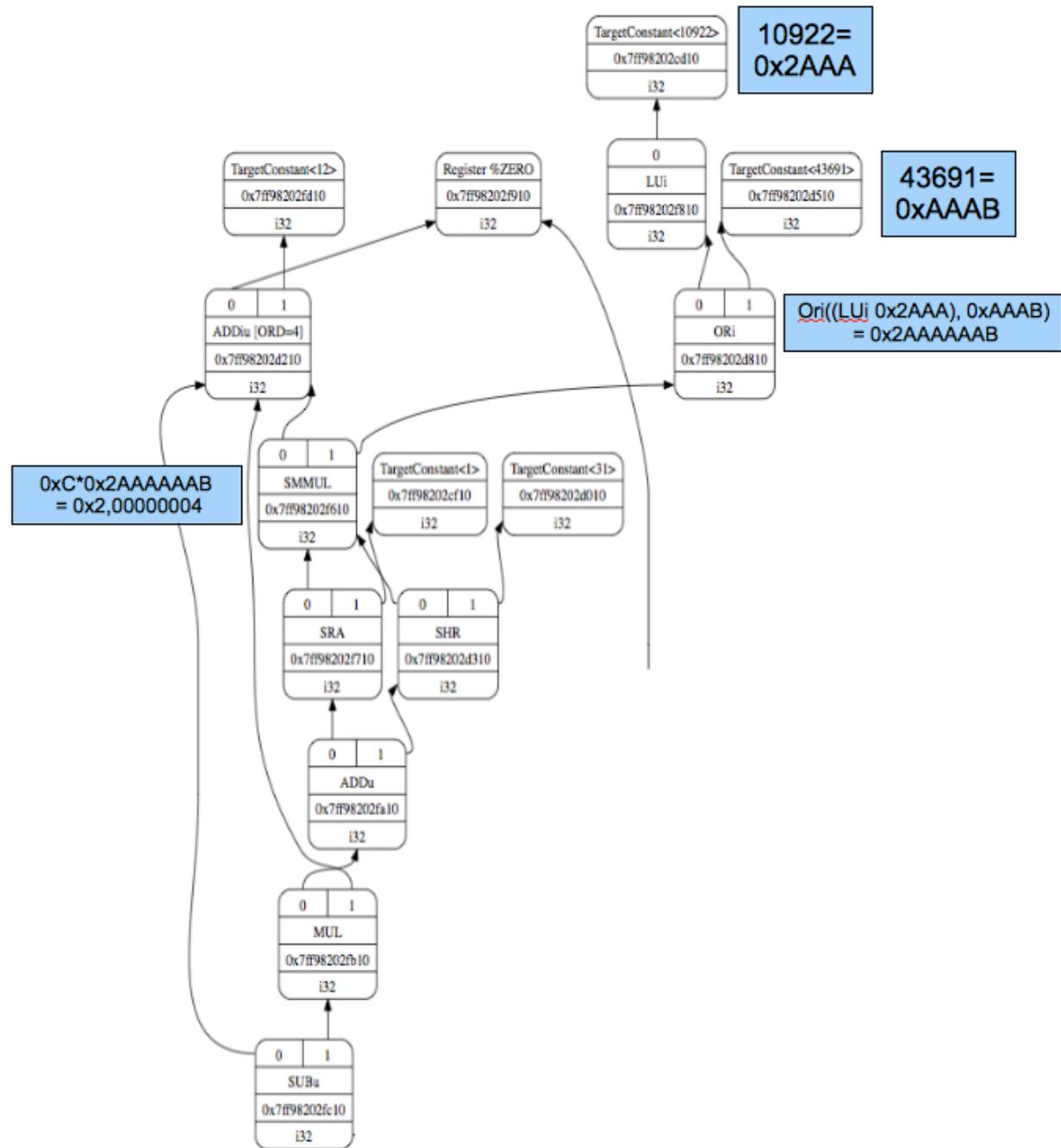


Fig. 4.3: DAG for ch4_1_mult.bc with ARM style SMMUL

- **HI, LO registers** in Chapter4_1/Cpu0RegisterInfo.td and Chapter4_1/MCTargetDesc/Cpu0BaseInfo.h
- **IIHiLo, IIImul** in Chapter4_1/Cpu0Schedule.td
- **SelectMULT()** in Chapter4_1/Cpu0ISelDAGToDAG.cpp

Except for custom types, LLVM IR operations of type **expand** and **promote** will call **Cpu0DAGToDAGISel::Select()** during instruction selection in the DAG translation process.

The function **selectMULT()**, which is called by **select()**, returns the **HI** part of the multiplication result to the **HI** register for IR operations **mulhs** or **mulhu**. After that, the **MFHI** instruction moves the **HI** register to the Cpu0 “**a**” register, **\$ra**.

Since the **MFHI** instruction follows the **FL** format and only utilizes the Cpu0 “**a**” register, we set **\$rb** and **imm16** to 0.

Fig. 4.4 and ch4_1_mult.cpu0.s show the compilation results of ch4_1_mult.bc.

```
118-165-66-82:input Jonathan$ cat ch4_1_mult.cpu0.s
```

```
...
# BB#0:
addiu $sp, $sp, -8
addiu $2, $zero, 11
st $2, 4($sp)
lui $2, 10922
ori $3, $2, 43691
addiu $2, $zero, 12
mult $2, $3
mfhi $3
shr $4, $3, 31
sra $3, $3, 1
addu $3, $3, $4
mul $3, $3, $2
subu $2, $2, $3
st $2, 4($sp)
addiu $sp, $sp, 8
ret $lr
```

Full Support for % and /

Attentive readers may notice that LLVM replaces **division** (`/`) with **multiplication** (`*`) when computing the **remainder** (`%`) in our example. This optimization occurs because the divisor in our example, “**(b+1) % 12**”, is a constant.

However, what happens if the divisor is a variable, such as in “**(b+1) % a**”? In this case, our current implementation would fail to handle it correctly.

Cpu0, like MIPS, uses the **LO** and **HI** registers to store the **quotient** and **remainder**, respectively. The instructions “**mflo**” and “**mfhi**” retrieve the results from the **LO** and **HI** registers.

Using this approach: - The operation `c = a / b` is implemented as:

```
div a, b
mflo c
```

- The operation `c = a % b` is implemented as:

```
div a, b
mfhi c
```

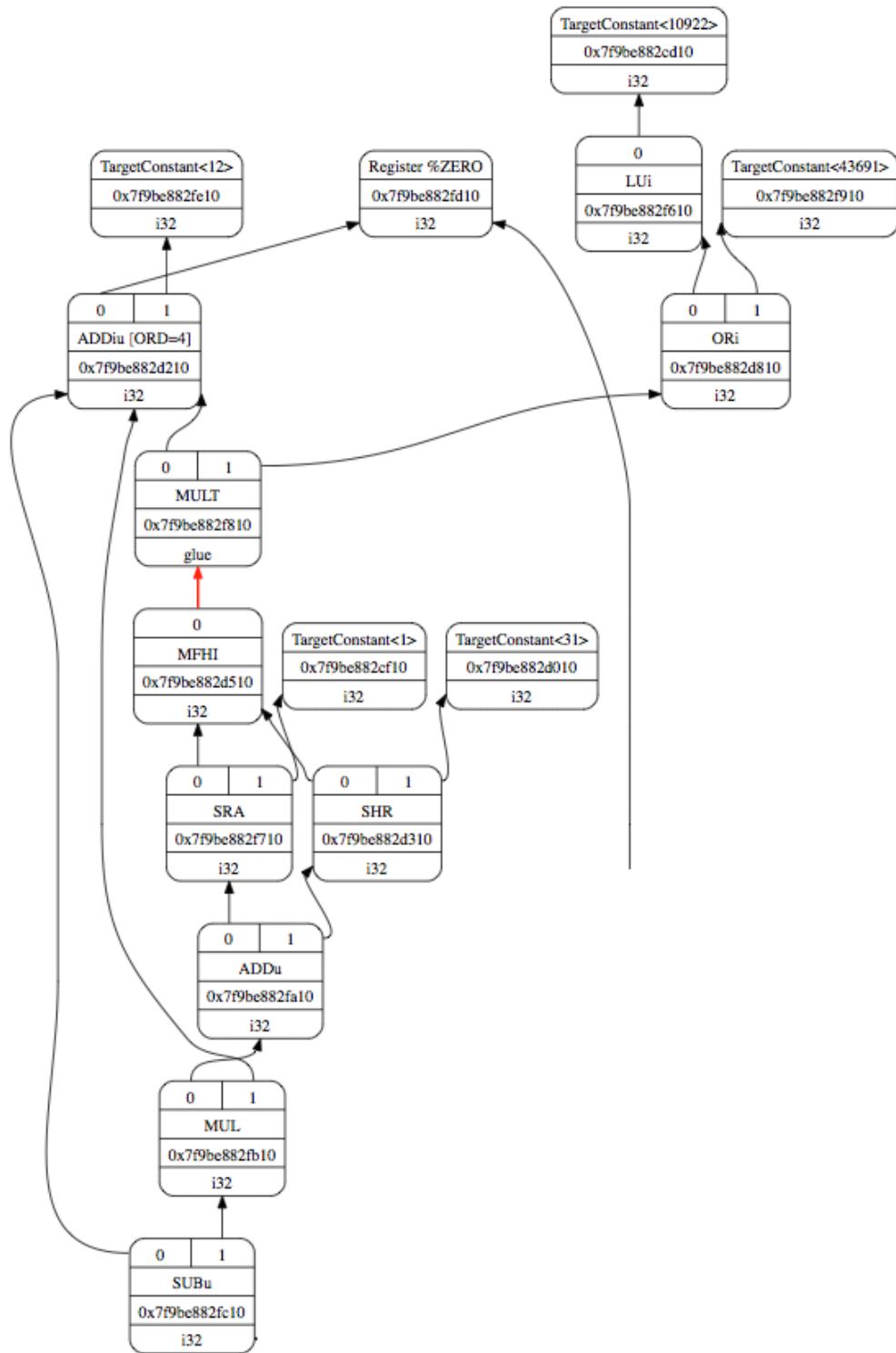


Fig. 4.4: DAG for ch4_1_mult.bc with Mips style MULT

To support the operators `%` and `/`, the following changes were added in **Chapter4_1/**:

1. **SDIV**, **UDIV**, and their reference classes, as well as DAG nodes in *Cpu0InstrInfo.td*.
2. **copyPhysReg()**, declared in *Cpu0InstrInfo.h* and implemented in *Cpu0InstrInfo.cpp*.
3. **setOperationAction(ISD::SDIV, MVT::i32, Expand)**, **setTargetDAGCombine(ISD::SDIVREM)** in the constructor of *Cpu0ISelLowering.cpp*, along with *PerformDivRemCombine()* and *PerformDAGCombine()* in *Cpu0ISelLowering.cpp*.

The LLVM IR instruction **sdiv** represents **signed division**, while **udiv** represents **unsigned division**.

Ibdex/input/ch4_1_mult2.cpp

```
int test_mult()
{
    int b = 11;
    int a = 12;

    b = (b+1)%a;

    return b;
}
```

When running *ch4_1_mult2.cpp*, the `div` instruction is not generated for the `%` operator. Instead, LLVM still replaces it with **multiplication** (*).

This happens because LLVM applies **Constant Propagation Optimization**, which optimizes expressions involving constants at compile time.

To force LLVM to generate a `div` instruction for `%`, we can use *ch4_1_mod.cpp*, which prevents LLVM from applying **Constant Propagation Optimization**.

Ibdex/input/ch4_1_mod.cpp

```
int test_mod()
{
    int b = 11;
    volatile int a = 12;

    b = (b+1)%a;

    return b;
}
```

```
118-165-77-79:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_mod.cpp -emit-llvm -o ch4_1_mod.bc
118-165-77-79:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm
ch4_1_mod.bc -o -
...
div $zero, $3, $2
mflo  $2
...
```

To explains how to work with “div”, let’s run *ch4_1_mod.cpp* with debug option as follows,

To understand how LLVM generates the `div` instruction, let's run *ch4_1_mod.cpp* with the debug option as follows:

```
118-165-83-58:input Jonathan$ clang -target mips-unknown-linux-gnu -c ch4_1_mod.cpp -I/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.8.sdk/usr/include/ -emit-llvm -o ch4_1_mod.bc
118-165-83-58:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch4_1_mod.bc -o -
...
===_Z8test_modi
Initial selection DAG: BB#0 '_Z8test_mod2i:'
SelectionDAG has 21 nodes:
...
0x2447448: <multiple use>
0x24470d0: <multiple use>
0x24471f8: i32 = Constant<1>

0x2447320: i32 = add 0x24470d0, 0x24471f8 [ORD=7]

0x2447448: <multiple use>
0x2447570: i32 = srem 0x2447320, 0x2447448 [ORD=9]

0x24468b8: <multiple use>
0x2446b08: <multiple use>
0x2448fc0: ch = store 0x2447448:1, 0x2447570, 0x24468b8, ...
0x2449210: i32 = Register %V0

0x2448fc0: <multiple use>
0x2449210: <multiple use>
0x2448fc0: <multiple use>
0x24468b8: <multiple use>
0x2446b08: <multiple use>
0x24490e8: i32,ch = load 0x2448fc0, 0x24468b8, 0x2446b08<LD4[%b]> [ORD=11]

0x2449338: ch,glue = CopyToReg 0x2448fc0, 0x2449210, 0x24490e8 [ORD=12]

0x2449338: <multiple use>
0x2449210: <multiple use>
0x2449338: <multiple use>
0x2449460: ch = Cpu0ISD::Ret 0x2449338, 0x2449210, 0x2449338:1 [ORD=12]

Replacing.1 0x24490e8: i32,ch = load 0x2448fc0, 0x24468b8, ...

With: 0x2447570: i32 = srem 0x2447320, 0x2447448 [ORD=9]
and 1 other values
...
Optimized lowered selection DAG: BB#0 '_Z8test_mod2i:'
...
0x2447570: i32 = srem 0x2447320, 0x2447448 [ORD=9]
...
```

(continues on next page)

(continued from previous page)

```
Type-legalized selection DAG: BB#0 '_Z8test_mod2i:'
SelectionDAG has 16 nodes:
...
0x7fed6882d610: i32,ch = load 0x7fed6882d210, 0x7fed6882cd10,
0x7fed6882cb10<LD4[%1]> [ORD=5] [ID=-3]

0x7fed6882d810: i32 = Constant<12> [ID=-3]

0x7fed6882d610: <multiple use>
0x7fed6882d710: i32 = srem 0x7fed6882d810, 0x7fed6882d610 [ORD=6] [ID=-3]
...
Legalized selection DAG: BB#0 '_Z8test_mod2i:'
...
... i32 = srem 0x2447320, 0x2447448 [ORD=9] [ID=-3]
...
... replacing: ...: i32 = srem 0x2447320, 0x2447448 [ORD=9] [ID=13]
with:       ...: i32,i32 = sdivrem 0x2447320, 0x2447448 [ORD=9]

Optimized legalized selection DAG: BB#0 '_Z8test_mod2i:'
SelectionDAG has 18 nodes:
...
0x2449588: i32 = Register %HI

0x24470d0: <multiple use>
0x24471f8: i32 = Constant<1> [ID=6]

0x2447320: i32 = add 0x24470d0, 0x24471f8 [ORD=7] [ID=12]

0x2447448: <multiple use>
0x24490e8: glue = Cpu0ISD::DivRem 0x2447320, 0x2447448 [ORD=9]

0x24496b0: i32,ch,glue = CopyFromReg 0x240d480, 0x2449588, 0x24490e8 [ORD=9]

0x2449338: <multiple use>
0x2449210: <multiple use>
0x2449338: <multiple use>
0x2449460: ch = Cpu0ISD::Ret 0x2449338, 0x2449210, ...
...
===== Instruction selection begins: BB#0 ''
...
Selecting: 0x24490e8: glue = Cpu0ISD::DivRem 0x2447320, 0x2447448 [ORD=9] [ID=14]

ISEL: Starting pattern match on root node: 0x24490e8: glue = Cpu0ISD::DivRem
0x2447320, 0x2447448 [ORD=9] [ID=14]

Initial Opcode index to 4044
Morphed node: 0x24490e8: i32,glue = SDIV 0x2447320, 0x2447448 [ORD=9]

ISEL: Match complete!
```

(continues on next page)

(continued from previous page)

```
=> 0x24490e8: i32,glue = SDIV 0x2447320, 0x2447448 [ORD=9]
...
```

Summary of DAG Translation Steps:

The translation of DAGs for the ` `%` operator follows these four steps:

1. Reduce DAG Nodes

- This occurs in the “Optimized Lowered Selection DAG” stage.
- Redundant store and load nodes in SSA form are removed.

2. Convert `srem` to `sdivrem`

- This transformation happens in the “Legalized Selection DAG” stage.

3. Convert `sdivrem` to `Cpu0ISD::DivRem`

- This occurs in the “Optimized Legalized Selection DAG” stage.

4. Add Register Mapping for HI Register

- In the “Optimized Legalized Selection DAG” stage, the following DAG nodes are added: - “*i32 = Register %HI*” - “*CopyFromReg ...*”

For a detailed breakdown, refer to:

- **Table: Stages for C Operator ` %`**
- **Table: Functions Handling DAG Translation and Pattern Matching for C Operator ` %`**

Table 4.3: Stages for C operator %

Stage	IR/DAG/instruction
.bc	srem
Legalized selection DAG	sdivrem
Optimized legalized selection DAG pattern match	Cpu0ISD::DivRem, CopyFromReg xx, Hi, Cpu0ISD::DivRem div, mfhi

Table 4.4: Functions handle the DAG translation and pattern match for C operator %

Translation	Do by
srem => sdivrem	setOperationAction(ISD::SREM, MVT::i32, Expand);
sdivrem => Cpu0ISD::DivRem	setTargetDAGCombine(ISD::SDIVREM);
sdivrem => CopyFromReg xx, Hi, xx	PerformDivRemCombine();
Cpu0ISD::DivRem => div	SDIV (Cpu0InstrInfo.td)
CopyFromReg xx, Hi, xx => mfhi	MFLO (Cpu0InstrInfo.td)

The more detailed transformation of DAGs during *llc* execution follows these steps:

2. Convert `srem` to `sdivrem`

- Triggered by the code: ` setOperationAction(ISD::SREM, MVT::i32, Expand);`
- Defined in *Cpu0ISelLowering.cpp*.

- For details on **Expand**, refer to⁷ and⁸.

3. Convert `sdivrem` to `Cpu0ISD::DivRem`

- Triggered by: ` setTargetDAGCombine (ISD::SDIVREM) ;`
- Also defined in *Cpu0ISelLowering.cpp*.

4. Handle `CopyFromReg` in DAG

- Managed by *PerformDivRemCombine()*, which is called by *performDAGCombine()*.
- The `%` operator (corresponding to *srem*) makes “*N->hasAnyUseOfValue(1)*” true in *PerformDivRemCombine()*, resulting in “*CopyFromReg*” DAG creation.
- The `/` operator makes “*N->hasAnyUseOfValue(0)*” true.
- For *sdivrem*:
 - *sdiv* sets “*N->hasAnyUseOfValue(0)*” true.
 - *srem* sets “*N->hasAnyUseOfValue(1)*” true.

Once these steps modify the DAGs during *llc* execution, pattern matching in *Chapter4_1/Cpu0InstrInfo.td* translates:

- `Cpu0ISD::DivRem` → `div`
- `CopyFromReg xxDAG, Register %H, Cpu0ISD::DivRem` → `mfhi`

The *ch4_1_div.cpp* file tests the */*(division) operator.

Rotate Instructions

Chapter4_1 includes support for **rotate operations**. The instructions `rol`, `ror`, `rolv`, and `rорv` are defined in *Cpu0InstrInfo.td* for translation.

Compiling *ch4_1_rotate.cpp* will generate the *Cpu0 rol* instruction.

Ibdex/input/ch4_1_rotate.cpp

```
// #define TEST_ROXV

int test_rotate_left()
{
    unsigned int a = 8;
    int result = ((a << 30) | (a >> 2));

    return result;
}

#ifndef TEST_ROXV

int test_rotate_left1()
{
    volatile unsigned int a = 4;
    volatile int n = 30;
    int result = ((a << n) | (a >> (32 - n)));
}
```

(continues on next page)

⁷ <http://llvm.org/docs/WritingAnLLVMBackend.html#expand>

⁸ <http://llvm.org/docs/CodeGenerator.html#selectiondag-legalizetypes-phase>

(continued from previous page)

```

    return result;
}

int test_rotate_right()
{
    volatile unsigned int a = 1;
    volatile int n = 30;
    int result = ((a >> n) | (a << (32 - n)));

    return result;
}

#endif

```

```

114-43-200-122:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_rotate.cpp -emit-llvm -o ch4_1_rotate.bc
114-43-200-122:input Jonathan$ llvm-dis ch4_1_rotate.bc -o -

```

```

#define i32 @_Z16test_rotate_leftv() #0 {
    %a = alloca i32, align 4
    %result = alloca i32, align 4
    store i32 8, i32* %a, align 4
    %1 = load i32* %a, align 4
    %2 = shl i32 %1, 30
    %3 = load i32* %a, align 4
    %4 = ashr i32 %3, 2
    %5 = or i32 %2, %4
    store i32 %5, i32* %result, align 4
    %6 = load i32* %result, align 4
    ret i32 %6
}

```

```

114-43-200-122:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_1_rotate.bc -o -
...
rol $2, $2, 30
...

```

4.2 Logical Instructions

Chapter4_2 introduces support for logical operators:

`&, |, ^, !=, !=, <, <=, >, >=`

These operations are straightforward to implement.

Below, you'll find:

- The added code with comments
- A table mapping **IR operations → DAG nodes → final instructions**
- The execution results of **bitcode (bc)** and **assembly (asm)** for `ch4_2_logic.cpp`

Please check the run results to verify the implementation.

Ibdex/chapters/Chapter4_2/Cpu0InstrInfo.td

```
let Predicates = [Ch4_2] in {
class CmpInstr<bits<8> op, string instr_asm,
    InstrItinClass itin, RegisterClass RC, RegisterClass RD,
    bit isComm = 0>:
FA<op, (outs RD:$ra), (ins RC:$rb, RC:$rc),
!strconcat(instr_asm, "\t$ra, $rb, $rc"), [], itin> {
let shamt = 0;
let isCommutable = isComm;
let Predicates = [HasCmp];
}
```

```
// Logical
class LogicNOR<bits<8> op, string instr_asm, RegisterClass RC>:
FA<op, (outs RC:$ra), (ins RC:$rb, RC:$rc),
!strconcat(instr_asm, "\t$ra, $rb, $rc"),
[(set RC:$ra, (not (or RC:$rb, RC:$rc)))], IIAlu> {
let shamt = 0;
let isCommutable = 1;
}
```

```
// SetCC
let Predicates = [Ch4_2] in {
class SetCC_R<bits<8> op, string instr_asm, PatFrag cond_op,
    RegisterClass RC>:
FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
!strconcat(instr_asm, "\t$ra, $rb, $rc"),
[(set GPROut:$ra, (cond_op RC:$rb, RC:$rc))],
IIAlu>, Requires<[HasSlt]> {
let shamt = 0;
}

class SetCC_I<bits<8> op, string instr_asm, PatFrag cond_op, Operand Od,
    PatLeaf imm_type, RegisterClass RC>:
FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
!strconcat(instr_asm, "\t$ra, $rb, $imm16"),
[(set GPROut:$ra, (cond_op RC:$rb, imm_type:$imm16))],
IIAlu>, Requires<[HasSlt]> {
}
}
```

```
let Predicates = [Ch4_2] in {
def ANDi      : ArithLogicI<0x0c, "andi", and, uimm16, immZExt16, CPURegs>;
}
```

```
let Predicates = [Ch4_2] in {
def XORi      : ArithLogicI<0x0e, "xori", xor, uimm16, immZExt16, CPURegs>;
}
```

```
let Predicates = [Ch4_2] in {
let Predicates = [HasCmp] in {
def CMP      : CmpInstr<0x2A, "cmp", IIAlu, CPURegs, SR, 0>;
def CMPu    : CmpInstr<0x2B, "cmpl", IIAlu, CPURegs, SR, 0>;
}
}
```

```
let Predicates = [Ch4_2] in {
def AND     : ArithLogicR<0x18, "and", and, IIAlu, CPURegs, 1>;
def OR      : ArithLogicR<0x19, "or", or, IIAlu, CPURegs, 1>;
def XOR     : ArithLogicR<0x1a, "xor", xor, IIAlu, CPURegs, 1>;
def NOR     : LogicNOR<0x1b, "nor", CPURegs>;
}
```

```
let Predicates = [Ch4_2] in {
let Predicates = [HasSlt] in {
def SLTi    : SetCC_I<0x26, "slti", setlt, simm16, immSExt16, CPURegs>;
def SLTiU   : SetCC_I<0x27, "sltiu", setult, simm16, immSExt16, CPURegs>;
def SLT     : SetCC_R<0x28, "slt", setlt, CPURegs>;
def SLTu    : SetCC_R<0x29, "sltu", setult, CPURegs>;
}
}
```

```
let Predicates = [Ch4_2] in {
def : Pat<(not CPURegs:$in),
// 1's complement of in, ex. not(0xf000000f) == 0xffffffff
(NOR CPURegs:$in, ZERO)>;
}
```

```
// setcc patterns

let Predicates = [Ch4_2] in {
// setcc for cmp instruction
multiclass SeteqPatsCmp<RegisterClass RC> {
// a == b
def : Pat<(seteq RC:$lhs, RC:$rhs),
(SHR (ANDi (CMP RC:$lhs, RC:$rhs), 2), 1)>;
// a != b
def : Pat<(setne RC:$lhs, RC:$rhs),
(XORi (SHR (ANDi (CMP RC:$lhs, RC:$rhs), 2), 1), 1)>;
}

// a < b
multiclass SetltPatsCmp<RegisterClass RC> {
def : Pat<(setlt RC:$lhs, RC:$rhs),
(ANDi (CMP RC:$lhs, RC:$rhs), 1)>;
// if cpu0 `define N `SW[31] instead of `SW[0] // Negative flag, then need
// 2 more instructions as follows,
// (XORi (ANDi (SHR (CMP RC:$lhs, RC:$rhs), (LUI 0x8000), 31), 1), 1)>;
def : Pat<(setult RC:$lhs, RC:$rhs),
(ANDi (CMPl RC:$lhs, RC:$rhs), 1)>;
}
```

(continues on next page)

(continued from previous page)

```

// a <= b
multiclass SetlePatsCmp<RegisterClass RC> {
    def : Pat<(setle RC:$lhs, RC:$rhs),
    // a <= b is equal to (XORi (b < a), 1)
        (XORi (ANDi (CMP RC:$rhs, RC:$lhs), 1), 1)>;
    def : Pat<(setule RC:$lhs, RC:$rhs),
        (XORi (ANDi (CMP RC:$rhs, RC:$lhs), 1), 1)>;
}

// a > b
multiclass SetgtPatsCmp<RegisterClass RC> {
    def : Pat<(setgt RC:$lhs, RC:$rhs),
    // a > b is equal to b < a is equal to setlt(b, a)
        (ANDi (CMP RC:$rhs, RC:$lhs), 1)>;
    def : Pat<(setugt RC:$lhs, RC:$rhs),
        (ANDi (CMPu RC:$rhs, RC:$lhs), 1)>;
}

// a >= b
multiclass SetgePatsCmp<RegisterClass RC> {
    def : Pat<(setge RC:$lhs, RC:$rhs),
    // a >= b is equal to b <= a
        (XORi (ANDi (CMP RC:$lhs, RC:$rhs), 1), 1)>;
    def : Pat<(setuge RC:$lhs, RC:$rhs),
        (XORi (ANDi (CMPu RC:$rhs, RC:$lhs), 1), 1)>;
}

// setcc for slt instruction
multiclass SeteqPatsSlt<RegisterClass RC, Instruction SLTiOp, Instruction XOROp,
                           Instruction SLTuOp, Register ZEROReg> {
    // a == b
    def : Pat<(seteq RC:$lhs, RC:$rhs),
        (SLTiOp (XOROp RC:$lhs, RC:$rhs), 1)>;
    // a != b
    def : Pat<(setne RC:$lhs, RC:$rhs),
        (SLTuOp ZEROReg, (XOROp RC:$lhs, RC:$rhs))>;
}

// a <= b
multiclass SetlePatsSlt<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
    def : Pat<(setle RC:$lhs, RC:$rhs),
    // a <= b is equal to (XORi (b < a), 1)
        (XORi (SLTOp RC:$rhs, RC:$lhs), 1)>;
    def : Pat<(setule RC:$lhs, RC:$rhs),
        (XORi (SLTuOp RC:$rhs, RC:$lhs), 1)>;
}

// a > b
multiclass SetgtPatsSlt<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
    def : Pat<(setgt RC:$lhs, RC:$rhs),
    // a > b is equal to b < a is equal to setlt(b, a)

```

(continues on next page)

(continued from previous page)

```

        (SLTOp RC:$rhs, RC:$lhs)>;
def : Pat<(setugt RC:$lhs, RC:$rhs),
          (SLTuOp RC:$rhs, RC:$lhs)>;
}

// a >= b
multiclass SetgePatsSlt<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
    def : Pat<(setge RC:$lhs, RC:$rhs),
// a >= b is equal to b <= a
          (XORi (SLTOp RC:$lhs, RC:$rhs), 1)>;
    def : Pat<(setuge RC:$lhs, RC:$rhs),
          (XORi (SLTuOp RC:$lhs, RC:$rhs), 1)>;
}

multiclass SetgeImmPatsSlt<RegisterClass RC, Instruction SLTiOp,
                           Instruction SLTiuOp> {
    def : Pat<(setge RC:$lhs, immSExt16:$rhs),
          (XORi (SLTiOp RC:$lhs, immSExt16:$rhs), 1)>;
    def : Pat<(setuge RC:$lhs, immSExt16:$rhs),
          (XORi (SLTiuOp RC:$lhs, immSExt16:$rhs), 1)>;
}

let Predicates = [HasSlt] in {
defm : SeteqPatsSlt<CPUREgs, SLTiu, XOR, SLTu, ZERO>;
defm : SetlePatsSlt<CPUREgs, SLT, SLTu>;
defm : SetgtPatsSlt<CPUREgs, SLT, SLTu>;
defm : SetgePatsSlt<CPUREgs, SLT, SLTu>;
defm : SetgeImmPatsSlt<CPUREgs, SLTi, SLTiu>;
}

let Predicates = [HasCmp] in {
defm : SeteqPatsCmp<CPUREgs>;
defm : SetltPatsCmp<CPUREgs>;
defm : SetlePatsCmp<CPUREgs>;
defm : SetgtPatsCmp<CPUREgs>;
defm : SetgePatsCmp<CPUREgs>;
}
} // let Predicates = [Ch4_2]

```

Ibdex/chapters/Chapter4_2/Cpu0ISelLowering.cpp

```

Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
: TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {

```

```

// Cpu0 doesn't have sext_inreg, replace them with shl/sra.
setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i1, Expand);
setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i8, Expand);
setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i16, Expand);
setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i32, Expand);

```

(continues on next page)

(continued from previous page)

```
setOperationAction(Instruction::SIGN_EXTEND_INREG, MVT::Other, Expand);
```

```
...
```

Ibdex/input/ch4_2_logic.cpp

```
int test_andorxornotcomplement()
{
    int a = 5;
    int b = 3;
    int c = 0, d = 0, e = 0, f = 0, g = 0;

    c = (a & b); // c = 1
    d = (a | b); // d = 7
    e = (a ^ b); // e = 6
    b = !a; // b = 0
    g = ~f; // 1's complement, ~0=(-1)=0xffffffff

    return (c+d+e+b+g); // 13
}

int test_setxx()
{
    int a = 5;
    int b = 3;
    int c, d, e, f, g, h;

    c = (a == b); // seq, c = 0
    d = (a != b); // sne, d = 1
    e = (a < b); // slt, e = 0
    f = (a <= b); // sle, f = 0
    g = (a > b); // sgt, g = 1
    h = (a >= b); // sge, g = 1

    return (c+d+e+f+g+h); // 3
}
```

```
114-43-204-152:input Jonathan$ clang -target mips-unknown-linux-gnu -c ch4_2_logic.cpp -emit-llvm -o ch4_2_logic.bc
```

```
114-43-204-152:input Jonathan$ llvm-dis ch4_2_logic.bc -o -
```

```
...
; Function Attrs: nounwind uwtable
define i32 @_Z16test_andorxornotv() #0 {
entry:
...
%and = and i32 %0, %1
...
%or = or i32 %2, %3
...
```

(continues on next page)

(continued from previous page)

```
%xor = xor i32 %4, %5
...
%tobool = icmp ne i32 %6, 0
%lnot = xor i1 %tobool, true
%conv = zext i1 %lnot to i32
...
}

; Function Attrs: nounwind uwtable
define i32 @_Z10test_setxxv() #0 {
entry:
...
%cmp = icmp eq i32 %0, %1
%conv = zext i1 %cmp to i32
store i32 %conv, i32* %c, align 4
...
%cmp1 = icmp ne i32 %2, %3
%conv2 = zext i1 %cmp1 to i32
store i32 %conv2, i32* %d, align 4
...
%cmp3 = icmp slt i32 %4, %5
%conv4 = zext i1 %cmp3 to i32
store i32 %conv4, i32* %e, align 4
...
%cmp5 = icmp sle i32 %6, %7
%conv6 = zext i1 %cmp5 to i32
store i32 %conv6, i32* %f, align 4
...
%cmp7 = icmp sgt i32 %8, %9
%conv8 = zext i1 %cmp7 to i32
store i32 %conv8, i32* %g, align 4
...
%cmp9 = icmp sge i32 %10, %11
%conv10 = zext i1 %cmp9 to i32
store i32 %conv10, i32* %h, align 4
...
}

114-43-204-152:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch4_2_logic.bc -o -

.globl _Z16test_andorxornotv
...
and $3, $4, $3
...
or $3, $4, $3
...
xor $3, $4, $3
...
cmp $sw, $3, $2
andi $2, $sw, 2
```

(continues on next page)

(continued from previous page)

```
shr $2, $2, 1
...
.globl _Z10test_setxxv
...
cmp $sw, $3, $2
andi $2, $sw, 2
shr $2, $2, 1
...
cmp $sw, $3, $2
andi $2, $sw, 2
shr $2, $2, 1
xori $2, $2, 1
...
cmp $sw, $3, $2
andi $2, $sw, 1
...
cmp $sw, $3, $2
andi $2, $sw, 1
xori $2, $2, 1
...
cmp $sw, $3, $2
andi $2, $sw, 1
...
cmp $sw, $3, $2
andi $2, $sw, 1
xori $2, $2, 1
...
```

```
114-43-204-152:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
ch4_2_logic.bc -o -
...
    sltiu    $2, $2, 1
    andi    $2, $2, 1
...

```

Table 4.5: Logic operators for cpu032I

C	.bc	Optimized legalized selection DAG	cpu032I
&, &&	and	and	and
,	or	or	or
^	xor	xor	xor
!	<ul style="list-style-type: none"> %tobool = icmp ne i32 %6, 0 %lnot = xor i1 %tobool, true %conv = zext i1 %lnot to i32 	<ul style="list-style-type: none"> %lnot = (setcc %tobool, 0, seteq) %conv = (and %lnot, 1) 	<ul style="list-style-type: none"> xor \$3, \$4, \$3
==	<ul style="list-style-type: none"> %cmp = icmp eq i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, seteq) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 shr \$2, \$2, 1 andi \$2, \$2, 1
!=	<ul style="list-style-type: none"> %cmp = icmp ne i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, setne) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 shr \$2, \$2, 1 andi \$2, \$2, 1
<	<ul style="list-style-type: none"> %cmp = icmp lt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setlt) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 andi \$2, \$2, 1 andi \$2, \$2, 1
<=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setle) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$2, \$3 andi \$2, \$sw, 1 xori \$2, \$2, 1 andi \$2, \$2, 1
>	<ul style="list-style-type: none"> %cmp = icmp gt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setgt) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$2, \$3 andi \$2, \$sw, 2 andi \$2, \$2, 1
>=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setle) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 1 xori \$2, \$2, 1 andi \$2, \$2, 1

Table 4.6: Logic operators for cpu032II

C	.bc	Optimized legalized selection DAG	cpu032II
&, &&	and	and	and
l,	or	or	or
^	xor	xor	xor
!	<ul style="list-style-type: none"> %tobool = icmp ne i32 %6, 0 %lnot = xor i1 %tobool, true %conv = zext i1 %lnot to i32 	<ul style="list-style-type: none"> %lnot = (setcc %tobool, 0, seteq) %conv = (and %lnot, 1) 	<ul style="list-style-type: none"> xor \$3, \$4, \$3
==	<ul style="list-style-type: none"> %cmp = icmp eq i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, seteq) and %cmp, 1 	<ul style="list-style-type: none"> xor \$2, \$3, \$2 sliu \$2, \$2, 1 andi \$2, \$2, 1
!=	<ul style="list-style-type: none"> %cmp = icmp ne i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, setne) and %cmp, 1 	<ul style="list-style-type: none"> xor \$2, \$3, \$2 sliu \$2, \$zero, 2 shr \$2, \$2, 1 andi \$2, \$2, 1
<	<ul style="list-style-type: none"> %cmp = icmp lt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setlt) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 andi \$2, \$2, 1
<=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setle) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 xori \$2, \$2, 1 andi \$2, \$2, 1
>	<ul style="list-style-type: none"> %cmp = icmp gt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setgt) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 andi \$2, \$2, 1
>=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setle) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 xori \$2, \$2, 1 andi \$2, \$2, 1

For `ch4_2_logic.cpp`, the relation operators such as `==, !=, <, <=, >, >=` follow the convention where:

- %0 = \$3 = 5
- %1 = \$2 = 3

Optimized Legalized Selection DAG

The “**Optimized Legalized Selection DAG**” is the final DAG stage before **instruction selection**, as mentioned earlier in this chapter. To view all DAG stages, use the command:

`lli -debug`

`### slt vs. cmp`

From the results, `slt` (set-less-than) requires fewer instructions than `cmp` for relation operator translation.

Additionally:

- `slt` operates using general-purpose registers.
- `cmp` requires the dedicated `\$sw` register.

This difference makes `slt` a more efficient choice in certain scenarios.

Ibdex/input/ch4_2_slt_explain.cpp

```
int test_OptSlt()
{
    int a = 3, b = 1;
```

(continues on next page)

(continued from previous page)

```

int d = 0, e = 0, f = 0;

d = (a < 1);
e = (b < 2);
f = d + e;

return (f);
}

```

```

118-165-78-10:input Jonathan$ clang -target mips-unknown-linux-gnu -O2
-c ch4_2_slt_explain.cpp -emit-llvm -o ch4_2_slt_explain.bc
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch4_2_slt_explain.bc -o -
...
ld $3, 20($sp)
cmp $sw, $3, $2
andi $2, $sw, 1
andi $2, $2, 1
st $2, 12($sp)
addiu $2, $zero, 2
ld $3, 16($sp)
cmp $sw, $3, $2
andi $2, $sw, 1
andi $2, $2, 1
...
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=static -filetype=asm
ch4_2_slt_explain.bc -o -
...
ld $2, 20($sp)
slti $2, $2, 1
andi $2, $2, 1
st $2, 12($sp)
ld $2, 16($sp)
slti $2, $2, 2
andi $2, $2, 1
st $2, 8($sp)
...

```

Run the following two *llc -mcpu* options with *ch4_2_slt_explain.cpp* to obtain the results discussed above.

Instruction Hazard in *llc -mcpu=cpu032I*

Regardless of the move operation between `\$\$sw` and general-purpose registers in *llc -mcpu=cpu032I*, the two `cmp` instructions introduce a hazard during instruction reordering. This occurs because both instructions rely on the `\$\$sw` register.

Avoiding Hazards with *llc -mcpu=cpu032II*

The *llc -mcpu=cpu032II* configuration avoids this issue by using `slti` (set-less-than immediate)⁹.

Reordering Optimization with *slti*

⁹ See book Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

The `slti` version allows safer instruction reordering, as demonstrated below:

```
...
ld $2, 16($sp)
slti $2, $2, 2
andi $2, $2, 1
st $2, 8($sp)
ld $2, 20($sp)
slti $2, $2, 1
andi $2, $2, 1
st $2, 12($sp)
...
```

Chapter 4.2 includes both `cmp` and `slt` instructions. Although `cpu032II` supports both instructions, `slt` takes priority because the directive:

let Predicates = [HasSlt]

appears **before**:

let Predicates = [HasCmp]

in *Cpu0InstrInfo.td*.

4.3 Summary

The following table summarizes the **C operators**, their corresponding **LLVM IR (.bc)**, **Optimized Legalized Selection DAG**, and **Cpu0 instructions** implemented in this chapter.

This chapter covers over **20 mathematical and logical operators**, spanning **approximately 400 lines** of source code.

Table 4.7: Chapter 4 mathematic operators

C	.bc	Optimized legalized selection DAG	Cpu0
+	add	add	addu
-	sub	sub	subu
*	mul	mul	mul
/	sdiv	Cpu0ISD::DivRem	div
•	udiv	Cpu0ISD::DivRemU	divu
<<	shl	shl	shl
>>	<ul style="list-style-type: none"> • ashr • lshr 	<ul style="list-style-type: none"> • sra • srl 	<ul style="list-style-type: none"> • sra • shr
!	<ul style="list-style-type: none"> • %tobool = icmp ne i32 %0, 0 • %lnot = xor i1 %tobool, true 	<ul style="list-style-type: none"> • %lnot = (setcc %tobool, 0, seteq) • %conv = (and %lnot, 1) 	<ul style="list-style-type: none"> • %1 = (xor %tobool, 0) • %true = (addiu \$r0, 1) • %lnot = (xor %1, %true)
•	<ul style="list-style-type: none"> • %conv = zext i1 %lnot to i32 	<ul style="list-style-type: none"> • %conv = (and %lnot, 1) 	<ul style="list-style-type: none"> • %conv = (and %lnot, 1)
%	<ul style="list-style-type: none"> • srem • sremu 	<ul style="list-style-type: none"> • Cpu0ISD::DivRem • Cpu0ISD::DivRemU 	<ul style="list-style-type: none"> • div • divu
(x<<n)!(x>>32-n)	shl + lshr	rotl, rotr	rol, rolv, ror, rorv

GENERATING OBJECT FILES

- *Translate into obj file*
- *ELF obj related code*
- *Work flow*
- *Backend Target Registration Structure*

The previous chapters focused solely on **assembly code generation**. This chapter extends that by **adding ELF object file support** and verifying the generated object files using the *objdump* utility.

With LLVM support, the **Cpu0 backend** can generate both **big-endian** and **little-endian** object files with minimal additional code.

Additionally, this chapter introduces the **Target Registration mechanism** and its structure.

Similar to Fig. 3.7 from the previous chapter *Backend structure*, but this chapter focuses on emitting **binary object instructions** as shown in Fig. 5.1.



Fig. 5.1: When “llc -filetype=obj”, Cpu0AsmPrinter extract MCInst from MachineInstr for obj encoding

5.1 Translate into obj file

Currently, the backend **only supports translating LLVM IR code into assembly code**. If you attempt to compile an LLVM IR input file into an **object file** using the *Chapter4_2/* compiler code, you will encounter the following error message:

```
bin$ pwd
$HOME/llvm/test/build/bin/
```

(continues on next page)

(continued from previous page)

```
bin$ llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1_math_math.bc -o
ch4_1_math.cpu0.o
~/llvm/test/build/bin/llc: target does not
support generation of this file type!
```

The *Chapter5_1/* implementation **supports object file generation**. It can produce object files for both **big-endian** and **little-endian** architectures using the following commands:

- **Big-endian:** `llc -march=cpu0`
- **Little-endian:** `llc -march=cpu0el`

Running these commands will generate the corresponding object files as shown below:

```
input$ cat ch4_1_math.cpu0.s
...
.set nomacro
# BB#0:                                # %entry
    addiu $sp, $sp, -40
$tmp1:
    .cfi_def_cfa_offset 40
    addiu $2, $zero, 5
    st $2, 36($fp)
    addiu $2, $zero, 2
    st $2, 32($fp)
    addiu $2, $zero, 0
    st $2, 28($fp)
...
bin$ pwd
$HOME/llvm/test/build/bin/
bin$ llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1_math.bc -o
ch4_1_math.cpu0.o
input$ objdump -s ch4_1_math.cpu0.o

ch4_1_math.cpu0.o:      file format elf32-big

Contents of section .text:
0000 09ddfffc8 09200005 022d0034 09200002 ..... .-.4. ...
0010 022d0030 0920ffffb 022d002c 012d0030 .-.0. ....,-.-.0
0020 013d0034 11232000 022d0028 012d0030 .=.4.# ...-.(-.0
0030 013d0034 12232000 022d0024 012d0030 .=.4.# ...-$.-.0
0040 013d0034 17232000 022d0020 012d0034 .=.4.# ...-. .-.4
0050 1e220002 022d001c 012d002c 1e220001 ."....-.,."...
0060 022d000c 012d0034 1d220002 022d0018 .-....4."....-
0070 012d002c 1f22001e 022d0008 09200001 .-.,."....-.... .
0080 013d0034 21232000 023d0014 013d0030 .=.4!20..=....=0
0090 21223000 022d0004 09200080 013d0034 !"0..-.... .-.4
00a0 22223000 022d0010 012d0034 013d0030 ""0..-....-4.=.0
00b0 20232000 022d0000 09dd0038 3ce00000 # ..-....8<...

input$ ~/llvm/test/
build/bin/llc -march=cpu0el -relocation-model=pic -filetype=obj
ch4_1_math.bc -o ch4_1_math.cpu0el.o
```

(continues on next page)

(continued from previous page)

```
input$ objdump -s ch4_1_math.cpu0el.o

ch4_1_math.cpu0el.o:      file format elf32-little

Contents of section .text:
0000 c8ffdd09 05002009 34002d02 02002009 ..... 4.-... .
0010 30002d02 fbf02009 2c002d02 30002d01 0.-... ,.-.0.-.
0020 34003d01 00202311 28002d02 30002d01 4.=.. #.(.-.0.-.
0030 34003d01 00202312 24002d02 30002d01 4.=.. #.$.-.0.-.
0040 34003d01 00202317 20002d02 34002d01 4.=.. #. .-.4.-.
0050 0200221e 1c002d02 2c002d01 0100221e .."....,-,-....".
0060 0c002d02 34002d01 0200221d 18002d02 ..-.4.-...."-....".
0070 2c002d01 1e00221f 08002d02 01002009 ,,-...."-.... .
0080 34003d01 00303221 14003d02 30003d01 4.=..02!..=.0.=.
0090 00302221 04002d02 80002009 34003d01 .0"!..-.4.=.
00a0 00302222 10002d02 34002d01 30003d01 .0""..-.4.-.0.=.
00b0 00202320 00002d02 3800dd09 0000e03c . # .-.8.....<
```

The first instruction is “**addiu \$sp, -56**”, and its corresponding obj is 0x09ddffc8. The opcode of addiu is 0x09 (8 bits); the \$sp register number is 13 (0xd) (4 bits); and the immediate value is -56 (0xffffc8) (16 bits), so it is correct.

The third instruction “**st \$2, 52(\$fp)**”, and its corresponding obj is 0x022b0034. The st opcode is 0x02, \$2 is 0x2, \$fp is 0xb, and the immediate value is 52 (0x0034).

Thanks to the Cpu0 instruction format, where the opcode, register operand, and offset (immediate value) sizes are multiples of 4 bits, the obj format is easy to verify manually.

For big-endian format: (B0, B1, B2, B3) = (09, dd, ff, c8), objdump from B0 to B3 is 0x09ddffc8.

For little-endian format: (B3, B2, B1, B0) = (09, dd, ff, c8), objdump from B0 to B3 is 0xc8ffffd09.

5.2 ELF obj related code

To support ELF object file generation, the following code was modified and added to Chapter5_1.

Ibdex/chapters/Chapter5_1/InstPrinter/Cpu0InstPrinter.cpp

```
#include "MCTargetDesc/Cpu0MCExpr.h"
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/CMakeLists.txt

```
Cpu0AsmBackend.cpp
Cpu0MCCodeEmitter.cpp
Cpu0MCExpr.cpp
Cpu0ELFObjectWriter.cpp
Cpu0TargetStreamer.cpp
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0AsmBackend.h

```
===== Cpu0AsmBackend.h - Cpu0 Asm Backend =====/
//                                            The LLVM Compiler Infrastructure
```

(continues on next page)

(continued from previous page)

```
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// This file defines the Cpu0AsmBackend class.  
//  
//=====//  
//  
  
#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ASMBACKEND_H  
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ASMBACKEND_H  
  
#include "Cpu0Config.h"  
  
#include "MCTargetDesc/Cpu0FixupKinds.h"  
#include "llvm/ADT/Triple.h"  
#include "llvm/MC/MCAsmBackend.h"  
  
namespace llvm {  
  
class MCAssembler;  
struct MCFixupKindInfo;  
class Target;  
class MCObjectWriter;  
  
class Cpu0AsmBackend : public MCAsmBackend {  
    Triple TheTriple;  
  
public:  
    Cpu0AsmBackend(const Target &T, const Triple &TT)  
        : MCAsmBackend(TT.isLittleEndian() ? support::little : support::big),  
          TheTriple(TT) {}  
  
    std::unique_ptr<MCObjectTargetWriter>  
    createObjectTargetWriter() const override;  
  
    void applyFixup(const MCAssembler &Asm, const MCFixup &Fixup,  
                    const MCValue &Target, MutableArrayRef<char> Data,  
                    uint64_t Value, bool IsResolved,  
                    const MCSubtargetInfo *STI) const override;  
  
    const MCFixupKindInfo &getFixupKindInfo(MCFixupKind Kind) const override;  
  
    unsigned getNumFixupKinds() const override {  
        return Cpu0::NumTargetFixupKinds;  
    }  
  
    /// @name Target Relaxation Interfaces  
    /// @{
```

(continues on next page)

(continued from previous page)

```

/// MayNeedRelaxation - Check whether the given instruction may need
/// relaxation.
///
/// \param Inst - The instruction to test.
bool mayNeedRelaxation(const MCInst &Inst,
                       const MCSUBtargetInfo &STI) const override {
    return false;
}

/// fixupNeedsRelaxation - Target specific predicate for whether a given
/// fixup requires the associated instruction to be relaxed.
bool fixupNeedsRelaxation(const MCFixup &Fixup, uint64_t Value,
                           const MCRelaxableFragment *DF,
                           const MCAsmLayout &Layout) const override {
    // FIXME.
    llvm_unreachable("RelaxInstruction() unimplemented");
    return false;
}

/// @}

bool writeNopData(raw_ostream &OS, uint64_t Count) const override;
}; // class Cpu0AsmBackend

} // namespace

#endif

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0AsmBackend.cpp

```

//===== Cpu0AsmBackend.cpp - Cpu0 Asm Backend -----//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
//
// This file implements the Cpu0AsmBackend class.
//
//=====//
//

#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "MCTargetDesc/Cpu0AsmBackend.h"

#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/MC/MCAsmBackend.h"
#include "llvm/MC/MCAssembler.h"

```

(continues on next page)

(continued from previous page)

```
#include "llvm/MC/MCDirectives.h"
#include "llvm/MC/MCELFObjectWriter.h"
#include "llvm/MC/MCFixupKindInfo.h"
#include "llvm/MC/MCOObjectWriter.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

static cl::opt<bool> HasLLD(
    "has-lld",
    cl::init(false),
    cl::desc("CPU0: Has lld linker for Cpu0."),
    cl::Hidden);

//@adjustFixupValue {
// Prepare value for the target space for it
static unsigned adjustFixupValue(const MCFixup &Fixup, uint64_t Value,
                                 MCContext &Ctx) {

    unsigned Kind = Fixup.getKind();

    // Add/subtract and shift
    switch (Kind) {
    default:
        return 0;
    case FK_GPRel_4:
    case FK_Data_4:
    case Cpu0::fixup_Cpu0_L016:
        break;
    case Cpu0::fixup_Cpu0_HI16:
    case Cpu0::fixup_Cpu0_GOT:
        // Get the higher 16-bits. Also add 1 if bit 15 is 1.
        Value = (Value >> 16) & 0xffff;
        break;
    }

    return Value;
}
//@adjustFixupValue }

std::unique_ptr<MCObjectTargetWriter>
Cpu0AsmBackend::createObjectTargetWriter() const {
    return createCpu0ELFObjectWriter(TheTriple);
}

/// ApplyFixup - Apply the \p Value for given \p Fixup into the provided
/// data fragment, at the offset specified by the fixup and following the
/// fixup kind as appropriate.
void Cpu0AsmBackend::applyFixup(const MCAssembler &Asm, const MCFixup &Fixup,
```

(continues on next page)

(continued from previous page)

```

        const MCValue &Target,
        MutableArrayRef<char> Data, uint64_t Value,
        bool IsResolved,
        const MCSubtargetInfo *STI) const {
MCFixupKind Kind = Fixup.getKind();
MCContext &Ctx = Asm.getContext();
Value = adjustFixupValue(Fixup, Value, Ctx);

if (!Value)
    return; // Doesn't change encoding.

// Where do we start in the object
unsigned Offset = Fixup.getOffset();
// Number of bytes we need to fixup
unsigned NumBytes = (getFixupKindInfo(Kind).TargetSize + 7) / 8;
// Used to point to big endian bytes
unsigned FullSize;

switch ((unsigned)Kind) {
default:
    FullSize = 4;
    break;
}

// Grab current value, if any, from bits.
uint64_t CurVal = 0;

for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = TheTriple.isLittleEndian() ? i : (FullSize - 1 - i);
    CurVal |= (uint64_t)((uint8_t)Data[Offset + Idx]) << (i*8);
}

uint64_t Mask = ((uint64_t)(-1) >>
                  (64 - getFixupKindInfo(Kind).TargetSize));
CurVal |= Value & Mask;

// Write out the fixed up bytes back to the code/data bits.
for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = TheTriple.isLittleEndian() ? i : (FullSize - 1 - i);
    Data[Offset + Idx] = (uint8_t)((CurVal >> (i*8)) & 0xff);
}
}

// @getFixupKindInfo {
const MCFixupKindInfo &Cpu0AsmBackend::
getFixupKindInfo(MCFixupKind Kind) const {
    unsigned JSUBReloRec = 0;
    if (HasLLD) {
        JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel;
    }
    else {
        JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel | MCFixupKindInfo::FKF_Constant;
    }
}
}

```

(continues on next page)

(continued from previous page)

```
}

const static MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds] = {
    // This table *must* be in same the order of fixup_* kinds in
    // Cpu0FixupKinds.h.
    //
    // name                  offset  bits  flags
    { "fixup_Cpu0_32",      0,      32,   0 },
    { "fixup_Cpu0_HI16",    0,      16,   0 },
    { "fixup_Cpu0_LO16",    0,      16,   0 },
    { "fixup_Cpu0_GPREL16", 0,      16,   0 },
    { "fixup_Cpu0_GOT",     0,      16,   0 },
    { "fixup_Cpu0_GOT_HI16", 0,      16,   0 },
    { "fixup_Cpu0_GOT_LO16", 0,      16,   0 }
};

if (Kind < FirstTargetFixupKind)
    return MCAsmBackend::getFixupKindInfo(Kind);

assert(unsigned(Kind - FirstTargetFixupKind) < getNumFixupKinds() &&
       "Invalid kind!");
return Infos[Kind - FirstTargetFixupKind];
}
//@getFixupKindInfo }

/// WriteNopData - Write an (optimal) nop sequence of Count bytes
/// to the given output. If the target cannot generate such a sequence,
/// it should return an error.
///
/// \return - True on success.
bool Cpu0AsmBackend::writeNopData(raw_ostream &OS, uint64_t Count) const {
    return true;
}

// MCAsmBackend
MCAsmBackend *llvm::createCpu0AsmBackend(const Target &T,
                                         const MCSubtargetInfo &STI,
                                         const MCRegisterInfo &MRI,
                                         const MCTargetOptions &Options) {
    return new Cpu0AsmBackend(T, STI.getTargetTriple());
}
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0BaseInfo.h

```
#include "Cpu0FixupKinds.h"
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```
===== Cpu0ELFObjectWriter.cpp - Cpu0 ELF Writer ===== //
//                                                 The LLVM Compiler Infrastructure
```

(continues on next page)

(continued from previous page)

```

// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
// -----
//===== / 

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/MC/MCAssembler.h"
#include "llvm/MC/MCELFObjectWriter.h"
#include "llvm/MC/MCEExpr.h"
#include "llvm/MC/MCSection.h"
#include "llvm/MC/MCValue.h"
#include "llvm/Support/ErrorHandling.h"
#include <list>

using namespace llvm;

namespace {
    class Cpu0ELFObjectWriter : public MCELFObjectTargetWriter {
public:
    Cpu0ELFObjectWriter(uint8_t OSABI, bool HasRelocationAddend, bool Is64);

    ~Cpu0ELFObjectWriter() = default;

    unsigned getRelocType(MCContext &Ctx, const MCValue &Target,
                          const MCFixup &Fixup, bool IsPCRel) const override;
    bool needsRelocateWithSymbol(const MCSymbol &Sym,
                                 unsigned Type) const override;
};

Cpu0ELFObjectWriter::Cpu0ELFObjectWriter(uint8_t OSABI,
                                         bool HasRelocationAddend, bool Is64)
    : MCELFObjectTargetWriter(/*Is64Bit_=false*/ Is64, OSABI, ELF::EM_CPU0,
                           /*HasRelocationAddend_ = false*/ HasRelocationAddend) {}

// @GetRelocType {
unsigned Cpu0ELFObjectWriter::getRelocType(MCContext &Ctx,
                                         const MCValue &Target,
                                         const MCFixup &Fixup,
                                         bool IsPCRel) const {
    // determine the type of the relocation
    unsigned Type = (unsigned)ELF::R_CPU0_NONE;
    unsigned Kind = (unsigned)Fixup.getKind();

    switch (Kind) {
    default:
        llvm_unreachable("invalid fixup kind!");
    }
}

```

(continues on next page)

(continued from previous page)

```

case FK_Data_4:
    Type = ELF::R_CPU0_32;
    break;
case Cpu0::fixup_Cpu0_32:
    Type = ELF::R_CPU0_32;
    break;
case Cpu0::fixup_Cpu0_GPREL16:
    Type = ELF::R_CPU0_GPREL16;
    break;
case Cpu0::fixup_Cpu0_GOT:
    Type = ELF::R_CPU0_GOT16;
    break;
case Cpu0::fixup_Cpu0_HI16:
    Type = ELF::R_CPU0_HI16;
    break;
case Cpu0::fixup_Cpu0_LO16:
    Type = ELF::R_CPU0_LO16;
    break;
case Cpu0::fixup_Cpu0_GOT_HI16:
    Type = ELF::R_CPU0_GOT_HI16;
    break;
case Cpu0::fixup_Cpu0_GOT_LO16:
    Type = ELF::R_CPU0_GOT_LO16;
    break;
}
}

return Type;
}
//@GetRelocType }

bool
Cpu0ELFObjectWriter::needsRelocateWithSymbol(const MCSymbol &Sym,
                                              unsigned Type) const {
    // FIXME: This is extremely conservative. This really needs to use a
    // whitelist with a clear explanation for why each relocation needs to
    // point to the symbol, not to the section.
    switch (Type) {
    default:
        return true;

    case ELF::R_CPU0_GOT16:
        // For Cpu0 pic mode, I think it's OK to return true but I didn't confirm.
        // llvm_unreachable("Should have been handled already");
        return true;

        // These relocations might be paired with another relocation. The pairing is
        // done by the static linker by matching the symbol. Since we only see one
        // relocation at a time, we have to force them to relocate with a symbol to
        // avoid ending up with a pair where one points to a section and another
        // points to a symbol.
    case ELF::R_CPU0_HI16:
    case ELF::R_CPU0_LO16:

```

(continues on next page)

(continued from previous page)

```
// R_CPU0_32 should be a relocation record, I don't know why Mips set it to
// false.
case ELF::R_CPU0_32:
    return true;

case ELF::R_CPU0_GPREL16:
    return false;
}

std::unique_ptr<MCObjectTargetWriter>
llvm::createCpu0ELFObjectWriter(const Triple &TT) {
    uint8_t OSABI = MCELFObjectTargetWriter::getOSABI(TT.getOS());
    bool IsN64 = false;
    bool HasRelocationAddend = TT.isArch64Bit();
    return std::make_unique<Cpu0ELFObjectWriter>(OSABI, HasRelocationAddend,
                                                IsN64);
}
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0FixupKinds.h

```
===== Cpu0FixupKinds.h - Cpu0 Specific Fixup Entries -----*- C++ -*==/*
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== */

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0FIXUPKINDS_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0FIXUPKINDS_H

#include "Cpu0Config.h"

#include "llvm/MC/MCFixup.h"

namespace llvm {
namespace Cpu0 {
    // Although most of the current fixup types reflect a unique relocation
    // one can have multiple fixup types for a given relocation and thus need
    // to be uniquely named.
    //
    // This table *must* be in the same order of
    // MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds]
    // in Cpu0AsmBackend.cpp.
    // @Fixups {
    enum Fixups {
        // @ Pure upper 32 bit fixup resulting in - R_CPU0_32.
        fixup_Cpu0_32 = FirstTargetFixupKind,
```

(continues on next page)

(continued from previous page)

```
// Pure upper 16 bit fixup resulting in = R_CPU0_HI16.  
fixup_Cpu0_HI16,  
  
// Pure lower 16 bit fixup resulting in = R_CPU0_LO16.  
fixup_Cpu0_LO16,  
  
// 16 bit fixup for GP offset resulting in = R_CPU0_GPREL16.  
fixup_Cpu0_GPREL16,  
  
// GOT (Global Offset Table)  
// Symbol fixup resulting in = R_CPU0_GOT16.  
fixup_Cpu0_GOT,  
  
  
// resulting in = R_CPU0_GOT_HI16  
fixup_Cpu0_GOT_HI16,  
  
// resulting in = R_CPU0_GOT_LO16  
fixup_Cpu0_GOT_LO16,  
  
// Marker  
LastTargetFixupKind,  
NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind  
};  
//@Fixups }  
} // namespace Cpu0  
} // namespace llvm  
  
#endif // LLVM_CPU0_CPU0FIXUPKINDS_H
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCCodeEmitter.h

```
===== Cpu0MCCodeEmitter.h - Convert Cpu0 Code to Machine Code =====  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//===== //  
//  
// This file defines the Cpu0MCCodeEmitter class.  
//  
//===== //  
//  
  
#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCODEEMITTER_H  
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCODEEMITTER_H
```

(continues on next page)

(continued from previous page)

```

#include "Cpu0Config.h"

#include "llvm/MC/MCCodeEmitter.h"
#include "llvm/Support/DataTypes.h"

using namespace llvm;

namespace llvm {
class MCContext;
class MCExpr;
class MCInst;
class MCInstrInfo;
class MCFixup;
class MCOperand;
class MCSubtargetInfo;
class raw_ostream;

class Cpu0MCCodeEmitter : public MCCodeEmitter {
    Cpu0MCCodeEmitter(const Cpu0MCCodeEmitter &) = delete;
    void operator=(const Cpu0MCCodeEmitter &) = delete;
    const MCInstrInfo &MCII;
    MCContext &Ctx;
    bool IsLittleEndian;

public:
    Cpu0MCCodeEmitter(const MCInstrInfo &mcii, MCContext &Ctx_, bool IsLittle)
        : MCII(mcii), Ctx(Ctx_), IsLittleEndian(IsLittle) {}

    ~Cpu0MCCodeEmitter() override {}

    void EmitByte(unsigned char C, raw_ostream &OS) const;

    void EmitInstruction(uint64_t Val, unsigned Size, raw_ostream &OS) const;

    void encodeInstruction(const MCInst &MI, raw_ostream &OS,
                          SmallVectorImpl<MCFixup> &Fixups,
                          const MCSubtargetInfo &STI) const override;

    // getBinaryCodeForInstr - TableGen'ered function for getting the
    // binary encoding for an instruction.
    uint64_t getBinaryCodeForInstr(const MCInst &MI,
                                 SmallVectorImpl<MCFixup> &Fixups,
                                 const MCSubtargetInfo &STI) const;

    // getMachineOpValue - Return binary encoding of operand. If the machine
    // operand requires relocation, record the relocation and return zero.
    unsigned getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                             SmallVectorImpl<MCFixup> &Fixups,
                             const MCSubtargetInfo &STI) const;

    unsigned getMemEncoding(const MCInst &MI, unsigned OpNo,
                           
```

(continues on next page)

(continued from previous page)

```
        SmallVectorImpl<MCFixup> &Fixups,
        const MCSubtargetInfo &STI) const;

    unsigned getExprOpValue(const MCExpr *Expr, SmallVectorImpl<MCFixup> &Fixups,
                           const MCSubtargetInfo &STI) const;
}; // class Cpu0MCCodeEmitter
} // namespace llvm.

#endif
```

Index/chapters/Chapter5_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
//===== Cpu0MCCodeEmitter.cpp - Convert Cpu0 Code to Machine Code =====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file implements the Cpu0MCCodeEmitter class.
//
//=====
//include "Cpu0MCCodeEmitter.h"

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "MCTargetDesc/Cpu0MCExpr.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/MC/MCCodeEmitter.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/raw_ostream.h"

#define DEBUG_TYPE "mccodeemitter"

#define GET_INSTRMAP_INFO
#include "Cpu0GenInstrInfo.inc"
#undef GET_INSTRMAP_INFO

using namespace llvm;
```

(continues on next page)

(continued from previous page)

```

MCCodeEmitter *llvm::createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                                const MCRegisterInfo &MRI,
                                                MCContext &Ctx) {
    return new Cpu0MCCodeEmitter(MCII, Ctx, false);
}

MCCodeEmitter *llvm::createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                                const MCRegisterInfo &MRI,
                                                MCContext &Ctx) {
    return new Cpu0MCCodeEmitter(MCII, Ctx, true);
}

void Cpu0MCCodeEmitter::EmitByte(unsigned char C, raw_ostream &OS) const {
    OS << (char)C;
}

void Cpu0MCCodeEmitter::EmitInstruction(uint64_t Val, unsigned Size, raw_ostream &OS) {
    // Output the instruction encoding in little endian byte order.
    for (unsigned i = 0; i < Size; ++i) {
        unsigned Shift = IsLittleEndian ? i * 8 : (Size - 1 - i) * 8;
        EmitByte((Val >> Shift) & 0xff, OS);
    }
}

/// encodeInstruction - Emit the instruction.
/// Size the instruction (currently only 4 bytes)
void Cpu0MCCodeEmitter::
encodeInstruction(const MCInst &MI, raw_ostream &OS,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const
{
    uint32_t Binary = getBinaryCodeForInstr(MI, Fixups, STI);

    // Check for unimplemented opcodes.
    // Unfortunately in CPU0 both NOT and SLL will come in with Binary == 0
    // so we have to special check for them.
    unsigned Opcode = MI.getOpcode();
    if ((Opcode != Cpu0::NOP) && (Opcode != Cpu0::SHL) && !Binary)
        llvm_unreachable("unimplemented opcode in encodeInstruction()");

    const MCInstrDesc &Desc = MCII.get(MI.getOpcode());
    uint64_t TSFlags = Desc.TSFlags;

    // Pseudo instructions don't get encoded and shouldn't be here
    // in the first place!
    if ((TSFlags & Cpu0II::FormMask) == Cpu0II::Pseudo)
        llvm_unreachable("Pseudo opcode found in encodeInstruction()");

    // For now all instructions are 4 bytes
    int Size = 4; // FIXME: Have Desc.getSize() return the correct value!
}

```

(continues on next page)

(continued from previous page)

```
    EmitInstruction(Binary, Size, OS);
}

// @getExprOpValue {
unsigned Cpu0MCCodeEmitter::
getExprOpValue(const MCExpr *Expr, SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {
// @getExprOpValue body {
    MCExpr::ExprKind Kind = Expr->getKind();
    if (Kind == MCExpr::Constant) {
        return cast<MCConstantExpr>(Expr)->getValue();
    }

    if (Kind == MCExpr::Binary) {
        unsigned Res = getExprOpValue(cast<MCBinaryExpr>(Expr)->getLHS(), Fixups, STI);
        Res += getExprOpValue(cast<MCBinaryExpr>(Expr)->getRHS(), Fixups, STI);
        return Res;
    }

    if (Kind == MCExpr::Target) {
        const Cpu0MCExpr *Cpu0Expr = cast<Cpu0MCExpr>(Expr);

        Cpu0::Fixups FixupKind = Cpu0::Fixups(0);
        switch (Cpu0Expr->getKind()) {
        default: llvm_unreachable("Unsupported fixup kind for target expression!");
        } // switch
        Fixups.push_back(MCFixup::create(0, Expr, MCFixupKind(FixupKind)));
        return 0;
    }

    // All of the information is in the fixup.
    return 0;
}

/// getMachineOpValue - Return binary encoding of operand. If the machine
/// operand requires relocation, record the relocation and return zero.
unsigned Cpu0MCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const {
    if (MO.isReg()) {
        unsigned Reg = MO.getReg();
        unsigned RegNo = Ctx.getRegisterInfo()->getEncodingValue(Reg);
        return RegNo;
    } else if (MO.isImm()) {
        return static_cast<unsigned>(MO.getImm());
    } else if (MO.isFPIImm()) {
        return static_cast<unsigned>(APFloat(MO.getFPIImm())
            .bitcastToAPInt().getHiBits(32).getLimitedValue());
    }
    // MO must be an Expr.
    assert(MO.isExpr());
}
```

(continues on next page)

(continued from previous page)

```

        return getExprOpValue(MO.getExpr(), Fixups, STI);
    }

/// getMemEncoding - Return binary encoding of memory related operand.
/// If the offset operand requires relocation, record the relocation.
unsigned
Cpu0MCCodeEmitter::getMemEncoding(const MCInst &MI, unsigned OpNo,
                                  SmallVectorImpl<MCFixup> &Fixups,
                                  const MCSubtargetInfo &STI) const {
    // Base register is encoded in bits 20-16, offset is encoded in bits 15-0.
    assert(MI.getOperand(OpNo).isReg());
    unsigned RegBits = getMachineOpValue(MI, MI.getOperand(OpNo), Fixups, STI) << 16;
    unsigned OffBits = getMachineOpValue(MI, MI.getOperand(OpNo+1), Fixups, STI);

    return (OffBits & 0xFFFF) | RegBits;
}

#include "Cpu0GenMCCodeEmitter.inc"

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCExpr.h

```

===== Cpu0MCExpr.h - Cpu0 specific MC expression classes -----*- C++ -*==/*
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== */

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCEXPR_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCEXPR_H

#include "Cpu0Config.h"
#if CH >= CH5_1

#include "llvm/MC/MCAsmLayout.h"
#include "llvm/MC/MCE Expr.h"
#include "llvm/MC/MCValue.h"

namespace llvm {

class Cpu0MCExpr : public MCTargetExpr {
public:
    enum Cpu0ExprKind {
        CEK_None,
        CEK_ABS_HI,
        CEK_ABS_LO,
        CEK_CALL_HI16,
        CEK_CALL_LO16,

```

(continues on next page)

(continued from previous page)

```

CEK_DTP_HI,
CEK_DTP_LO,
CEK_GOT,
CEK_GOTPREL,
CEK_GOT_CALL,
CEK_GOT_DISP,
CEK_GOT_HI16,
CEK_GOT_LO16,
CEK_GPREL,
CEK_TLSGD,
CEK_TLSLDM,
CEK_TP_HI,
CEK_TP_LO,
CEK_Special,
};

private:
    const Cpu0ExprKind Kind;
    const MCExpr *Expr;

    explicit Cpu0MCExpr(Cpu0ExprKind Kind, const MCExpr *Expr)
        : Kind(Kind), Expr(Expr) {}

public:
    static const Cpu0MCExpr *create(Cpu0ExprKind Kind, const MCExpr *Expr,
                                    MCContext &Ctx);
    static const Cpu0MCExpr *create(const MCSymbol *Symbol,
                                    Cpu0MCExpr::Cpu0ExprKind Kind, MCContext &Ctx);
    static const Cpu0MCExpr *createGpOff(Cpu0ExprKind Kind, const MCExpr *Expr,
                                         MCContext &Ctx);

    /// Get the kind of this expression.
    Cpu0ExprKind getKind() const { return Kind; }

    /// Get the child of this expression.
    const MCExpr *getSubExpr() const { return Expr; }

    void printImpl(raw_ostream &OS, const MCAsmInfo *MAI) const override;
    bool evaluateAsRelocatableImpl(MCValue &Res, const MCAsmLayout *Layout,
                                   const MCFixup *Fixup) const override;
    void visitUsedExpr(MCStreamer &Streamer) const override;
    MCFragment *findAssociatedFragment() const override {
        return getSubExpr()->findAssociatedFragment();
    }

    void fixELFSymbolsInTLSFixups(MCAssembler &Asm) const override;

    static bool classof(const MCExpr *E) {
        return E->getKind() == MCExpr::Target;
    }

    bool isGpOff(Cpu0ExprKind &Kind) const;

```

(continues on next page)

(continued from previous page)

```

bool isGpOff() const {
    Cpu0ExprKind Kind;
    return isGpOff(Kind);
}
};

} // end namespace llvm

#endif // #if CH >= CH5_1

#endif

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCExpr.cpp

```

//===== Cpu0MCExpr.cpp - Cpu0 specific MC expression classes ===== //
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===== //

#include "Cpu0.h"

#if CH >= CH5_1

#include "Cpu0MCExpr.h"
#include "llvm/BinaryFormat/ELF.h"
#include "llvm/MC/MCAsmInfo.h"
#include "llvm/MC/MCAssembler.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCObjectStreamer.h"
#include "llvm/MC/MCSymbolELF.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0mcexpr"

const Cpu0MCExpr *Cpu0MCExpr::create(Cpu0MCExpr::Cpu0ExprKind Kind,
                                       const MCExpr *Expr, MCContext &Ctx) {
    return new (Ctx) Cpu0MCExpr(Kind, Expr);
}

const Cpu0MCExpr *Cpu0MCExpr::create(const MCSymbol *Symbol, Cpu0MCExpr::Cpu0ExprKind_
                                      Kind,
                                      MCContext &Ctx) {
    const MCSymbolRefExpr *MCSym =
        MCSymbolRefExpr::create(Symbol, MCSymbolRefExpr::VK_None, Ctx);
    return new (Ctx) Cpu0MCExpr(Kind, MCSym);
}

```

(continues on next page)

(continued from previous page)

```
const Cpu0MCExpr *Cpu0MCExpr::createGpOff(Cpu0MCExpr::Cpu0ExprKind Kind,
                                            const MCExpr *Expr, MCContext &Ctx) {
    return create(Kind, create(CEK_None, create(CEK_GPREL, Expr, Ctx), Ctx), Ctx);
}

void Cpu0MCExpr::printImpl(raw_ostream &OS, const MCAsmInfo *MAI) const {
    int64_t AbsVal;

    switch (Kind) {
        case CEK_None:
        case CEK_Special:
            llvm_unreachable("CEK_None and CEK_Special are invalid");
            break;
        case CEK_CALL_HI16:
            OS << "%call_hi";
            break;
        case CEK_CALL_LO16:
            OS << "%call_lo";
            break;
        case CEK_DTP_HI:
            OS << "%dtp_hi";
            break;
        case CEK_DTP_LO:
            OS << "%dtp_lo";
            break;
        case CEK_GOT:
            OS << "%got";
            break;
        case CEK_GOTPREL:
            OS << "%gottprel";
            break;
        case CEK_GOT_CALL:
            OS << "%call16";
            break;
        case CEK_GOT_DISP:
            OS << "%got_disp";
            break;
        case CEK_GOT_HI16:
            OS << "%got_hi";
            break;
        case CEK_GOT_LO16:
            OS << "%got_lo";
            break;
        case CEK_GPREL:
            OS << "%gp_rel";
            break;
        case CEK_ABS_HI:
            OS << "%hi";
            break;
        case CEK_ABS_LO:
            OS << "%lo";
            break;
    }
}
```

(continues on next page)

(continued from previous page)

```

case CEK_TLSGD:
    OS << "%tsgd";
break;
case CEK_TLSLDM:
    OS << "%tsldm";
break;
case CEK_TP_HI:
    OS << "%tp_hi";
break;
case CEK_TP_LO:
    OS << "%tp_lo";
break;
}

OS << '(';
if (Expr->evaluateAsAbsolute(AbsVal))
    OS << AbsVal;
else
    Expr->print(OS, MAI, true);
OS << ')';
}

bool
Cpu0MCExpr::evaluateAsRelocatableImpl(MCValue &Res,
                                       const MCAsmLayout *Layout,
                                       const MCFixup *Fixup) const {
    return getSubExpr()->evaluateAsRelocatable(Res, Layout, Fixup);
}

void Cpu0MCExpr::visitUsedExpr(MCStreamer &Streamer) const {
    Streamer.visitUsedExpr(*getSubExpr());
}

void Cpu0MCExpr::fixELFSymbolsInTLSFixups(MCAssembler &Asm) const {
    switch ((int)getKind()) {
        case CEK_None:
        case CEK_Special:
            llvm_unreachable("CEK_None and CEK_Special are invalid");
            break;
        case CEK_CALL_HI16:
        case CEK_CALL_LO16:
            break;
    }
}

bool Cpu0MCExpr::isGpOff(Cpu0ExprKind &Kind) const {
    if (const Cpu0MCExpr *S1 = dyn_cast<const Cpu0MCExpr>(getSubExpr())) {
        if (const Cpu0MCExpr *S2 = dyn_cast<const Cpu0MCExpr>(S1->getSubExpr())) {
            if (S1->getKind() == CEK_None && S2->getKind() == CEK_GPREL) {
                Kind = getKind();
                return true;
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        }
    }

    return false;
}

#endif

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.h

```

MCCodeEmitter *createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                         const MCRegisterInfo &MRI,
                                         MCContext &Ctx);
MCCodeEmitter *createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                         const MCRegisterInfo &MRI,
                                         MCContext &Ctx);

MCAsmBackend *createCpu0AsmBackend(const Target &T,
                                   const MCSubtargetInfo &STI,
                                   const MCRegisterInfo &MRI,
                                   const MCTargetOptions &Options);

std::unique_ptr<MCObjectTargetWriter> createCpu0ELFObjectWriter(const Triple &TT);

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

```

static MCStreamer *createMCStreamer(const Triple &TT, MCContext &Context,
                                    std::unique_ptr<MCAsmBackend> &&MAB,
                                    std::unique_ptr<MCObjectWriter> &&OW,
                                    std::unique_ptr<MCCodeEmitter> &&Emitter,
                                    bool RelaxAll) {
    return createELFStreamer(Context, std::move(MAB), std::move(OW),
                             std::move(Emitter), RelaxAll);
}

static MCTargetStreamer *createCpu0AsmTargetStreamer(MCStreamer &S,
                                                     formatted_raw_ostream &OS,
                                                     MCInstPrinter *InstPrint,
                                                     bool isVerboseAsm) {
    return new Cpu0TargetAsmStreamer(S, OS);
}

```

```
extern "C" void LLVMInitializeCpu0TargetMC() {
```

```

    // Register the elf streamer.
    TargetRegistry::RegisterELFStreamer(*T, createMCStreamer);

    // Register the asm target streamer.
    TargetRegistry::RegisterAsmTargetStreamer(*T, createCpu0AsmTargetStreamer);

    // Register the asm backend.
    TargetRegistry::RegisterMCAsmBackend(*T, createCpu0AsmBackend);

```

```
// Register the MC Code Emitter
TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                      createCpu0MCCodeEmitterEB);
TargetRegistry::RegisterMCCodeEmitter(TheCpu0e1Target,
                                      createCpu0MCCodeEmitterEL);

}
```

Ibdex/chapters/Chapter5_1/Cpu0MCInstLower.h

```
#include "MCTargetDesc/Cpu0MCExpr.h"
```

5.3 Work flow

In Chapter 3_2, *OutStreamer->emitInstruction* prints the assembly code. To support ELF object file generation, this chapter creates *MCELFObjectStreamer*, which inherits from *OutStreamer* by calling *createELFStreamer* in *Cpu0MCTargetDesc.cpp* above.

Once *MCELFObjectStreamer* is created, *OutStreamer->emitInstruction* will work with other code added in the *MCTargetDesc* directory of this chapter. The details of the explanation are as follows:

Ilvm/include/Ilvm/CodeGen/AsmPrinter.h

```
class AsmPrinter : public MachineFunctionPass {
public:
    ...
    std::unique_ptr<MCStreamer> OutStreamer;
    ...
}
```

Ibdex/chapters/Chapter3_2/Cpu0AsmPrinter.h

```
class LLVM_LIBRARY_VISIBILITY Cpu0AsmPrinter : public AsmPrinter {
```

Ibdex/chapters/Chapter3_2/Cpu0AsmPrinter.cpp

```
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
    ...
    do {
        ...
        OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
        ...
    } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}
```

- AsmPrinter::OutStreamer is nMCObjectStreamer if llc -filetype=obj; AsmPrinter::OutStreamer is nMCAsmStreamer if llc -filetype=asm as Fig. 3.8.

The ELF encoder calling functions are shown in Fig. 5.2 above. *AsmPrinter::OutStreamer* is set to *MCOObjectStreamer* by the *llc* driver when the user inputs *llc -filetype=obj*.

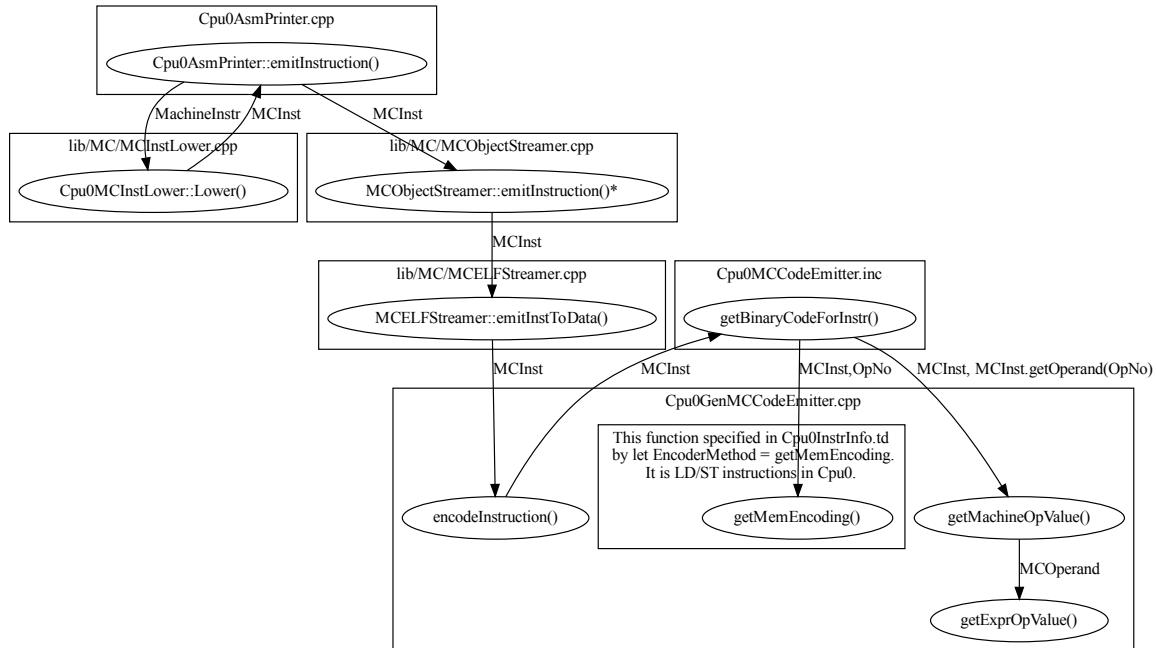


Fig. 5.2: Calling Functions of elf encoder

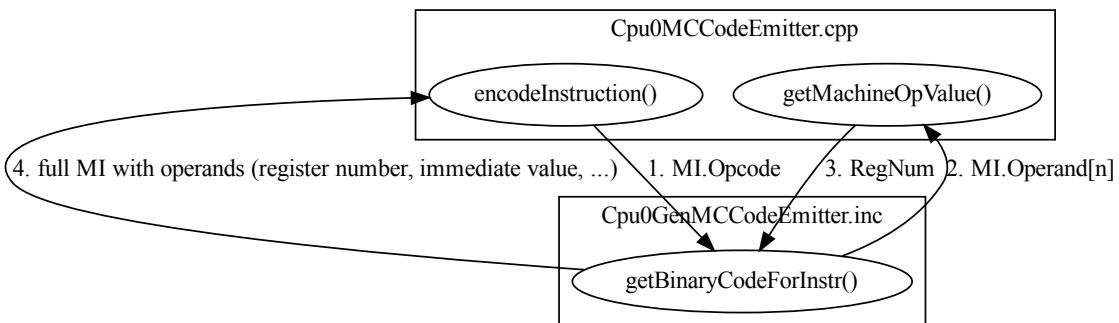


Fig. 5.3: DFD flow for instruction encode

The instruction operand information for the encoder is obtained as shown in Fig. 5.3 above. The steps are as follows:

1. The function `encodeInstruction()` passes `MI.Opcode` to `getBinaryCodeForInstr()`.
2. `getBinaryCodeForInstr()` passes `MI.Operand[n]` to `getMachineOpValue()`, and then,
3. It retrieves the register number by calling `getMachineOpValue()`.
4. `getBinaryCodeForInstr()` returns `MI` with all register numbers to `encodeInstruction()`.

The `MI.Opcode` is set in the Instruction Selection Stage. The table generation function `getBinaryCodeForInstr()` gathers all operand information from the `.td` files defined by the programmer, as shown in Fig. 5.4.

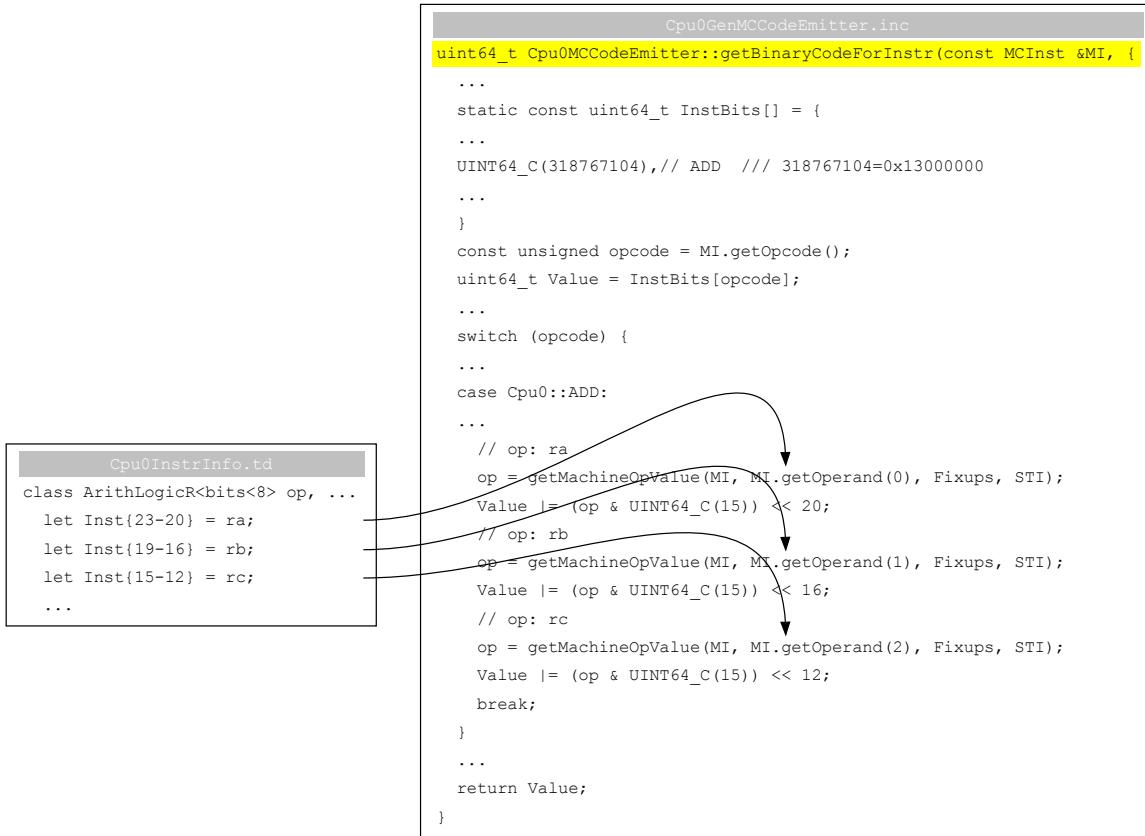


Fig. 5.4: Instruction encode, for instance: `addu $v0, $at, $v1n v0:MI.getOperand(0), at:MI.getOperand(1), v1:MI.getOperand(2)`

For instance, the Cpu0 backend will generate “`addu $v0, $at, $v1`” for the IR “`%0 = add %1, %2`” once LLVM allocates registers `$v0`, `$at`, and `$v1` for operands `%0`, `%1`, and `%2` individually. The `MCOperand` structure for `MI.Operands[]` includes the register number set in the pass where LLVM allocates registers, which can be retrieved in `getMachineOpValue()`.

The function `getEncodingValue(Reg)` in `getMachineOpValue()`, as shown below, retrieves the `RegNo` for encoding from register names such as `AT`, `V0`, or `V1`, using table generation information from `Cpu0RegisterInfo.td`. My comments are after “`///`”.

include/llvm/MC/MCRegisterInfo.h

```
void InitMCRegisterInfo(...,
                       const uint16_t *RET) {
    ...
    RegEncodingTable = RET;
}

unsigned Cpu0MCCodeEmitter::getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                                              SmallVectorImpl<MCFixup> &Fixups,
                                              const MCSubtargetInfo &STI) const {
    if (MO.isReg()) {
        unsigned Reg = MO.getReg();
        unsigned RegNo = Ctx.getRegisterInfo()->getEncodingValue(Reg);
        return RegNo;
    }
}
```

include/llvm/MC/MCRegisterInfo.h

```
void InitMCRegisterInfo(...,
                       const uint16_t *RET) {
    ...
    RegEncodingTable = RET;
}

/// \brief Returns the encoding for RegNo
uint16_t getEncodingValue(unsigned RegNo) const {
    assert(RegNo < NumRegs &&
           "Attempting to get encoding for invalid register number!");
    return RegEncodingTable[RegNo];
}
```

Index/chapters/Chapter5_1/Cpu0RegisterInfo.td

```
let Namespace = "Cpu0" in {
    ...
    def AT    : Cpu0GPRReg<1,  "1">,      DwarfRegNum<[1]>;
    def V0    : Cpu0GPRReg<2,  "2">,      DwarfRegNum<[2]>;
    def V1    : Cpu0GPRReg<3,  "3">,      DwarfRegNum<[3]>;
    ...
}
```

build/lib/Target/Cpu0/Cpu0GenRegisterInfo.inc

```
namespace Cpu0 {
enum {
    NoRegister,
    AT = 1,
    ...
    V0 = 19,
```

(continues on next page)

(continued from previous page)

```

v1 = 20,
NUM_TARGET_REGS      // 21
};

} // end namespace Cpu0

extern const uint16_t Cpu0RegEncodingTable[] = {
0,
1,    /// 1, AT
1,
12,
11,
0,
0,
14,
0,
13,
15,
0,
4,
5,
9,
10,
7,
8,
6,
2,    /// 19, V0
3,    /// 20, V1
};

static inline void InitCpu0MCRegisterInfo(MCRegisterInfo *RI, ...) {
    RI->InitMCRegisterInfo(..., Cpu0RegEncodingTable);
}

```

The `applyFixup()` function in `Cpu0AsmBackend.cpp` will fix up the `jeq`, `jub`, and other instructions related to “address control flow statements” or “function call statements” used in later chapters.

The setting of `true` or `false` for each relocation record in `needsRelocateWithSymbol()` of `Cpu0ELFObjectWriter.cpp` depends on whether this relocation record needs to adjust the address value during linking.

- If set to `true`, the linker has the opportunity to adjust this address value with the correct information.
- If set to `false`, the linker lacks the correct information to adjust this relocation record.

The concept of relocation records will be introduced in a later chapter on **ELF Support**.

When emitting ELF object format instructions, the `EncodeInstruction()` function in `Cpu0MCCCodeEmitter.cpp` will be called, as it overrides the function of the same name in the parent class `MCCCodeEmitter`.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```

// Address operand
def mem : Operand<iPTR> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops GPROut, simm16);
    let EncoderMethod = "getMemEncoding";
}

```

```
}
```

```
class LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
               bit Pseudo = 0>
: LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {

// 32-bit store.
class StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
               bit Pseudo = 0>
: StoreM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
```

As shown in Fig. 2.2, **ld** and **st** follow the L-Type format, whereas **ADD** and similar instructions follow the R-Type format.

The statement “*let EncoderMethod = “getMemEncoding”;*” in *Cpu0InstrInfo.td*, as mentioned above, ensures that LLVM calls the function *getMemEncoding()* whenever an **ld** or **st** instruction is issued in the ELF object file. This happens because both of these instructions use the **mem** operand.

Below is the implementation and the corresponding TableGen code for them.

Ibdex/chapters/Chapter5_1/Cpu0InstrInfo.td

```
def mem : Operand<iPTR> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops GPROut, simm16);
    let EncoderMethod = "getMemEncoding";
}
```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
/// If the offset operand requires relocation, record the relocation.
unsigned
Cpu0MCCodeEmitter::getMemEncoding(const MCInst &MI, unsigned OpNo,
                                  SmallVectorImpl<MCFixup> &Fixups,
                                  const MCSubtargetInfo &STI) const {
    // Base register is encoded in bits 20-16, offset is encoded in bits 15-0.
    assert(MI.getOperand(OpNo).isReg());
    unsigned RegBits = getMachineOpValue(MI, MI.getOperand(OpNo), Fixups, STI) << 16;
    unsigned OffBits = getMachineOpValue(MI, MI.getOperand(OpNo+1), Fixups, STI);

    return (OffBits & 0xFFFF) | RegBits;
}

#include "Cpu0GenMCCodeEmitter.inc"
```

build/lib/Target/Cpu0/Cpu0GenMCCodeEmitter.inc

```

case Cpu0::LD
case Cpu0::ST: {
    // op: ra
    op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
    op &= UINT64_C(15);
    op <= 20;
    Value |= op;
    // op: addr
    op = getMemEncoding(MI, 1, Fixups, STI);
    op &= UINT64_C(1048575);
    Value |= op;
    break;
}

```

The other functions in *Cpu0MCCodeEmitter.cpp* are called by these two functions.

After encoding, the following code will write the encoded instructions to the buffer.

src/lib/MC/MCELFStreamer.cpp

```

void MCELFStreamer::EmitInstToData(const MCInst &Inst,
                                    const MCSubtargetInfo &STI) {
    ...
    DF->setHasInstructions(true);
    DF->getContents().append(Code.begin(), Code.end());
    ...
}

```

Then, *ELFObjectWriter::writeObject()* will write the buffer to the ELF file.

5.4 Backend Target Registration Structure

Now, let's examine *Cpu0MCTargetDesc.cpp*. *Cpu0MCTargetDesc.cpp* performs target registration as mentioned in the previous chapter here¹. The assembly output has been explained here².

The register functions for ELF object output are listed as follows:

Register function of elf streamer

```

// Register the elf streamer.
TargetRegistry::RegisterELFStreamer(*T, createMCStreamer);

static MCStreamer *createMCStreamer(const Triple &TT, MCContext &Context,
                                    MCAsmBackend &MAB, raw_pwrite_stream &OS,
                                    MCCodeEmitter *Emitter, bool RelaxAll) {
    return createELFStreamer(Context, MAB, OS, Emitter, RelaxAll);
}

// MCELFStreamer.cpp

```

(continues on next page)

¹ <http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration>

² <http://jonathan2251.github.io/lbd/backendstructure.html#add-asmprinter>

(continued from previous page)

```
MCStreamer *llvm::createELFStreamer(MCContext &Context, MCAsmBackend &MAB,
                                    raw_pwrite_stream &OS, MCCodeEmitter *CE,
                                    bool RelaxAll) {
    MCELFSStreamer *S = new MCELFSStreamer(Context, MAB, OS, CE);
    if (RelaxAll)
        S->getAssembler().setRelaxAll(true);
    return S;
}
```

Above, `createELFStreamer` handles the ELF object streamer. Fig. 5.5 below shows the `MCELFSStreamer` inheritance tree. You can find many operations within that inheritance tree.



Fig. 5.5: MCELFSStreamer inherit tree

Register function of asm target streamer

```
// Register the asm target streamer.
TargetRegistry::RegisterAsmTargetStreamer(*T, createCpu0AsmTargetStreamer);

static MCTargetStreamer *createCpu0AsmTargetStreamer(MCStreamer &S,
                                                       formatted_raw_ostream &OS,
```

(continues on next page)

(continued from previous page)

```

MCInstPrinter *InstPrint,
bool isVerboseAsm) {

return new Cpu0TargetAsmStreamer(S, OS);
}

// Cpu0TargetStreamer.h
class Cpu0TargetStreamer : public MCTargetStreamer {
public:
    Cpu0TargetStreamer(MCStreamer &S);
};

// This part is for ascii assembly output
class Cpu0TargetAsmStreamer : public Cpu0TargetStreamer {
    formatted_raw_ostream &OS;

public:
    Cpu0TargetAsmStreamer(MCStreamer &S, formatted_raw_ostream &OS);
};

```

Above instancing MCTargetStreamer instance.

Register function of MC Code Emitter

```

// Register the MC Code Emitter
TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                      createCpu0MCCodeEmitterEB);
TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                                      createCpu0MCCodeEmitterEL);

// Cpu0MCCodeEmitter.cpp
MCCodeEmitter *llvm::createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                                const MCRegisterInfo &MRI,
                                                MCContext &Ctx) {
    return new Cpu0MCCodeEmitter(MCII, Ctx, false);
}

MCCodeEmitter *llvm::createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                              const MCRegisterInfo &MRI,
                                              MCContext &Ctx) {
    return new Cpu0MCCodeEmitter(MCII, Ctx, true);
}

```

Above, two *Cpu0MCCodeEmitter* objects are instantiated. One for big-endian and the other for little-endian. They handle the object format generation, while *RegisterELFStreamer()* reuses the ELF streamer class.

Readers may wonder: “What are the actual arguments in *createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII, const MCSubtargetInfo &STI, MCContext &Ctx)?*” and “When are they assigned?”

At this point, we have not assigned them yet. Instead, we register the *createXXX()* function using a function pointer (according to C, *TargetRegistry::RegisterXXX(TheCpu0Target, createXXX())*, where *createXXX* is a function pointer).

LLVM stores a function pointer to *createXXX()* when we call target registration. Later, during the target registration process (*RegisterXXX()*), LLVM invokes these *createXXX()* functions with the necessary arguments (LLVM invokes these arguments during run time in LLVM structure)`.

Register function of asm backend

```
// Register the asm backend.
TargetRegistry::RegisterMCAsmBackend(TheCpu0Target,
                                      createCpu0AsmBackendEB32);
TargetRegistry::RegisterMCAsmBackend(TheCpu0elTarget,
                                      createCpu0AsmBackendEL32);

// Cpu0AsmBackend.cpp
MCAsmBackend *llvm::createCpu0AsmBackendEL32(const Target &T,
                                              const MCRegisterInfo &MRI,
                                              const Triple &TT, StringRef CPU) {
    return new Cpu0AsmBackend(T, TT.getOS(), /*IsLittle*/true);
}

MCAsmBackend *llvm::createCpu0AsmBackendEB32(const Target &T,
                                              const MCRegisterInfo &MRI,
                                              const Triple &TT, StringRef CPU) {
    return new Cpu0AsmBackend(T, TT.getOS(), /*IsLittle*/false);
}

// Cpu0AsmBackend.h
class Cpu0AsmBackend : public MCAsmBackend {
    ...
}
```

The *Cpu0AsmBackend* class serves as the bridge between assembly and object file generation. Two instances handle big-endian and little-endian formats, respectively.

This class is derived from *MCAsmBackend*. Most of the code responsible for object file generation is implemented in *MCELFStreamer* and its parent class, *MCAsmBackend*.

GLOBAL VARIABLES

- *Cpu0 Global Variable Options*
- *Static mode*
 - *data or bss*
 - *sdata or sbss*
- *PIC Mode*
 - *sdata or sbss*
 - *data or bss*
- *Global variable print support*
- *Summary*

In the last three chapters, we accessed only local variables. This chapter focuses on translating global variable access.

The global variable DAG translation differs from previous DAG translations discussed so far. It creates IR DAG nodes at runtime in backend C++ code based on the `llc -relocation-model` option. In contrast, other DAGs translate IR DAGs directly to Machine DAGs according to the input IR DAG files (except for the Pseudo instruction `RetLR` used in Chapter3_4).

Readers should focus on how to add code for creating DAG nodes at runtime and how to define pattern matching in `.td` files for these runtime-created DAG nodes. Additionally, if your backend has assembly directives (macros) related to global variables, ensure that the machine instruction printing function handles them correctly.

Chapter6_1/ introduces global variable support. Let's compile `ch6_1.cpp` with this version first, then review the code changes.

Ibdex/input/ch6_1.cpp

```
int gStart = 3;
int gI = 100;
int test_global()
{
    int c = 0;

    c = gI;
```

(continues on next page)

(continued from previous page)

```
return c;
}
```

```
118-165-78-166:input Jonathan$ llvm-dis ch6_1.bc -o -
...
@gStart = global i32 2, align 4
@gI = global i32 100, align 4

define i32 @_Z3funv() nounwind uwtable ssp {
    %1 = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 0, i32* %1
    store i32 0, i32* %c, align 4
    %2 = load i32* @gI, align 4
    store i32 %2, i32* %c, align 4
    %3 = load i32* %c, align 4
    ret i32 %3
}
```

6.1 Cpu0 Global Variable Options

Like MIPS, Cpu0 supports both static and PIC modes. There are two different layouts for global variables in static mode, controlled by the option `cpu0-use-small-section`.

Chapter6_1/ introduces global variable translation. Let's run Chapter6_1/ with `ch6_1.cpp` using four different options to trace the DAGs and Cpu0 instructions:

- `llc -relocation-model=static -cpu0-use-small-section=false`
- `llc -relocation-model=static -cpu0-use-small-section=true`
- `llc -relocation-model=pic -cpu0-use-small-section=false`
- `llc -relocation-model=pic -cpu0-use-small-section=true`

```
118-165-78-166:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch6_1.cpp -emit-llvm -o ch6_1.bc
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=false
-filetype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 12 nodes:
...
    0x7ffd5902cc10: <multiple use>
    0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
    0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7ffd5902d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7ffd5902cc10: <multiple use>
    0x7ffd5902d110: i32, ch = load 0x7ffd5902cf10, 0x7ffd5902d010,
```

(continues on next page)

(continued from previous page)

```

0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=-3]
...
Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 16 nodes:
...
0x7ffd5902cc10: <multiple use>
0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=8]

0x7ffd5902d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=5]

0x7ffd5902d710: i32 = Cpu0ISD::Hi 0x7ffd5902d310

0x7ffd5902d610: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=6]

0x7ffd5902d810: i32 = Cpu0ISD::Lo 0x7ffd5902d610

0x7ffd5902fe10: i32 = add 0x7ffd5902d710, 0x7ffd5902d810

0x7ffd5902cc10: <multiple use>
0x7ffd5902d110: i32, ch = load 0x7ffd5902cf10, 0x7ffd5902fe10,
0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=9]
...
lui $2, %hi(gI)
ori $2, $2, %lo(gI)
    ld      $2, 0($2)
...
.type   gStart,@object          # @gStart
.data
.globl  gStart
.align   2
gStart:
    .4byte  2                  # 0x2
    .size    gStart, 4

.type   gI,@object            # @gI
.globl  gI
.align   2
gI:
    .4byte  100                # 0x64
    .size    gI, 4

```

```

118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=true -filetype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 12 nodes:
...
0x7fc5f382cc10: <multiple use>

```

(continues on next page)

(continued from previous page)

```
0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,  
0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=-3]  
  
0x7fc5f382d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]  
  
0x7fc5f382cc10: <multiple use>  
0x7fc5f382d110: i32, ch = load 0x7fc5f382cf10, 0x7fc5f382d010,  
0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=-3]  
...  
Legalized selection DAG: BB#0 '_Z11test_globalv:'  
SelectionDAG has 15 nodes:  
...  
0x7fc5f382cc10: <multiple use>  
0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,  
0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=8]  
  
0x7fc5f382d710: i32 = register %GP  
  
0x7fc5f382d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=4]  
  
0x7fc5f382d610: i32 = Cpu0ISD::GPRel 0x7fc5f382d310  
  
0x7fc5f382d810: i32 = add 0x7fc5f382d710, 0x7fc5f382d610  
  
0x7fc5f382cc10: <multiple use>  
0x7fc5f382d110: i32, ch = load 0x7fc5f382cf10, 0x7fc5f382d810,  
0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=9]  
...  
  
ori      $2, $gp, %gp_rel(gI)  
ld       $2, 0($2)  
...  
.type   gStart,@object          # @gStart  
.section .sdata,"aw",@progbits  
.globl  gStart  
.align   2  
gStart:  
.4byte  2                      # 0x2  
.size    gStart, 4  
  
.type   gI,@object            # @gI  
.globl  gI  
.align   2  
gI:  
.4byte  100                     # 0x64  
.size    gI, 4
```

```
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/  
bin/llc -march=cpu0 -relocation-model=pic -cpu0-use-small-section=false  
-filetype=asm -debug ch6_1.bc -o -
```

...

(continues on next page)

(continued from previous page)

```
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 11 nodes:
...
0x7fe03c02e010: <multiple use>
0x7fe03c02e118: ch = store 0x7fe03b50dee0, 0x7fe03c02de00, 0x7fe03c02df08,
0x7fe03c02e010<ST4[%c]> [ORD=3] [ID=-3]

0x7fe03c02e220: i32 = GlobalAddress<i32* @gI> 0 [ORD=4] [ID=-3]

0x7fe03c02e010: <multiple use>
0x7fe03c02e328: i32,ch = load 0x7fe03c02e118, 0x7fe03c02e220,
0x7fe03c02e010<LD4[@gI]> [ORD=4] [ID=-3]
...
Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 15 nodes:
...
0x7fe03c02e010: <multiple use>
0x7fe03c02e118: ch = store 0x7fe03b50dee0, 0x7fe03c02de00, 0x7fe03c02df08,
0x7fe03c02e010<ST4[%c]> [ORD=3] [ID=6]

0x7fe03c02e538: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=5] [ORD=4]

0x7fe03c02ea60: i32 = Cpu0ISD::Hi 0x7fe03c02e538 [ORD=4]

0x7fe03c02e958: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=6] [ORD=4]

0x7fe03c02eb68: i32 = Cpu0ISD::Lo 0x7fe03c02e958 [ORD=4]

0x7fe03c02ec70: i32 = add 0x7fe03c02ea60, 0x7fe03c02eb68 [ORD=4]

0x7fe03c02e010: <multiple use>
0x7fe03c02e328: i32,ch = load 0x7fe03c02e118, 0x7fe03c02ec70,
0x7fe03c02e010<LD4[@gI]> [ORD=4] [ID=7]
...
    lui    $2, %got_hi(gI)
    addu   $2, $2, $gp
    ld     $2, %got_lo(gI) ($2)

...
.type gStart,@object          # @gStart
.data
.globl  gStart
.align  2
gStart:
    .4byte 3                  # 0x3
    .size gStart, 4

.type gI,@object              # @gI
.globl  gI
.align  2
gI:
    .4byte 100                # 0x64
    .size gI, 4
```

```
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -cpu0-use-small-section=true
-ffiletype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 11 nodes:
...
    0x7fad7102cc10: <multiple use>
0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=-3]

0x7fad7102d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

0x7fad7102cc10: <multiple use>
0x7fad7102d110: i32,ch = load 0x7fad7102cf10, 0x7fad7102d010,
0x7fad7102cc10<LD4[@gI]> [ORD=3] [ID=-3]
...
Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 14 nodes:
0x7ff3c9c10b98: ch = EntryToken [ORD=1] [ID=0]
...
    0x7fad7102cc10: <multiple use>
0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=8]

0x7fad70c10b98: <multiple use>
0x7fad7102d610: i32 = Register %GP

0x7fad7102d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=1]

0x7fad7102d710: i32 = Cpu0ISD::Wrapper 0x7fad7102d610, 0x7fad7102d310

0x7fad7102cc10: <multiple use>
0x7fad7102d810: i32,ch = load 0x7fad70c10b98, 0x7fad7102d710,
0x7fad7102cc10<LD4[<unknown>]>

0x7ff3ca02cc10: <multiple use>
0x7ff3ca02d110: i32,ch = load 0x7ff3ca02cf10, 0x7ff3ca02d810,
0x7ff3ca02cc10<LD4[@gI]> [ORD=3] [ID=9]
...
.set noreorder
.cupload      $6
.set nomacro
...
ld      $2, %got(gI)($gp)
ld      $2, 0($2)
...
.type   gStart,@object          # @gStart
.data
.globl  gStart
.align  2
gStart:
```

(continues on next page)

(continued from previous page)

```

.4byte 2                      # 0x2
.size   gStart, 4

.type   gI,@object           # @gI
.globl  gI
.align  2

gI:
.4byte 100                   # 0x64
.size   gI, 4

```

Summary above information to Table: Cpu0 global variable options.

Table 6.1: Cpu0 global variable options

option name	default	other option value	description
-relocation-model	pic	static	<ul style="list-style-type: none"> • pic: Postion Independent Address • static: Absolute Address
-cpu0-use-small-section	false	true	<ul style="list-style-type: none"> • false: .data or .bss, 32 bits addressable • true: .sdata or .sbss, 16 bits addressable

Table 6.2: Cpu0 DAGs and instructions for -relocation-model=static

option: cpu0-use-small-section	false	true
addressing mode	absolute	\$gp relative
addressing	absolute	\$gp+offset
Legalized selection DAG	(add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>)	(add register %GP, Cpu0ISD::GPRel<gI offset>)
Cpu0 relocation records solved	lui \$2, %hi(gI); ori \$2, \$2, %lo(gI); link time	ori \$2, \$gp, %gp_rel(gI); link time

- In static mode with `cpu0-use-small-section=true`, the offset between `gI` and `.data` can be calculated since `$gp` is assigned a fixed address at the start of the global address table.
- In static mode with `cpu0-use-small-section=false`, the high and low addresses of `gI` (`%hi(gI)` and `%lo(gI)`) are translated into an absolute address.

Table 6.3: Cpu0 DAGs and instructions for -relocation-model=pic

option: cpu0-use-small-section	false	true
addressing mode	\$gp relative	\$gp relative
addressing	\$gp+offset	\$gp+offset
Legalized selection DAG	(load (Cpu0ISD::Wrapper register %GP, <gI offset>))	(load EntryToken, (Cpu0ISD::Wrapper (add Cpu0ISD::Hi<gI offset Hi16>, Register %GP), Cpu0ISD::Lo<gI offset Lo16>))
Cpu0	ld \$2, %got(gI)(\$gp);	lui \$2, %got_hi(gI); add \$2, \$2, \$gp; ld \$2, %got_lo(gI)(\$2);
relocation records solved	link/load time	link/load time

- In PIC mode, the offset between `gI` and `.data` cannot be calculated if the function is loaded at runtime (dynamic linking). However, the offset can be determined if static linking is used.

- In C, all variable names are bound statically. In C++, overloaded variables or functions are bound dynamically.

According to the system programming book, there are two addressing modes: **Absolute Addressing Mode** and **Position-Independent Addressing Mode**. Dynamic functions must be compiled using Position-Independent Addressing Mode.

In general, the `-relocation-model` option is used to generate either Absolute Addressing or Position-Independent Addressing. However, an exception occurs when using `-relocation-model=static` with `-cpu0-use-small-section=false`. In this case, the `$gp` register is reserved and set at the start address of the global variable area. Cpu0 uses `$gp`-relative addressing in this mode.

To support global variables, first add the **UseSmallSectionOpt** command variable to `Cpu0Subtarget.cpp`.

After that, users can run `llc` with the option `llc -cpu0-use-small-section=false` to explicitly set **UseSmallSectionOpt** to `false`. By default, **UseSmallSectionOpt** is `false` unless specified otherwise.

For more details about the `cl::opt` command-line variable, refer to the documentation¹.

Ibdex/chapters/Chapter6_1/Cpu0Subtarget.h

```
extern bool Cpu0ReserveGP;
extern bool Cpu0NoCupload;

class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
    ...

    // UseSmallSection - Small section is used.
    bool UseSmallSection;

    bool useSmallSection() const { return UseSmallSection; }

    ...
};
```

Ibdex/chapters/Chapter6_1/Cpu0Subtarget.cpp

```
static cl::opt<bool> UseSmallSectionOpt
    ("cpu0-use-small-section", cl::Hidden, cl::init(false),
     cl::desc("Use small section. Only work when -relocation-model="
              "static. pic always not use small section."));

static cl::opt<bool> ReserveGPOpt
    ("cpu0-reserve-gp", cl::Hidden, cl::init(false),
     cl::desc("Never allocate $gp to variable"));

static cl::opt<bool> NoCuploadOpt
    ("cpu0-no-cupload", cl::Hidden, cl::init(false),
     cl::desc("No issue .cupload"));

bool Cpu0ReserveGP;
bool Cpu0NoCupload;
```

¹ <http://llvm.org/docs/CommandLine.html>

```
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                           StringRef FS, bool little,
                           const Cpu0TargetMachine &_TM) :
```

```
// Set UseSmallSection.
UseSmallSection = UseSmallSectionOpt;
Cpu0ReserveGP = ReserveGPOpt;
Cpu0NoCpload = NoCploadOpt;
```

```
...
}
```

The options **ReserveGPOpt** and **NoCploadOpt** will be used in the Cpu0 linker in a later chapter.

Next, add the following code to the files:

- Cpu0BaseInfo.h
- Cpu0TargetObjectFile.h
- Cpu0TargetObjectFile.cpp
- Cpu0RegisterInfo.cpp
- Cpu0ISelLowering.cpp

[Index/chapters/Chapter6_1/Cpu0BaseInfo.h](#)

```
enum TOF {
    ...
    /// MO_GOT16 - Represents the offset into the global offset table at which
    /// the address the relocation entry symbol resides during execution.
    MO_GOT16,
    MO_GOT,
    ...
}; // enum TOF {
```

[Index/chapters/Chapter6_1/Cpu0TargetObjectFile.h](#)

```
bool IsGlobalInSmallSection(const GlobalObject *GO, const TargetMachine &TM,
                           SectionKind Kind) const;
bool IsGlobalInSmallSectionImpl(const GlobalObject *GO,
                               const TargetMachine &TM) const;
```

[Index/chapters/Chapter6_1/Cpu0TargetObjectFile.cpp](#)

```
// A address must be loaded from a small section if its size is less than the
// small section size threshold. Data in this section must be addressed using
// gp_rel operator.
static bool IsInSmallSection(uint64_t Size) {
    return Size > 0 && Size <= SSThreshold;
}

bool Cpu0TargetObjectFile::IsGlobalInSmallSection(
```

(continues on next page)

(continued from previous page)

```
const GlobalObject *GO, const TargetMachine &TM) const {
// We first check the case where global is a declaration, because finding
// section kind using getKindForGlobal() is only allowed for global
// definitions.
if (GO->isDeclaration() || GO->hasAvailableExternallyLinkage())
    return IsGlobalInSmallSectionImpl(GO, TM);

return IsGlobalInSmallSection(GO, TM, getKindForGlobal(GO, TM));
}

/// IsGlobalInSmallSection - Return true if this global address should be
/// placed into small data/bss section.
bool Cpu0TargetObjectFile::
IsGlobalInSmallSection(const GlobalObject *GO, const TargetMachine &TM,
                      SectionKind Kind) const {
    return IsGlobalInSmallSectionImpl(GO, TM) &&
        (Kind.isData() || Kind.isBSS() || Kind.isCommon() ||
         Kind.isReadOnly());
}

/// Return true if this global address should be placed into small data/bss
/// section. This method does all the work, except for checking the section
/// kind.
bool Cpu0TargetObjectFile::
IsGlobalInSmallSectionImpl(const GlobalObject *GV,
                          const TargetMachine &TM) const {
    const Cpu0Subtarget &Subtarget =
        *static_cast<const Cpu0TargetMachine &>(TM).getSubtargetImpl();

    // Return if small section is not available.
    if (!Subtarget.useSmallSection())
        return false;

    // Only global variables, not functions.
    const GlobalVariable *GVA = dyn_cast<GlobalVariable>(GV);
    if (!GVA)
        return false;

    Type *Ty = GV->getValueType();
    return IsInSmallSection(
        GV->getParent()->getDataLayout().getTypeAllocSize(Ty));
}

MCSection *
Cpu0TargetObjectFile::SelectSectionForGlobal(
    const GlobalObject *GO, SectionKind Kind, const TargetMachine &TM) const {
    // TODO: Could also support "weak" symbols as well with ".gnu.linkonce.s.*"
    // sections?

    // Handle Small Section classification here.
    if (Kind.isBSS() && IsGlobalInSmallSection(GO, TM, Kind))
        return IsGlobalInSmallSection(GO, TM, getKindForGlobal(GO, TM));
}
```

(continues on next page)

(continued from previous page)

```

    return SmallBSSSection;
if (Kind.isData() && IsGlobalInSmallSection(GO, TM, Kind))
    return SmallDataSection;
if (Kind.isReadOnly() && IsGlobalInSmallSection(GO, TM, Kind))
    return SmallDataSection;

// Otherwise, we work the same as ELF.
return TargetLoweringObjectFileELF::SelectSectionForGlobal(GO, Kind, TM);
}

```

Ibdex/chapters/Chapter6_1/Cpu0RegisterInfo.cpp

```

BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
...
    Reserved.set(Cpu0::GP);
...
}

```

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```

SDValue getGlobalReg(SelectionDAG &DAG, EVT Ty) const;

// This method creates the following nodes, which are necessary for
// computing a local symbol's address:
//
// (add (load (wrapper $gp, %got(sym)), %lo(sym)))
template<class NodeTy>
SDValue getAddrLocal(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
    SDLoc DL(N);
    unsigned GOTFlag = Cpu0II::MO_GOT;
    SDValue GOT = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                               getTargetNode(N, Ty, DAG, GOTFlag));
    SDValue Load =
        DAG.getLoad(Ty, DL, DAG.getEntryNode(), GOT,
                    MachinePointerInfo::getGOT(DAG.getMachineFunction()));
    unsigned LoFlag = Cpu0II::MO_ABS_LO;
    SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, Ty,
                             getTargetNode(N, Ty, DAG, LoFlag));
    return DAG.getNode(ISD::ADD, DL, Ty, Load, Lo);
}

//@getAddrGlobal {
// This method creates the following nodes, which are necessary for
// computing a global symbol's address:
//
// (load (wrapper $gp, %got(sym)))
template<class NodeTy>
SDValue getAddrGlobal(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                     unsigned Flag, SDValue Chain,

```

(continues on next page)

(continued from previous page)

```

        const MachinePointerInfo &PtrInfo) const {
    SDLoc DL(N);
    SDValue Tgt = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                             getTargetNode(N, Ty, DAG, Flag));
    return DAG.getLoad(Ty, DL, Chain, Tgt, PtrInfo);
}
//@getAddrGlobal }

//@getAddrGlobalLargeGOT {
// This method creates the following nodes, which are necessary for
// computing a global symbol's address in large-GOT mode:
//
// (load (wrapper (add %hi(sym), $gp), %lo(sym)))
template<class NodeTy>
SDValue getAddrGlobalLargeGOT(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                            unsigned HiFlag, unsigned LoFlag,
                            SDValue Chain,
                            const MachinePointerInfo &PtrInfo) const {
    SDLoc DL(N);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, Ty,
                           getTargetNode(N, Ty, DAG, HiFlag));
    SDValue Lo = DAG.getNode(ISD::ADD, DL, Ty, Hi, getGlobalReg(DAG, Ty));
    SDValue Wrapper = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, Hi,
                                 getTargetNode(N, Ty, DAG, LoFlag));
    return DAG.getLoad(Ty, DL, Chain, Wrapper, PtrInfo);
}
//@getAddrGlobalLargeGOT }

//@getAddrNonPIC
// This method creates the following nodes, which are necessary for
// computing a symbol's address in non-PIC mode:
//
// (add %hi(sym), %lo(sym))
template<class NodeTy>
SDValue getAddrNonPIC(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
    SDLoc DL(N);
    SDValue Hi = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_HI);
    SDValue Lo = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_LO);
    return DAG.getNode(ISD::ADD, DL, Ty,
                      DAG.getNode(Cpu0ISD::Hi, DL, Ty, Hi),
                      DAG.getNode(Cpu0ISD::Lo, DL, Ty, Lo));
}

```

[Index/chapters/Chapter6_1/Cpu0ISelLowering.cpp](#)

```

SDValue Cpu0TargetLowering::getGlobalReg(SelectionDAG &DAG, EVT Ty) const {
    Cpu0FunctionInfo *FI = DAG.getMachineFunction().getInfo<Cpu0FunctionInfo>();
    return DAG.getRegister(FI->getGlobalBaseReg(), Ty);
}

//@getTargetNode(GlobalAddressSDNode

```

(continues on next page)

(continued from previous page)

```
SDValue Cpu0TargetLowering::getTargetNode(GlobalAddressSDNode *N, EVT Ty,
                                         SelectionDAG &DAG,
                                         unsigned Flag) const {
    return DAG.getTargetGlobalAddress(N->getGlobal(), SDLoc(N), Ty, 0, Flag);
}

//@getTargetNode(ExternalSymbolSDNode
SDValue Cpu0TargetLowering::getTargetNode(ExternalSymbolSDNode *N, EVT Ty,
                                         SelectionDAG &DAG,
                                         unsigned Flag) const {
    return DAG.getTargetExternalSymbol(N->getSymbol(), Ty, Flag);
}
```

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
: TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
    setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
```

```
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    switch (Op.getOpcode())
    {

        case ISD::GlobalAddress:      return lowerGlobalAddress(Op, DAG);

    }
    return SDValue();
}
```

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
//@lowerGlobalAddress
    SDLoc DL(Op);
    const Cpu0TargetObjectFile *TLOF =
        static_cast<const Cpu0TargetObjectFile *>(
            getTargetMachine().getObjFileLowering());
//@lga 1 {
    EVT Ty = Op.getValueType();
    GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
    const GlobalValue *GV = N->getGlobal();
//@lga 1 }

    if (!isPositionIndependent()) {
        //@ %gp_rel relocation
        const GlobalObject *GO = GV->getBaseObject();
```

(continues on next page)

(continued from previous page)

```

if (GO && TLOF->IsGlobalInSmallSection(GO, getTargetMachine())) {
    SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,
                                             Cpu0II::MO_GPREL);
    SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                    DAG.getVTList(MVT::i32), GA);
    SDValue GPReg = DAG.getRegister(Cpu0::GP, MVT::i32);
    return DAG.getNode(ISD::ADD, DL, MVT::i32, GPReg, GPRelNode);
}

//@ %hi/%lo relocation
return getAddrNonPIC(N, Ty, DAG);
}

if (GV->hasInternalLinkage() || (GV->hasLocalLinkage() && !isa<Function>(GV)))
    return getAddrLocal(N, Ty, DAG);

//@large section
const GlobalObject *GO = GV->getBaseObject();
if (GO && !TLOF->IsGlobalInSmallSection(GO, getTargetMachine()))
    return getAddrGlobalLargeGOT(
        N, Ty, DAG, Cpu0II::MO_GOT_HI16, Cpu0II::MO_GOT_LO16,
        DAG.getEntryNode(),
        MachinePointerInfo::getGOT(DAG.getMachineFunction()));
return getAddrGlobal(
    N, Ty, DAG, Cpu0II::MO_GOT, DAG.getEntryNode(),
    MachinePointerInfo::getGOT(DAG.getMachineFunction()));
}

```

The `setOperationAction(ISD::GlobalAddress, MVT::i32, Custom)` directive informs `llc` that the global address operation is implemented in the C++ function `Cpu0TargetLowering::LowerOperation()`. LLVM will call this function only when it needs to translate the IR DAG for loading a global variable into machine code.

Although all custom IR operations set by `setOperationAction(ISD::XXX, MVT::XXX, Custom)` in the constructor `Cpu0TargetLowering()` trigger calls to `Cpu0TargetLowering::LowerOperation()` during the “Legalized selection DAG” stage, the global address access operation can be specifically identified by checking whether the DAG node’s opcode is `ISD::GlobalAddress`.

Finally, add the following code to `Cpu0ISelDAGToDAG.cpp` and `Cpu0InstrInfo.td`.

Ibdex/chapters/Chapter6_1/Cpu0ISelDAGToDAG.h

```
SDNode *getGlobalBaseReg();
```

Ibdex/chapters/Chapter6_1/Cpu0ISelDAGToDAG.cpp

```

/// getGlobalBaseReg - Output the instructions required to put the
/// GOT address into a register.
SDNode *Cpu0DAGToDAGISel::getGlobalBaseReg() {
    unsigned GlobalBaseReg = MF->getInfo<Cpu0FunctionInfo>()->getGlobalBaseReg();
    return CurDAG->getRegister(GlobalBaseReg, getTargetLowering()->getPointerTy(
        CurDAG->getDataLayout()))
    .getNode();
}

```

```
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
```

```
// on PIC code Load GA
if (Addr.getOpcode() == Cpu0ISD::Wrapper) {
    Base = Addr.getOperand(0);
    Offset = Addr.getOperand(1);
    return true;
}

//@static
if (TM.getRelocationModel() != Reloc::PIC_) {
    if ((Addr.getOpcode() == ISD::TargetExternalSymbol ||
        Addr.getOpcode() == ISD::TargetGlobalAddress))
        return false;
}
```

```
...
```

```
/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
void Cpu0DAGToDAGISel::Select(SDNode *Node) {
```

```
// Get target GOT address.
case ISD::GLOBAL_OFFSET_TABLE:
    ReplaceNode(Node, getGlobalBaseReg());
    return;
```

```
...
```

Ibdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
// Hi and Lo nodes are used to handle global addresses. Used on
// Cpu0ISelLowering to lower stuff like GlobalAddress, ExternalSymbol
// static model. (nothing to do with Cpu0 Registers Hi and Lo)
def Cpu0Hi : SDNode<"Cpu0ISD::Hi", SDTIntUnaryOp>;
def Cpu0Lo : SDNode<"Cpu0ISD::Lo", SDTIntUnaryOp>;
def Cpu0GPRel : SDNode<"Cpu0ISD::GPRel", SDTIntUnaryOp>;
```

```
def Cpu0Wrapper : SDNode<"Cpu0ISD::Wrapper", SDTIntBinOp>;
```

```
def RelocPIC : Predicate<"TM.getRelocationModel() == Reloc::PIC_">;
```

```
// hi/lo relocs
let Predicates = [Ch6_1] in {
```

(continues on next page)

(continued from previous page)

```
def : Pat<(Cpu0Hi tglobaladdr:$in), (LUi tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Lo tglobaladdr:$in), (ORi ZERO, tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(add CPUREgs:$hi, (Cpu0Lo tglobaladdr:$lo)),
          (ORi CPUREgs:$hi, tglobaladdr:$lo)>;
}
```

```
// gp_rel relocs
let Predicates = [Ch6_1] in {
def : Pat<(add CPUREgs:$gp, (Cpu0GPRel tglobaladdr:$in)),
          (ORi CPUREgs:$gp, tglobaladdr:$in)>;
}

// @ wrapper_pic
let Predicates = [Ch6_1] in {
class WrapperPat<SDNode node, Instruction ORiOp, RegisterClass RC>:
    Pat<(Cpu0Wrapper RC:$gp, node:$in),
          (ORiOp RC:$gp, node:$in)>

def : WrapperPat<tglobaladdr, ORi, GPROut>;
}
```

6.2 Static mode

From the table **Cpu0 global variable options**, the option `cpu0-use-small-section=false` places global variables in `.data/.bss`, while `cpu0-use-small-section=true` places them in `.sdata/.sbss`. The `.sdata` section stands for the small data area.

The `.data` and `.sdata` sections store global variables with an initial value (e.g., `int gI = 100;`), while the `.bss` and `.sbss` sections store global variables without an initial value (e.g., `int gI;`).

6.2.1 data or bss

The `.data/.bss` sections are 32-bit addressable areas since Cpu0 is a 32-bit architecture. When `cpu0-use-small-section=false` is set, the following instructions will be generated.

```
...
lui $2, %hi(gI)
ori $2, $2, %lo(gI)
ld $2, 0($2)
...
.type      gStart,@object      # @gStart
.data
.globl    gStart
.align    2
gStart:
```

(continues on next page)

(continued from previous page)

```

.4byte      2          # 0x2
.size       gStart, 4

.type       gI,@object   # @gI
.globl     gI
.align     2

gI:
.4byte      100         # 0x64
.size       gI, 4

```

As above code, it loads the high address part of gI PC relative address (16 bits) to register \$2 and shift 16 bits. Now, the register \$2 got it's high part of gI absolute address. Next, it adds register \$2 and low part of gI absolute address into \$2. At this point, it gets the gI memory address. Finally, it gets the gI content by instruction “ld \$2, 0(\$2)”. The `llc -relocation-model=static` is for absolute address mode which must be used in static link mode. The dynamic link must be encoded with Position Independent Addressing. As you can see, the PC relative address can be solved in static link (The offset between the address of gI and instruction “`lui $2, %hi(gI)`” can be caculated). Since Cpu0 uses PC relative address coding, this program can be loaded to any address and run correctly there. If this program uses absolute address and can be loaded at a specific address known at link stage, the relocation record of gI variable access instruction such as “`lui $2, %hi(gI)`” and “`ori $2, $2, %lo(gI)`” can be solved at link time. On the other hand, if this program use absolute address and the loading address is known at load time, then this relocation record will be solved by loader at load time.

`IsGlobalInSmallSection()` returns true or false depends on `UseSmallSectionOpt`.

The code fragment of `lowerGlobalAddress()` as the following corresponding option `llc -relocation-model=static -cpu0-use-small-section=false` will translate DAG (`GlobalAddress<i32* @gI> 0`) into (add `Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>`) in stage “Legalized selection DAG” as below.

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```

// This method creates the following nodes, which are necessary for
// computing a symbol's address in non-PIC mode:
//
// (add %hi(sym), %lo(sym))
template<class NodeTy>
SDValue getAddrNonPIC(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
    SDLoc DL(N);
    SDValue Hi = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_HI);
    SDValue Lo = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_LO);
    return DAG.getNode(ISD::ADD, DL, Ty,
                        DAG.getNode(Cpu0ISD::Hi, DL, Ty, Hi),
                        DAG.getNode(Cpu0ISD::Lo, DL, Ty, Lo));
}

```

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```

SDValue Cpu0TargetLowering::getTargetNode(GlobalAddressSDNode *N, EVT Ty,
                                         SelectionDAG &DAG,
                                         unsigned Flag) const {
    return DAG.getTargetGlobalAddress(N->getGlobal(), SDLoc(N), Ty, 0, Flag);
}

```

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
    ...
    EVT Ty = Op.getValueType();
    GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
    ...

    if (getTargetMachine().getRelocationModel() != Reloc::PIC_) {
        ...
        // %hi/%lo relocation
        return getAddrNonPIC(N, Ty, DAG);
    }
    ...
}
```

```
118-165-78-166:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch6_1.cpp -emit-llvm -o ch6_1.bc
118-165-78-166:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -cpu0-use-small-section=false
-filetype=asm -debug ch6_1.bc -o -
...

Type-legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 12 nodes:
...
0x7ffd5902cc10: <multiple use>
0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=-3]

0x7ffd5902d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

0x7ffd5902cc10: <multiple use>
0x7ffd5902d110: i32,ch = load 0x7ffd5902cf10, 0x7ffd5902d010,
0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=-3]
...

Legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 16 nodes:
...
0x7ffd5902cc10: <multiple use>
0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=8]

0x7ffd5902d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=5]

0x7ffd5902d710: i32 = Cpu0ISD::Hi 0x7ffd5902d310

0x7ffd5902d610: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=6]

0x7ffd5902d810: i32 = Cpu0ISD::Lo 0x7ffd5902d610

0x7ffd5902fe10: i32 = add 0x7ffd5902d710, 0x7ffd5902d810
```

(continues on next page)

(continued from previous page)

```
0x7ffd5902cc10: <multiple use>
0x7ffd5902d110: i32, ch = load 0x7ffd5902cf10, 0x7ffd5902fe10,
0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=9]
```

Finally, the pattern defined in Cpu0InstrInfo.td as the following will translate DAG (add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>) into Cpu0 instructions as below.

Ibdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
// Hi and Lo nodes are used to handle global addresses. Used on
// Cpu0ISelLowering to lower stuff like GlobalAddress, ExternalSymbol
// static model. (nothing to do with Cpu0 Registers Hi and Lo)
def Cpu0Hi : SDNode<"Cpu0ISD::Hi", SDTIntUnaryOp>;
def Cpu0Lo : SDNode<"Cpu0ISD::Lo", SDTIntUnaryOp>;
```

```
// hi/lo relocs
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Hi tglobaladdr:$in), (LUi tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Lo tglobaladdr:$in), (ORi ZERO, tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)),
          (ORi CPURegs:$hi, tglobaladdr:$lo)>;
}
```

```
...
lui $2, %hi(gI)
ori $2, $2, %lo(gI)
...
```

As above, Pat<(...),(...)> include two lists of DAGs. The left is IR DAG and the right is machine instruction DAG. “Pat<(Cpu0Hi tglobaladdr:\$in), (LUi, tglobaladdr:\$in)>,” will translate DAG (Cpu0ISD::Hi tglobaladdr) into (lui (ori ZERO, tglobaladdr), 16). “Pat<(add CPURegs:\$hi, (Cpu0Lo tglobaladdr:\$lo)), (ORi CPURegs:\$hi, tglobaladdr:\$lo)>,” will translate DAG (add Cpu0ISD::Hi, Cpu0ISD::Lo) into Cpu0 instruction (ori Cpu0ISD::Hi, Cpu0ISD::Lo).

6.2.2 sdata or sbss

The sdata/sbss are 16 bits addressable areas which placed in ELF for fast access. Option cpu0-use-small-section=true will generate the following instructions.

```
ori $2, $gp, %gp_rel(gI)
ld $2, 0($2)
...
.type      gStart,@object      # @gStart
.section   .sdata,"aw",@progbits
.globl    gStart
```

(continues on next page)

(continued from previous page)

```
.align      2
gStart:
    .4byte   2                      # 0x2
    .size     gStart, 4

    .type     gI,@object          # @gI
    .globl   gI
    .align      2
gI:
    .4byte   100                  # 0x64
    .size     gI, 4
```

The code fragment of lowerGlobalAddress() as the following corresponding option `llc -relocation-model=static -cpu0-use-small-section=true` will translate DAG (`GlobalAddress<i32* @gI> 0`) into (add register `%GP` `Cpu0ISD::GPRel<gI offset>`) in stage “Legalized selection DAG” as below.

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
    //@@lowerGlobalAddress }
    SDLoc DL(Op);
    const Cpu0TargetObjectFile *TLOF =
        static_cast<const Cpu0TargetObjectFile *>(
            getTargetMachine().getObjFileLowering());
    //@@lga 1 {
    EVT Ty = Op.getValueType();
    GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
    const GlobalValue *GV = N->getGlobal();
    //@@lga 1 }

    if (!isPositionIndependent()) {
        //@ %gp_rel relocation
        const GlobalObject *GO = GV->getBaseObject();
        if (GO && TLOF->IsGlobalInSmallSection(GO, getTargetMachine())) {
            SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,
                                                     Cpu0II::MO_GPREL);
            SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                             DAG.getVTList(MVT::i32), GA);
            SDValue GPReg = DAG.getRegister(Cpu0::GP, MVT::i32);
            return DAG.getNode(ISD::ADD, DL, MVT::i32, GPReg, GPRelNode);
        }
    }
```

```
...
}
...
}
```

```
...
Type-legalized selection DAG: BB#0 '_Z3funv:entry'
```

(continues on next page)

(continued from previous page)

SelectionDAG has 12 nodes:

```
...
0x7fc5f382cc10: <multiple use>
0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,
0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=-3]

0x7fc5f382d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

0x7fc5f382cc10: <multiple use>
0x7fc5f382d110: i32,ch = load 0x7fc5f382cf10, 0x7fc5f382d010,
0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=-3]
```

Legalized selection DAG: BB#0 '_Z3funv:entry'

SelectionDAG has 15 nodes:

```
...
0x7fc5f382cc10: <multiple use>
0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,
0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=8]

0x7fc5f382d710: i32 = register %GP

0x7fc5f382d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=4]

0x7fc5f382d610: i32 = Cpu0ISD::GPRel 0x7fc5f382d310

0x7fc5f382d810: i32 = add 0x7fc5f382d710, 0x7fc5f382d610

0x7fc5f382cc10: <multiple use>
0x7fc5f382d110: i32,ch = load 0x7fc5f382cf10, 0x7fc5f382d810,
0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=9]
...
```

Finally, the pattern defined in Cpu0InstrInfo.td as the following will translate DAG (add register %GP Cpu0ISD::GPRel<gI offset>) into Cpu0 instruction as below.

Ibdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
def Cpu0GPRel : SDNode<"Cpu0ISD::GPRel", SDTIntUnaryOp>;
```

```
// gp_rel relocs
let Predicates = [Ch6_1] in {
def : Pat<(add CPUREgs:$gp, (Cpu0GPRel tglobaladdr:$in)),
            (ORi CPUREgs:$gp, tglobaladdr:$in)>;
}
```

```
ori $2, $gp, %gp_rel(gI)
...
```

"Pat<(add CPUREgs:\$gp, (Cpu0GPRel tglobaladdr:\$in)), (ADD CPUREgs:\$gp, (ORi ZERO, tglobaladdr:\$in))>" will translate (add register %GP Cpu0ISD::GPRel tglobaladdr) into (add \$gp, (ori ZERO, tglobaladdr)).

In this mode, the \$gp content is assigned at compile/link time, changed only at program be loaded, and is fixed during

the program running; on the contrary, when -relocation-model=pic the \$gp can be changed during program running. For this example code, if \$gp is assigned to the start address of .sdata by loader when program ch6_1.cpu0.s is loaded, then linker can calculate %gp_rel(gI) (= the relative address distance between gI and start of .sdata section). Which meaning this relocation record can be solved at link time, that's why it is static mode.

In this mode, we reserve \$gp to a specific fixed address of the program is loaded. As a result, the \$gp cannot be allocated as a general purpose for variables. The following code tells llvm never allocate \$gp for variables.

Ibdex/chapters/Chapter6_1/Cpu0Subtarget.cpp

```
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                           StringRef FS, bool little,
                           const Cpu0TargetMachine &_TM) :

// Set UseSmallSection.
UseSmallSection = UseSmallSectionOpt;
Cpu0ReserveGP = ReserveGPOpt;
Cpu0NoCpload = NoCploadOpt;
#ifndef ENABLE_GPRESTORE
if (!_TM.isPositionIndependent() && !UseSmallSection && !Cpu0ReserveGP)
    FixGlobalBaseReg = false;
else
#endif
    FixGlobalBaseReg = true;

}
```

Ibdex/chapters/Chapter6_1/Cpu0RegisterInfo.cpp

```
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
// @getReservedRegs body {

#ifndef ENABLE_GPRESTORE //1
const Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
// Reserve GP if globalBaseRegFixed()
if (Cpu0FI->globalBaseRegFixed())
#endif
    Reserved.set(Cpu0::GP);

...
}
```

6.3 PIC Mode

6.3.1 sdata or sbss

Option llc -relocation-model=pic -cpu0-use-small-section=true will generate the following instructions.

```
...
.set      noreorder
```

(continues on next page)

(continued from previous page)

```
.cupload      $6
.set         nomacro
...
ld  $2, %got(gI)($gp)
ld  $2, 0($2)

...
.type        gStart,@object          # @gStart
.data
.globl       gStart
.align       2
gStart:
.4byte      2                      # 0x2
.size        gStart, 4

.type        gI,@object           # @gI
.globl       gI
.align       2
gI:
.4byte      100                 # 0x64
.size        gI, 4
```

The following code fragment of Cpu0AsmPrinter.cpp will emit **.cupload** asm pseudo instruction at function entry point as below.

Ibdex/chapters/Chapter6_1/Cpu0MachineFunction.h

```
/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
    Cpu0FunctionInfo(MachineFunction& MF)
        : MF(MF),
        VarArgsFrameIndex(0),
        GlobalBaseReg(0),
        globalBaseRegFixed() const,
        globalBaseRegSet() const,
        getGlobalBaseReg();
    /// GlobalBaseReg - keeps track of the virtual register initialized for
    /// use as the global base register. This is used for PIC in some PIC
    /// relocation models.
    unsigned GlobalBaseReg;
    int GPFI; // Index of the frame object for restoring $gp
};
```

Ibdex/chapters/Chapter6_1/Cpu0MachineFunction.cpp

```
bool Cpu0FunctionInfo::globalBaseRegFixed() const {
    return FixGlobalBaseReg;
}

bool Cpu0FunctionInfo::globalBaseRegSet() const {
    return GlobalBaseReg;
}

unsigned Cpu0FunctionInfo::getGlobalBaseReg() {
    return GlobalBaseReg = Cpu0::GP;
}
```

Ibdex/chapters/Chapter6_1/Cpu0AsmPrinter.cpp

```
/// EmitFunctionBodyStart - Targets can override this to emit stuff before
/// the first basic block in the function.
void Cpu0AsmPrinter::emitFunctionBodyStart() {

    bool EmitCPLoad = (MF->getTarget().getRelocationModel() == Reloc::PIC_) &&
        Cpu0FI->globalBaseRegSet() &&
        Cpu0FI->globalBaseRegFixed();
    if (Cpu0NoCupload)
        EmitCPLoad = false;

    // Emit .cupload directive if needed.
    if (EmitCPLoad)
        OutStreamer->emitRawText(StringRef("\t.cupload\t$t9"));

} else if (EmitCPLoad) {
    SmallVector<MCInst, 4> MCInsts;
    MCInstLowering.LowerCPLOAD(MCInsts);
    for (SmallVector<MCInst, 4>::iterator I = MCInsts.begin();
          I != MCInsts.end(); ++I)
        OutStreamer->emitInstruction(*I, getSubtargetInfo());
}

}

...
.set      noreorder
.cupload  $6
.set      nomacro
...
```

The **.cupload** is the assembly directive (macro) which will expand to several instructions. Issue **.cupload** before **.set nomacro** since the **.set nomacro** option causes the assembler to print a warning message whenever an assembler operation generates more than one machine language instruction, reference Mips ABI².

Following code will expand **.cupload** into machine instructions as below. “0fa00000 09aa0000 13aa6000” is the **.cupload** machine instructions displayed in comments of **Cpu0MCInstLower.cpp**.

² <http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf>

Ibdex/chapters/Chapter6_1/Cpu0MCInstLower.h

```

/// This class is used to lower an MachineInstr into an MCInst.
class LLVM_LIBRARY_VISIBILITY Cpu0MCInstLower {

    void LowerCPOLOAD(SmallVector<MCInst, 4>& MCInsts);

private:
    MCOperand LowerSymbolOperand(const MachineOperand &MO,
                                 MachineOperandType MOTy, unsigned Offset) const;

    ...
}

```

Ibdex/chapters/Chapter6_1/Cpu0MCInstLower.cpp

```

// Lower ".cupload $reg" to
// "lui $gp, %hi(_gp_disp)"
// "addiu $gp, $gp, %lo(_gp_disp)"
// "addu $gp, $gp, $t9"
void Cpu0MCInstLower::LowerCPOLOAD(SmallVector<MCInst, 4>& MCInsts) {
    MCOperand GPRReg = MCOperand::createReg(Cpu0::GP);
    MCOperand T9Reg = MCOperand::createReg(Cpu0::T9);
    StringRef SymName("_gp_disp");
    const MCSymbol *Sym = Ctx->getOrCreateSymbol(SymName);
    const Cpu0MCEExpr *MCSSym;

    MCSym = Cpu0MCEExpr::create(Sym, Cpu0MCEExpr::CEK_ABS_HI, *Ctx);
    MCOperand SymHi = MCOperand::createExpr(MCSym);
    MCSym = Cpu0MCEExpr::create(Sym, Cpu0MCEExpr::CEK_ABS_LO, *Ctx);
    MCOperand SymLo = MCOperand::createExpr(MCSym);

    MCInsts.resize(3);

    CreateMCInst(MCInsts[0], Cpu0::LUI, GPRReg, SymHi);
    CreateMCInst(MCInsts[1], Cpu0::ORI, GPRReg, GPRReg, SymLo);
    CreateMCInst(MCInsts[2], Cpu0::ADD, GPRReg, GPRReg, T9Reg);
}

```

```

118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch6_1.bc -o ch6_1.cpu0.o
118-165-76-131:input Jonathan$ gobjdump -s ch6_1.cpu0.o

```

```
ch6_1.cpu0.o:      file format elf32-big
```

```
Contents of section .text:
0000 0fa00000 0daa0000 13aa6000 ...
...
```

```
118-165-76-131:input Jonathan$ gobjdump -tr ch6_1.cpu0.o
...
```

(continues on next page)

(continued from previous page)

```
RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE           VALUE
00000000 UNKNOWN        _gp_disp
00000008 UNKNOWN        _gp_disp
00000020 UNKNOWN        gI
```

Note

// **Mips ABI: _gp_disp** After calculating the gp, a function allocates the local stack space and saves the gp on the stack, so it can be restored after subsequent function calls. In other words, the gp is a caller saved register.

...

_gp_disp represents the offset between the beginning of the function and the global offset table. Various optimizations are possible in this code example and the others that follow. For example, the calculation of gp need not be done for a position-independent function that is strictly local to an object module.

The _gp_disp as above is a relocation record, it means both the machine instructions 0da00000 (offset 0) and 0daa0000 (offset 8) which equal to assembly “ori \$gp, \$zero, %hi(_gp_disp)” and assembly “ori \$gp, \$gp, %lo(_gp_disp)”, respectively, are relocated records depend on _gp_disp. The loader or OS can caculate _gp_disp by (x - start address of .data) when load the dynamic function into memory x, and adjusts these two instructions offset correctly. Since shared function is loaded when this function is called, the relocation record “ld \$2, %got(gI)(\$gp)” cannot be resolved in link time. In spite of the reloation record is solved on load time, the name binding is static, since linker deliver the memory address to loader, and loader can solve this just by caculate the offset directly. The memory reference bind with the offset of _gp_disp at link time. The ELF relocation records will be introduced in Chapter ELF Support. So, don't worry if you don't quite understand it at this point.

The code fragment of lowerGlobalAddress() as the following corresponding option `llc -relocation-model=pic` will translate DAG (GlobalAddress<i32* @gI> 0) into (load EntryToken, (Cpu0ISD::Wrapper Register %GP, Target-GlobalAddress<i32* @gI> 0)) in stage “Legalized selection DAG” as below.

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
// This method creates the following nodes, which are necessary for
// computing a global symbol's address:
//
// (load (wrapper $gp, %got(sym)))
template<class NodeTy>
SDValue getAddrGlobal(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                      unsigned Flag, SDValue Chain,
                      const MachinePointerInfo &PtrInfo) const {
    SDLoc DL(N);
    SDValue Tgt = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                               getTargetNode(N, Ty, DAG, Flag));
    return DAG.getLoad(Ty, DL, Chain, Tgt, PtrInfo);
}
```

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
```

```
EVT Ty = Op.getValueType();
GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
const GlobalValue *GV = N->getGlobal();
```

```
const GlobalObject *GO = GV->getBaseObject();
if (GO && TLOF->IsGlobalInSmallSection(GO, getTargetMachine())) {
    SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,
                                             Cpu0II::MO_GPREL);
    SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                    DAG.getVTList(MVT::i32), GA);
    SDValue GPReg = DAG.getRegister(Cpu0::GP, MVT::i32);
    return DAG.getNode(ISD::ADD, DL, MVT::i32, GPReg, GPRelNode);
}
```

```
...
```

Ibdex/chapters/Chapter6_1/Cpu0ISelDAGToDAG.cpp

```
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
```

```
// on PIC code Load GA
if (Addr.getOpcode() == Cpu0ISD::Wrapper) {
    Base = Addr.getOperand(0);
    Offset = Addr.getOperand(1);
    return true;
}
```

```
...
```

```
...
Type-legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 12 nodes:
...
0x7fad7102cc10: <multiple use>
0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=-3]

0x7fad7102d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

0x7fad7102cc10: <multiple use>
0x7fad7102d110: i32,ch = load 0x7fad7102cf10, 0x7fad7102d010,
0x7fad7102cc10<LD4[@gI]> [ORD=3] [ID=-3]
...
Legalized selection DAG: BB#0 '_Z3funv:entry'
```

(continues on next page)

(continued from previous page)

```
SelectionDAG has 15 nodes:
0x7ff3c9c10b98: ch = EntryToken [ORD=1] [ID=0]
...
0x7fad7102cc10: <multiple use>
0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=8]

0x7fad70c10b98: <multiple use>
0x7fad7102d610: i32 = Register %GP

0x7fad7102d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=1]

0x7fad7102d710: i32 = Cpu0ISD::Wrapper 0x7fad7102d610, 0x7fad7102d310

0x7fad7102cc10: <multiple use>
0x7fad7102d810: i32, ch = load 0x7fad70c10b98, 0x7fad7102d710,
0x7fad7102cc10<LD4[<unknown>]>
0x7ff3ca02cc10: <multiple use>
0x7ff3ca02d110: i32, ch = load 0x7ff3ca02cf10, 0x7ff3ca02d810,
0x7ff3ca02cc10<LD4[@gI]> [ORD=3] [ID=9]
...
```

Finally, the pattern Cpu0 instruction **Id** defined before in `Cpu0InstrInfo.td` will translate DAG (load `EntryToken`, (`Cpu0ISD::Wrapper` `Register %GP`, `TargetGlobalAddress<i32* @gI> 0`)) into Cpu0 instruction as follows,

```
...
ld $2, %got(gI)($gp)
...
```

Remind in pic mode, Cpu0 uses “`.cupload`” and “`ld $2, %got(gI)($gp)`” to access global variable as Mips. It takes 4 instructions in both Cpu0 and Mips. The cost came from we didn’t assume that register `$gp` is always assigned to address `.sdata` and fixed there. Even we reserve `$gp` in this function, the `$gp` register can be changed at other functions. In last sub-section, the `$gp` is assumed to preserved at any function. If `$gp` is fixed during the run time, then “`.cupload`” can be removed here and have only one instruction cost in global variable access. The advantage of “`.cupload`” removing come from losing one general purpose register `$gp` which can be allocated for variables. In last sub-section, `.sdata` mode, we use “`.cupload`” removing since it is static link. In pic mode, the dynamic loading takes too much time. Remove “`.cupload`” with the cost of losing one general purpose register at all functions is not deserved here. The relocation records of “`.cupload`” from `llc -relocation-model=pic` can also be solved in link stage if we want to link this function by static link.

6.3.2 data or bss

The code fragment of `lowerGlobalAddress()` as the following corresponding option `llc -relocation-model=pic` will translate DAG (`GlobalAddress<i32* @gI> 0`) into (`load EntryToken, (Cpu0ISD::Wrapper (add Cpu0ISD::Hi<gI offset Hi16>, Register %GP), TargetGlobalAddress<i32* @gI> 0)`) in stage “Legalized selection DAG” as below.

[Index/chapters/Chapter6_1/Cpu0ISelLowering.h](#)

```
// This method creates the following nodes, which are necessary for
// computing a global symbol's address in large-GOT mode:
//
// (load (wrapper (add %hi(sym), $gp), %lo(sym)))
template<class NodeTy>
```

(continues on next page)

(continued from previous page)

```
SDValue getAddrGlobalLargeGOT(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                               unsigned HiFlag, unsigned LoFlag,
                               SDValue Chain,
                               const MachinePointerInfo &PtrInfo) const {
    SDLoc DL(N);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, Ty,
                             getTargetNode(N, Ty, DAG, HiFlag));
    Hi = DAG.getNode(ISD::ADD, DL, Ty, Hi, getGlobalReg(DAG, Ty));
    SDValue Wrapper = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, Hi,
                                   getTargetNode(N, Ty, DAG, LoFlag));
    return DAG.getLoad(Ty, DL, Chain, Wrapper, PtrInfo);
}
```

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
```

```
EVT Ty = Op.getValueType();
GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
const GlobalValue *GV = N->getGlobal();
```

```
const GlobalObject *GO = GV->getBaseObject();
if (GO && !TLOF->IsGlobalInSmallSection(GO, getTargetMachine()))
    return getAddrGlobalLargeGOT(
        N, Ty, DAG, Cpu0II::MO_GOT_HI16, Cpu0II::MO_GOT_LO16,
        DAG.getEntryNode(),
        MachinePointerInfo::getGOT(DAG.getMachineFunction()));
return getAddrGlobal(
    N, Ty, DAG, Cpu0II::MO_GOT, DAG.getEntryNode(),
    MachinePointerInfo::getGOT(DAG.getMachineFunction()));
}
```

```
...
Type-legalized selection DAG: BB#0 '_Z3funv:'
SelectionDAG has 10 nodes:
...
0x7fb77a02cd10: ch = store 0x7fb779c10a08, 0x7fb77a02ca10, 0x7fb77a02cb10,
0x7fb77a02cc10<ST4[%c]> [ORD=1] [ID=-3]
```

```
0x7fb77a02ce10: i32 = GlobalAddress<i32* @gI> 0 [ORD=2] [ID=-3]
```

```
0x7fb77a02cc10: <multiple use>
0x7fb77a02cf10: i32, ch = load 0x7fb77a02cd10, 0x7fb77a02ce10,
0x7fb77a02cc10<LD4[@gI]> [ORD=2] [ID=-3]
```

```
...
```

```
Legalized selection DAG: BB#0 '_Z3funv:'
SelectionDAG has 16 nodes:
```

```
...
0x7fb77a02cd10: ch = store 0x7fb779c10a08, 0x7fb77a02ca10, 0x7fb77a02cb10,
```

(continues on next page)

(continued from previous page)

```

0x7fb77a02cc10<ST4[%c]> [ORD=1] [ID=6]

0x7fb779c10a08: <multiple use>
    0x7fb77a02d110: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=19]

0x7fb77a02d410: i32 = Cpu0ISD::Hi 0x7fb77a02d110

0x7fb77a02d510: i32 = Register %GP

0x7fb77a02d610: i32 = add 0x7fb77a02d410, 0x7fb77a02d510

0x7fb77a02d710: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=20]

0x7fb77a02d810: i32 = Cpu0ISD::Wrapper 0x7fb77a02d610, 0x7fb77a02d710

0x7fb77a02cc10: <multiple use>
0x7fb77a02fe10: i32, ch = load 0x7fb779c10a08, 0x7fb77a02d810,
0x7fb77a02cc10<LD4[GOT]>

0x7fb77a02cc10: <multiple use>
0x7fb77a02cf10: i32, ch = load 0x7fb77a02cd10, 0x7fb77a02fe10,
0x7fb77a02cc10<LD4[@gI]> [ORD=2] [ID=7]
...

```

Finally, the pattern Cpu0 instruction **Id** defined before in `Cpu0InstrInfo.td` will translate DAG (load `EntryToken`, (`Cpu0ISD::Wrapper` (add `Cpu0ISD::Hi<gI offset Hi16>`, `Register %GP`), `Cpu0ISD::Lo<gI offset Lo16>`)) into Cpu0 instructions as below.

```

...
ori $2, $zero, %got_hi(gI)
shl $2, $2, 16
add $2, $2, $gp
ld $2, %got_lo(gI)($2)
...

```

The following code in `Cpu0InstrInfo.td` is needed for example input `ch8_2_select_global_pic.cpp`. Since `ch8_2_select_global_pic.cpp` uses LLVM IR `select`, it cannot be run at this point. It will be run in later Chapter Control flow statements.

[Index/chapters/Chapter6_1/Cpu0InstrInfo.td](#)

```
def Cpu0Wrapper : SDNode<"Cpu0ISD::Wrapper", SDTIntBinOp>;
```

```

let Predicates = [Ch6_1] in {
class WrapperPat<SDNode node, Instruction ORiOp, RegisterClass RC>:
    Pat<(Cpu0Wrapper RC:$gp, node:$in),
        (ORiOp RC:$gp, node:$in)>

def : WrapperPat<tglobaladdr, ORi, GPROut>;
}

```

Ibdex/input/ch8_2_select_global_pic.cpp

```

volatile int a1 = 1;
volatile int b1 = 2;

int gI1 = 100;
int gJ1 = 50;

int test_select_global_pic()
{
    if (a1 < b1)
        return gI1;
    else
        return gJ1;
}

```

6.4 Global variable print support

Above code is for global address DAG translation. Next, add the following code to Cpu0MCInstLower.cpp and Cpu0ISelLowering.cpp for global variable printing operand function.

Ibdex/chapters/Chapter6_1/Cpu0MCInstLower.cpp

```

MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                                MachineOperandType MOTy,
                                                unsigned Offset) const {
    MCSymbolRefExpr::VariantKind Kind = MCSymbolRefExpr::VK_None;
    Cpu0MCE Expr::Cpu0ExprKind TargetKind = Cpu0MCE Expr::CEK_None;
    const MCSymbol *Symbol;

    switch(MO.getTargetFlags()) {
    default:
        llvm_unreachable("Invalid target flag!");
    case Cpu0II::MO_NO_FLAG:
        break;

// Cpu0_GPREL is for llc -march=cpu0 -relocation-model=static -cpu0-islinux-
// format=false (global var in .sdata).
    case Cpu0II::MO_GPREL:
        TargetKind = Cpu0MCE Expr::CEK_GPREL;
        break;

    case Cpu0II::MO_GOT:
        TargetKind = Cpu0MCE Expr::CEK_GOT;
        break;
// ABS_HI and ABS_LO is for llc -march=cpu0 -relocation-model=static (global
// var in .data).
    case Cpu0II::MO_ABS_HI:
        TargetKind = Cpu0MCE Expr::CEK_ABS_HI;
        break;
    case Cpu0II::MO_ABS_LO:
        TargetKind = Cpu0MCE Expr::CEK_ABS_LO;
        break;
    }
}

```

(continues on next page)

(continued from previous page)

```

case Cpu0II::MO_GOT_HI16:
    TargetKind = Cpu0MCExpr::CEK_GOT_HI16;
    break;
case Cpu0II::MO_GOT_LO16:
    TargetKind = Cpu0MCExpr::CEK_GOT_LO16;
    break;
}

switch (MOTy) {
case MachineOperand::MO_GlobalAddress:
    Symbol = AsmPrinter.getSymbol(MO.getGlobal());
    Offset += MO.getOffset();
    break;

default:
    llvm_unreachable("<unknown operand type>");
}

const MCExpr *Expr = MCSymbolRefExpr::create(Symbol, Kind, *Ctx);

if (Offset) {
    // Assume offset is never negative.
    assert(Offset > 0);
    Expr = MCBinaryExpr::createAdd(Expr, MCConstantExpr::create(Offset, *Ctx),
                                   *Ctx);
}

if (TargetKind != Cpu0MCExpr::CEK_None)
    Expr = Cpu0MCExpr::create(TargetKind, Expr, *Ctx);

return MCOperand::createExpr(Expr);

}

```

```

MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                         unsigned offset) const {

```

```

MachineOperandType MOTy = MO.getType();

switch (MOTy) {

case MachineOperand::MO_GlobalAddress:
//@1
    return LowerSymbolOperand(MO, MOTy, offset);

    ...
}
...
}
```

The Cpu0MCExpr::printImpl() of Cpu0InstPrinter.cpp in last chapter is for global variable printing operand function too. The following function is for `llc -debug` this chapter DAG node name printing. It is added at Chapter3_1 already.

Ibdex/chapters/Chapter3_1/Cpu0ISelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
    switch (Opcode) {
    ...
    case Cpu0ISD::GPRel:           return "Cpu0ISD::GPRel";
    ...
    case Cpu0ISD::Wrapper:         return "Cpu0ISD::Wrapper";
    ...
}
```

OS is the output stream which output to the assembly file.

6.5 Summary

The global variable Instruction Selection for DAG translation is not like the ordinary IR node translation, it has static (absolute address) and pic mode. Backend deals this translation by create DAG nodes in function lowerGlobalAddress() which called by LowerOperation(). Function LowerOperation() takes care all Custom type of operation. Backend set global address as Custom operation by **"setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);"** in Cpu0TargetLowering() constructor. Different address mode create their own DAG list at run time. By setting the pattern Pat<> in Cpu0InstrInfo.td, the llvm can apply the compiler mechanism, pattern match, in the Instruction Selection stage.

There are three types for setXXXAction(), Promote, Expand and Custom. Except Custom, the other two maybe no need to coding. Here³ is the references.

As shown in this chapter, the global variable can be laid in .sdata/.sbss by option -cpu0-use-small-section=true. It is possible that the variables of small data section (16 bits addressable) are full out at link stage. When that happens, linker will highlights that error and forces the toolchain users to fix it. As the result, the toolchain user need to reconsider which global variables should be moved from .sdata/.sbss to .data/.bss by set option -cpu0-use-small-section=false in Makefile as follows,

Makefile

```
# Set the global variables declared in a.cpp to .data/.bss
llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=false \
-filetype=obj a.bc -o a.cpu0.o
# Set the global variables declared in b.cpp to .sdata/.sbss
llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=true \
-filetype=obj b.bc -o b.cpu0.o
```

The rule for global variables allocation is “set the small and frequent variables in small 16 addressable area”.

³ <http://llvm.org/docs/WritingAnLLVMBackend.html#the-selectiondag-legalize-phase>

OTHER DATA TYPE

- *Local Variable Pointer*
- *char, short int and bool*
- *long long*
- *float and double*
- *Array and struct support*
- *Vector type (SIMD) support*

Until now, we have only handled int and long types of 32-bit size. This chapter introduces other types, such as pointers and types that are not 32-bit, including bool, char, short int, and long long.

7.1 Local Variable Pointer

To support pointers to local variables, add the following code fragment to Cpu0InstrInfo.td and Cpu0InstPrinter.cpp:

Ibdex/chapters/Chapter7_1/Cpu0InstrInfo.td

```
def mem_ea : Operand<iPTR> {
    let PrintMethod = "printMemOperandEA";
    let MIOperandInfo = (ops GPROut, simm16);
    let EncoderMethod = "getMemEncoding";
}
```

```
class EffectiveAddress<string instr_asm, RegisterClass RC, Operand Mem> :
    FMem<0x09, (outs RC:$ra), (ins Mem:$addr),
        instr_asm, [(set RC:$ra, addr:$addr)], IIAlu>;
}
```

```
// FrameIndexes are legalized when they are operands from load/store
// instructions. The same not happens for stack address copies, so an
// add op with mem ComplexPattern is used and the stack address copy
// can be matched. It's similar to Sparc LEA_ADDri
def LEA_ADDiu : EffectiveAddress<"addiu\t$ra, $addr", CPUREgs, mem_ea> {
    let isCodeGenOnly = 1;
}
```

Ibdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.h

```
void printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O);
```

Ibdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
// The DAG data node, mem_ea of Cpu0InstrInfo.td, cannot be disabled by
// ch7_1, only opcode node can be disabled.
void Cpu0InstPrinter:::
printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O) {
    // when using stack locations for not load/store instructions
    // print the same way as all normal 3 operand instructions.
    printOperand(MI, opNum, O);
    O << ", ";
    printOperand(MI, opNum+1, O);
    return;
}
```

As noted in Cpu0InstPrinter.cpp, the printMemOperandEA function was added in an earlier Chapter 3.2 because the DAG data node mem_ea in Cpu0InstrInfo.td cannot be disabled by ch7_1_localpointer; only the opcode node can be disabled.

Run ch7_1_localpointer.cpp with the Chapter7_1/ directory, which supports pointers to local variables. The expected result is as follows:

Ibdex/input/ch7_1_localpointer.cpp

```
int test_local_pointer()
{
    int b = 3;

    int* p = &b;

    return *p;
}
```

```
118-165-66-82:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_localpointer.cpp -emit-llvm -o ch7_1_localpointer.bc
118-165-66-82:input Jonathan$ llvm-dis ch7_1_localpointer.bc -o -
...
; Function Attrs: nounwind
define i32 @_Z18test_local_pointerv() #0 {
    %b = alloca i32, align 4
    %p = alloca i32*, align 4
    store i32 3, i32* %b, align 4
    store i32* %b, i32** %p, align 4
    %1 = load i32** %p, align 4
    %2 = load i32* %1, align 4
    ret i32 %2
}
...

118-165-66-82:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
```

(continues on next page)

(continued from previous page)

```
-march=cpu0 -relocation-model=pic -filetype=asm
ch7_1_localpointer.bc -o -
...
    addiu $sp, $sp, -8
    addiu $2, $zero, 3
    st    $2, 4($fp)
    addiu $2, $fp, 4      // b address is 4($sp)
    st    $2, 0($fp)
    ld    $2, 4($fp)
    addiu $sp, $sp, 8
    ret   $lr
...
```

7.2 char, short int and bool

To support signed and unsigned char and short int, add the following code to Chapter7_1/:

Ibdex/chapters/Chapter7_1/Cpu0InstrInfo.td

```
def sextloadi16_a : AlignedLoad<sextloadi16>;
def zextloadi16_a : AlignedLoad<zextloadi16>;
def extloadi16_a : AlignedLoad<extloadi16>;
```

```
def truncstorei16_a : AlignedStore<truncstorei16>;
```

```
let Predicates = [Ch7_1] in {
def LB     : LoadM32<0x03, "lb", sextloadi8>;
def LBu    : LoadM32<0x04, "lbu", zextloadi8>;
def SB     : StoreM32<0x05, "sb", truncstorei8>;
def LH     : LoadM32<0x06, "lh", sextloadi16_a>;
def LHu    : LoadM32<0x07, "lhu", zextloadi16_a>;
def SH     : StoreM32<0x08, "sh", truncstorei16_a>;
}
```

Run Chapter7_1/ with ch7_1_char_in_struct.cpp to obtain the following result.

Ibdex/input/ch7_1_char_in_struct.cpp

```
struct Date
{
    short year;
    char month;
    char day;
    char hour;
    char minute;
    char second;
};

unsigned char b[4] = {'a', 'b', 'c', '\0'};
```

(continues on next page)

(continued from previous page)

```
int test_char()
{
    unsigned char a = b[1];
    char c = (char)b[1];
    Date date1 = {2012, (char)11, (char)25, (char)9, (char)40, (char)15};
    char m = date1.month;
    char s = date1.second;

    return 0;
}
```

```
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch7_1_char_in_struct.bc -o -
define i32 @_Z9test_charv() #0 {
    %a = alloca i8, align 1
    %c = alloca i8, align 1
    %date1 = alloca %struct.Date, align 2
    %m = alloca i8, align 1
    %s = alloca i8, align 1
    %1 = load i8* getelementptr inbounds ([4 x i8]* @b, i32 0, i32 1), align 1
    store i8 %1, i8* %a, align 1
    %2 = load i8* getelementptr inbounds ([4 x i8]* @b, i32 0, i32 1), align 1
    store i8 %2, i8* %c, align 1
    %3 = bitcast %struct.Date* %date1 to i8*
    call void @llvm.memcpy.p0i8.p0i8.i32(i8* %3, i8* bitcast ({ i16, i8, i8, i8,
    i8, i8, i8 }* @_ZZ9test_charvE5date1 to i8*), i32 8, i32 2, i1 false)
    %4 = getelementptr inbounds %struct.Date* %date1, i32 0, i32 1
    %5 = load i8* %4, align 1
    store i8 %5, i8* %m, align 1
    %6 = getelementptr inbounds %struct.Date* %date1, i32 0, i32 5
    %7 = load i8* %6, align 1
    store i8 %7, i8* %s, align 1
    ret i32 0
}

118-165-64-245:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_char_in_struct.cpp -emit-llvm -o ch7_1_char_in_struct.bc
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch7_1_char_in_struct.bc -o -
...
# BB#0:                                # %entry
addiu $sp, $sp, -24
lui $2, %got_hi(b)
addu $2, $2, $gp
ld $2, %got_lo(b)($2)
lbu $3, 1($2)
sb $3, 20($fp)
lbu $2, 1($2)
sb $2, 16($fp)
ld $2, %got($_ZZ9test_charvE5date1)($gp)
```

(continues on next page)

(continued from previous page)

```

addiu $2, $2, %lo($_ZZ9test_charvE5date1)
lhu $3, 4($2)
shl $3, $3, 16
lhu $4, 6($2)
or $3, $3, $4
st $3, 12($fp) // store hour, minute and second on 12($sp)
lhu $3, 2($2)
lhu $2, 0($2)
shl $2, $2, 16
or $2, $2, $3
st $2, 8($fp) // store year, month and day on 8($sp)
lbu $2, 10($fp) // m = date1.month;
sb $2, 4($fp)
lbu $2, 14($fp) // s = date1.second;
sb $2, 0($fp)
addiu $sp, $sp, 24
ret $lr
.set macro
.set reorder
.end _Z9test_charv
$tmp1:
.size _Z9test_charv, ($tmp1)-_Z9test_charv

.type b,@object          # @b
.data
.globl b
b:
.asciz "abc"
.size b, 4

.type $_ZZ9test_charvE5date1,@object # @_ZZ9test_charvE5date1
.section .rodata.cst8,"aM",@progbits,8
.align 1
$_ZZ9test_charvE5date1:
.2byte 2012             # 0x7dc
.byte 11                 # 0xb
.byte 25                 # 0x19
.byte 9                  # 0x9
.byte 40                 # 0x28
.byte 15                 # 0xf
.space 1
.size $_ZZ9test_charvE5date1, 8

```

Run Chapter7_1/ with ch7_1_char_short.cpp to obtain the following result.

Ibdex/input/ch7_1_char_short.cpp

```

int test_signed_char()
{
    char a = 0x80;
    int i = (signed int)a;
    i = i + 2; // i = (-128+2) = -126

```

(continues on next page)

(continued from previous page)

```
    return i;
}

int test_unsigned_char()
{
    unsigned char c = 0x80;
    unsigned int ui = (unsigned int)c;
    ui = ui + 2; // i = (128+2) = 130

    return (int)ui;
}

int test_signed_short()
{
    short a = 0x8000;
    int i = (signed int)a;
    i = i + 2; // i = (-32768+2) = -32766

    return i;
}

int test_unsigned_short()
{
    unsigned short c = 0x8000;
    unsigned int ui = (unsigned int)c;
    ui = ui + 2; // i = (32768+2) = 32770
    c = (unsigned short)ui;

    return (int)ui;
}
```

```
1-160-136-236:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch7_1_char_short.bc -o -
...
define i32 @_Z16test_signed_charv() #0 {
    ...
    %1 = load i8* %a, align 1
    %2 = sext i8 %1 to i32
    ...
}

; Function Attrs: nounwind
define i32 @_Z18test_unsigned_charv() #0 {
    ...
    %1 = load i8* %c, align 1
    %2 = zext i8 %1 to i32
    ...
}

; Function Attrs: nounwind
```

(continues on next page)

(continued from previous page)

```

define i32 @_Z17test_signed_shortv() #0 {
    ...
    %1 = load i16* %a, align 2
    %2 = sext i16 %1 to i32
    ...
}

; Function Attrs: nounwind
define i32 @_Z19test_unsigned_shortv() #0 {
    ...
    %1 = load i16* %c, align 2
    %2 = zext i16 %1 to i32
    ...
}

attributes #0 = { nounwind }

1-160-136-236:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_1_char_short.bc -o -
...
.globl _Z16test_signed_charv
...
lb $2, 4($sp)
...
.end _Z16test_signed_charv

.globl _Z18test_unsigned_charv
...
lbu $2, 4($sp)
...
.end _Z18test_unsigned_charv

.globl _Z17test_signed_shortv
...
lh $2, 4($sp)
...
.end _Z17test_signed_shortv

.globl _Z19test_unsigned_shortv
...
lhu $2, 4($sp)
...
.end _Z19test_unsigned_shortv
...

```

As shown, lb/lh instructions are used for signed byte/short types, while lbu/lhu are used for unsigned byte/short types. To efficiently support C type-casting and type-conversion features, Cpu0 provides the lb instruction, which converts a char to an int with a single instruction. The instructions lbu, lh, lhu, sb, and sh are applied to both signed and unsigned byte and short conversions. Their differences were explained in Chapter 2.

To support loading the bool type, add the following code:

Ibdex/chapters/Chapter7_1/Cpu0ISellowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
// Cpu0 does not have i1 type, so use i32 for
// setcc operations results (slt, sgt, ...).
setBooleanContents(ZeroOrOneBooleanContent);
setBooleanVectorContents(ZeroOrNegativeOneBooleanContent);

// Load extented operations for i1 types must be promoted
for (MVT VT : MVT::integer_valuetypes()) {
    setLoadExtAction(ISD::EXTLOAD, VT, MVT::i1, Promote);
    setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::i1, Promote);
    setLoadExtAction(ISD::SEXTLOAD, VT, MVT::i1, Promote);
}
```

...
}

The purpose of setBooleanContents() is as follows, but its details are not well understood. Without it, ch7_1_bool2.ll still works as shown below.

The IR input file ch7_1_bool2.ll is used for testing, as the C++ version requires flow control, which is not supported at this point. The file ch_run_backend.cpp includes a test fragment for bool, as shown below.

include/llvm/Target/TargetLowering.h

```
enum BooleanContent { // How the target represents true/false values.
    UndefinedBooleanContent, // Only bit 0 counts, the rest can hold garbage.
    ZeroOrOneBooleanContent, // All bits zero except for bit 0.
    ZeroOrNegativeOneBooleanContent // All bits equal to bit 0.
};

...
protected:
    /// setBooleanContents - Specify how the target extends the result of a
    /// boolean value from i1 to a wider type. See getBooleanContents.
    void setBooleanContents(BooleanContent Ty) { BooleanContents = Ty; }
    /// setBooleanVectorContents - Specify how the target extends the result
    /// of a vector boolean value from a vector of i1 to a wider type. See
    /// getBooleanContents.
    void setBooleanVectorContents(BooleanContent Ty) {
        BooleanVectorContents = Ty;
    }
```

Ibdex/input/ch7_1_bool2.ll

```
define zeroext i1 @verify_load_bool() #0 {
entry:
    %retval = alloca i1, align 1
    store i1 1, i1* %retval, align 1
```

(continues on next page)

(continued from previous page)

```
%0 = load i1, i1* %retval
ret i1 %0
}
```

```
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_1_bool2.ll -o -

.section .mdebug.abi32
.previous
.file "ch7_1_bool2.ll"
.text
.globl verify_load_bool
.align 2
.type verify_load_bool,@function
.ent verify_load_bool      # @verify_load_bool
verify_load_bool:
.cfi_startproc
.frame $sp,8,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:                      # %entry
addiu $sp, $sp, -8
$tmp1:
.cfi_def_cfa_offset 8
addiu $2, $zero, 1
sb $2, 7($sp)
addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
.end verify_load_bool
$tmp2:
.size verify_load_bool, ($tmp2)-verify_load_bool
.cfi_endproc
```

The ch7_1_bool.cpp file provides a bool test version for C. You can run it in Chapter8_1/ to obtain results similar to ch7_1_bool2.ll.

Index/input/ch7_1_bool.cpp

```
bool test_load_bool()
{
    int a = 1;

    if (a < 0)
        return false;

    return true;
}
```

Summary Table

Table 7.1: The C, IR, and DAG translation for char, short and bool translation (ch7_1_char_short.cpp and ch7_1_bool2.ll).

C	.bc	Optimized legalized selection DAG
char a =0x80;	%1 = load i8* %a, align 1	•
int i = (signed int)a;	%2 = sext i8 %1 to i32	load ..., <..., sext from i8>
unsigned char c = 0x80;	%1 = load i8* %c, align 1	•
unsigned int ui = (unsigned int)c;	%2 = zext i8 %1 to i32	load ..., <..., zext from i8>
short a =0x8000;	%1 = load i16* %a, align 2	•
int i = (signed int)a;	%2 = sext i16 %1 to i32	load ..., <..., sext from i16>
unsigned short c = 0x8000;	%1 = load i16* %c, align 2	•
unsigned int ui = (unsigned int)c;	%2 = zext i16 %1 to i32	load ..., <..., zext from i16>
c = (unsigned short)ui;	%6 = trunc i32 %5 to i16	•
•	store i16 %6, i16* %c, align 2	store ..., <..., trunc to i16>
return true;	store i1 1, i1* %retval, align 1	store ..., <..., trunc to i8>

Table 7.2: The backend translation for char, short and bool translation (ch7_1_char_short.cpp and ch7_1_bool2.ll).

Optimized legalized selection DAG	Cpu0	pattern in Cpu0InstrInfo.td
load ..., <..., sext from i8>	lb	LB : LoadM32<0x03, "lb", sextloadi8>;
load ..., <..., zext from i8>	lbu	LBu : LoadM32<0x04, "lbu", zextloadi8>;
load ..., <..., sext from i16>	lh	LH : LoadM32<0x06, "lh", sextloadi16_a>;
load ..., <..., zext from i16>	lhu	LHu : LoadM32<0x07, "lhu", zextloadi16_a>;
store ..., <..., trunc to i16>	sh	SH : StoreM32<0x08, "sh", truncstorei16_a>;
store ..., <..., trunc to i8>	sb	SB : StoreM32<0x05, "sb", truncstorei8>;

7.3 long long

Like MIPS, the long type in Cpu0 is 32-bit, while long long is 64-bit in C. To support long long, add the following code to Chapter7_1/:

Ibdex/chapters/Chapter7_1/Cpu0SEISelDAGToDAG.cpp

```
void Cpu0SEIDAGToDAGISel::selectAddESubE(unsigned MOp, SDValue InFlag,
                                         SDValue CmpLHS, const SDLoc &DL,
                                         SDNode *Node) const {
    unsigned Opc = InFlag.getOpcode(); (void)Opc;
    assert((((Opc == ISD::ADDC || Opc == ISD::ADDE) ||
             (Opc == ISD::SUBC || Opc == ISD::SUBLE)) &&
            "(ADD|SUB)E flag operand must come from (ADD|SUB)C/E insn");

    SDValue Ops[] = { CmpLHS, InFlag.getOperand(1) };
    SDValue LHS = Node->getOperand(0), RHS = Node->getOperand(1);
    EVT VT = LHS.getValueType();

    SDNode *Carry;
    if (Subtarget->hasCpu032II())
        Carry = CurDAG->getMachineNode(Cpu0::SLTu, DL, VT, Ops);
```

(continues on next page)

(continued from previous page)

```

else {
    SDNode *StatusWord = CurDAG->getMachineNode(Cpu0::CMP, DL, VT, Ops);
    SDValue Constant1 = CurDAG->getTargetConstant(1, DL, VT);
    Carry = CurDAG->getMachineNode(Cpu0::ANDi, DL, VT,
                                      SDValue(StatusWord, 0), Constant1);
}
SDNode *AddCarry = CurDAG->getMachineNode(Cpu0::ADDu, DL, VT,
                                             SDValue(Carry, 0), RHS);

CurDAG->SelectNodeTo(Node, MOp, VT, MVT::Glue, LHS, SDValue(AddCarry, 0));
}

```

```

bool Cpu0SEDAgToDAGISel::trySelect(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();
    SDLoc DL(Node);

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
    **/

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
    **/

    EVT NodeTy = Node->getValueType(0);
    unsigned MultOpc;

    switch(Opcode) {
    default: break;

```

```

        case ISD::SUBE: {
            SDValue InFlag = Node->getOperand(2);
            selectAddESubE(Cpu0::SUBu, InFlag, InFlag.getOperand(0), DL, Node);
            return true;
        }

        case ISD::ADDE: {
            SDValue InFlag = Node->getOperand(2);
            selectAddESubE(Cpu0::ADDu, InFlag, InFlag.getValue(0), DL, Node);
            return true;
        }

        /// Mul with two results
        case ISD::SMUL_LOHI:
        case ISD::UMUL_LOHI: {
            MultOpc = (Opcode == ISD::UMUL_LOHI ? Cpu0::MULTu : Cpu0::MULT);
            std::pair<SDNode*, SDNode*> LoHi =
                selectMULT(Node, MultOpc, DL, NodeTy, true, true);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
if (!SDValue(Node, 0).use_empty())
    ReplaceUses(SDValue(Node, 0), SDValue(LoHi.first, 0));

if (!SDValue(Node, 1).use_empty())
    ReplaceUses(SDValue(Node, 1), SDValue(LoHi.second, 0));

CurDAG->RemoveDeadNode(Node);
return true;
}

...
}
```

Ibdex/chapters/Chapter7_1/Cpu0ISelLowering.h

```
class Cpu0TargetLowering : public TargetLowering {

    bool isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const override;

    ...
}
```

Ibdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                         const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {

    // Handle i64 shl
    setOperationAction(ISD::SHL_PARTS,           MVT::i32,      Expand);
    setOperationAction(ISD::SRA_PARTS,           MVT::i32,      Expand);
    setOperationAction(ISD::SRL_PARTS,           MVT::i32,      Expand);

    ...
}
```

The additional code in Cpu0ISelLowering.cpp handles shift operations for long long (64-bit). Using the `<<` and `>>` operators on 64-bit variables generates DAG SHL_PARTS, SRA_PARTS, and SRL_PARTS, which manage 32-bit operands during LLVM DAG translation.

At this point, ch9_7.cpp, which includes 64-bit shift operations, cannot be executed. It will be verified in the later chapter Function Call.

Run Chapter7_1/ with ch7_1_longlong.cpp to obtain the following result.

Ibdex/input/ch7_1_longlong.cpp

```

long long test_longlong()
{
    long long a = 0x300000002;
    long long b = 0x100000001;
    int a1 = 0x3001000;
    int b1 = 0x2001000;

    long long c = a + b;    // c = 0x00000004,00000003
    long long d = a - b;    // d = 0x00000002,00000001
    long long e = a * b;    // e = 0x00000005,00000002
    long long f = (long long)a1 * (long long)b1; // f = 0x00060050,01000000

    long long g = ((-7 * 8) + 1) >> 4; // g = -55/16=-3.4375=-4

    return (c+d+e+f+g); // (0x0006005b,01000002) = (393307,16777218)
}

```

1-160-134-62:input Jonathan\$ clang -target mips-unknown-linux-gnu -c ch7_1_longlong.cpp -emit-llvm -o ch7_1_longlong.bc

1-160-134-62:input Jonathan\$ /Users/Jonathan/llvm/test/build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm ch7_1_longlong.bc -o -

...

BB#0:

```

        addiu $sp, $sp, -72
        st    $8, 68($fp)           # 4-byte Folded Spill
        addiu $2, $zero, 2
        st    $2, 60($fp)
        addiu $2, $zero, 3
        st    $2, 56($fp)
        addiu $2, $zero, 1
        st    $2, 52($fp)
        st    $2, 48($fp)
        lui   $2, 768
        ori   $2, $2, 4096
        st    $2, 44($fp)
        lui   $2, 512
        ori   $2, $2, 4096
        st    $2, 40($fp)
        ld    $2, 52($fp)
        ld    $3, 60($fp)
        addu $3, $3, $2
        ld    $4, 56($fp)
        ld    $5, 48($fp)
        st    $3, 36($fp)
        cmp   $sw, $3, $2
        andi $2, $sw, 1
        addu $2, $2, $5
        addu $2, $4, $2
        st    $2, 32($fp)

```

(continues on next page)

(continued from previous page)

```
ld    $2, 52($fp)
ld    $3, 60($fp)
subu $4, $3, $2
ld    $5, 56($fp)
ld    $t9, 48($fp)
st    $4, 28($fp)
cmp   $sw, $3, $2
andi $2, $sw, 1
addu $2, $2, $t9
subu $2, $5, $2
st    $2, 24($fp)
ld    $2, 52($fp)
ld    $3, 60($fp)
multu $3, $2
mflo $4
mfhi $5
ld    $t9, 56($fp)
ld    $7, 48($fp)
st    $4, 20($fp)
mul   $3, $3, $7
addu $3, $5, $3
mul   $2, $t9, $2
addu $2, $3, $2
st    $2, 16($fp)
ld    $2, 40($fp)
ld    $3, 44($fp)
mult  $3, $2
mflo $2
mfhi $4
st    $2, 12($fp)
st    $4, 8($fp)
ld    $5, 28($fp)
ld    $3, 36($fp)
addu $t9, $3, $5
ld    $7, 20($fp)
addu $8, $t9, $7
addu $3, $8, $2
cmp   $sw, $3, $2
andi $2, $sw, 1
addu $2, $2, $4
cmp   $sw, $t9, $5
st    $sw, 4($fp)          # 4-byte Folded Spill
cmp   $sw, $8, $7
andi $4, $sw, 1
ld    $5, 16($fp)
addu $4, $4, $5
ld    $sw, 4($fp)          # 4-byte Folded Reload
andi $5, $sw, 1
ld    $t9, 24($fp)
addu $5, $5, $t9
ld    $t9, 32($fp)
addu $5, $t9, $5
```

(continues on next page)

(continued from previous page)

```

addu $4, $5, $4
addu $2, $4, $2
ld $8, 68($fp)           # 4-byte Folded Reload
addiu $sp, $sp, 72
ret $lr
...

```

7.4 float and double

At this point, Cpu0 only supports integer instructions. For floating-point operations, the Cpu0 backend calls library functions to convert integers to floats, as follows:

Ibdex/input/ch7_1_fmul.c

```

/*
~/llvm/debug/build/bin/clang -target mips-unknown-linux-gnu -emit-llvm -S ch7_1_fmul.c
...
%mul = fmul float %0, %1

~/llvm/debug/build/bin/lld -march=mips ch7_1_fmul.ll -relocation-model=static -o -
...
v_log_f32_e32 v1, v0
v_mul_legacy_f32_e32 v0, v0, v1
v_exp_f32_e32 v0, v0

~/llvm/test/build/bin/lld -march=cpu0 ch7_1_fmul.ll -relocation-model=static -o -
...
jsub __mulsf3
*/


float ch7_1_fmul(float a, float b) {
    float c = a * b;
    return c;
}

```

Floating-point function calls for Cpu0 will be supported in the Function Call chapter. Due to hardware cost constraints, many CPUs do not include floating-point hardware instructions. Instead, they rely on library functions. MIPS separates floating-point operations into a dedicated co-processor for applications that require floating-point arithmetic.

To support the floating-point library (part of compiler-rt)², the following code is added to support clz and clo instructions. Although these instructions are implemented in compiler-rt, they are integer operations that improve floating-point application performance.

Ibdex/chapters/Chapter7_1/Cpu0InstrInfo.td

```

let Predicates = [Ch7_1] in {
// Count Leading Ones/Zeros in Word
class CountLeading0<bits<8> op, string instr_asm, RegisterClass RC>:
    FA<op, (outs GPROut:$ra), (ins RC:$rb),
    !strconcat(instr_asm, "\t$ra, $rb"),
}

```

(continues on next page)

² <http://jonathan2251.github.io/lbt/lib.html#compiler-rt>

(continued from previous page)

```
[ (set GPROut:$ra, (ctlz RC:$rb))], II_CLZ> {
let rc = 0;
let shamt = 0;
}

class CountLeading1<bits<8> op, string instr_asm, RegisterClass RC>:
FA<op, (outs GPROut:$ra), (ins RC:$rb),
!strconcat(instr_asm, "\t$ra, $rb"),
[(set GPROut:$ra, (ctlz (not RC:$rb)))] , II_CLO> {
let rc = 0;
let shamt = 0;
}
```

```
let Predicates = [Ch7_1] in {
/// Count Leading
def CLZ : CountLeading0<0x15, "clz", CPURegs>;
def CLO : CountLeading1<0x16, "clo", CPURegs>;
```

7.5 Array and struct support

LLVM uses `getelementptr` to represent array and struct types in C. For details, refer to¹.

For `ch7_1_globalstructoffset.cpp`, the LLVM IR is as follows:

Ibdex/input/ch7_1_globalstructoffset.cpp

```
struct Date
{
    int year;
    int month;
    int day;
};

Date date = {2012, 10, 12};
int a[3] = {2012, 10, 12};

int test_struct()
{
    int day = date.day;
    int i = a[1];

    return (i+day); // 10+12=22
}
```

```
// ch7_1_globalstructoffset.ll
; ModuleID = 'ch7_1_globalstructoffset.bc'
...
```

(continues on next page)

¹ <http://llvm.org/docs/LangRef.html#getelementptr-instruction>

(continued from previous page)

```
%struct.Date = type { i32, i32, i32 }

@date = global %struct.Date { i32 2012, i32 10, i32 12 }, align 4
@a = global [3 x i32] [i32 2012, i32 10, i32 12], align 4

; Function Attrs: nounwind
define i32 @_Z11test_structv() #0 {
    %day = alloca i32, align 4
    %i = alloca i32, align 4
    %1 = load i32* getelementptr inbounds (%struct.Date* @date, i32 0, i32 2), align 4
    store i32 %1, i32* %day, align 4
    %2 = load i32* getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1), align 4
    store i32 %2, i32* %i, align 4
    %3 = load i32* %i, align 4
    %4 = load i32* %day, align 4
    %5 = add nsw i32 %3, %4
    ret i32 %5
}
```

Run Chapter6_1/ with ch7_1_globalstructoffset.bc on static mode will get the incorrect asm file as follows,

```
1-160-134-62:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/
llc -march=cpu0 -relocation-model=static -filetype=asm
ch7_1_globalstructoffset.bc -o -
...
    lui $2, %hi(date)
    ori $2, $2, %lo(date)
    ld $2, 0($2) // the correct one is ld $2, 8($2)
...
```

For day = date.day, the correct instruction is **ld \$2, 8(\$2)**, not **ld \$2, 0(\$2)**, since date.day has an offset of 8 bytes (the date struct contains year and month before day). Use the debug option in llc to analyze this:

```
jonathantekiimac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -debug -relocation-model=static
-filetype=asm ch6_2.bc -o ch6_2.cpu0.static.s
...
==== main
Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 20 nodes:
0x7f7f5b02d210: i32 = undef [ORD=1]

    0x7f7f5ac10590: ch = EntryToken [ORD=1]

    0x7f7f5b02d010: i32 = Constant<0> [ORD=1]

    0x7f7f5b02d110: i32 = FrameIndex<0> [ORD=1]

    0x7f7f5b02d210: <multiple use>
0x7f7f5b02d310: ch = store 0x7f7f5ac10590, 0x7f7f5b02d010, 0x7f7f5b02d110,
0x7f7f5b02d210<ST4[%retval]> [ORD=1]
```

(continues on next page)

(continued from previous page)

```
0x7f7f5b02d410: i32 = GlobalAddress<%struct.Date* @date> 0 [ORD=2]

0x7f7f5b02d510: i32 = Constant<8> [ORD=2]

0x7f7f5b02d610: i32 = add 0x7f7f5b02d410, 0x7f7f5b02d510 [ORD=2]

0x7f7f5b02d210: <multiple use>
0x7f7f5b02d710: i32, ch = load 0x7f7f5b02d310, 0x7f7f5b02d610, 0x7f7f5b02d210
<LD4[getelementptr inbounds (%struct.Date* @date, i32 0, i32 2)]> [ORD=3]

0x7f7f5b02db10: i64 = Constant<4>

0x7f7f5b02d710: <multiple use>
0x7f7f5b02d710: <multiple use>
0x7f7f5b02d810: i32 = FrameIndex<1> [ORD=4]

0x7f7f5b02d210: <multiple use>
0x7f7f5b02d910: ch = store 0x7f7f5b02d710:1, 0x7f7f5b02d710, 0x7f7f5b02d810,
0x7f7f5b02d210<ST4[%day]> [ORD=4]

0x7f7f5b02da10: i32 = GlobalAddress<[3 x i32]* @a> 0 [ORD=5]

0x7f7f5b02dc10: i32 = Constant<4> [ORD=5]

0x7f7f5b02dd10: i32 = add 0x7f7f5b02da10, 0x7f7f5b02dc10 [ORD=5]

0x7f7f5b02d210: <multiple use>
0x7f7f5b02de10: i32, ch = load 0x7f7f5b02d910, 0x7f7f5b02dd10, 0x7f7f5b02d210
<LD4[getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1)]> [ORD=6]

...

Replacing.3 0x7f7f5b02dd10: i32 = add 0x7f7f5b02da10, 0x7f7f5b02dc10 [ORD=5]
With: 0x7f7f5b030010: i32 = GlobalAddress<[3 x i32]* @a> + 4

Replacing.3 0x7f7f5b02d610: i32 = add 0x7f7f5b02d410, 0x7f7f5b02d510 [ORD=2]
With: 0x7f7f5b02db10: i32 = GlobalAddress<%struct.Date* @date> + 8

Optimized lowered selection DAG: BB#0 'main:entry'
SelectionDAG has 15 nodes:
0x7f7f5b02d210: i32 = undef [ORD=1]

0x7f7f5ac10590: ch = EntryToken [ORD=1]

0x7f7f5b02d010: i32 = Constant<0> [ORD=1]

0x7f7f5b02d110: i32 = FrameIndex<0> [ORD=1]
```

(continues on next page)

(continued from previous page)

```

0x7f7f5b02d210: <multiple use>
0x7f7f5b02d310: ch = store 0x7f7f5ac10590, 0x7f7f5b02d010, 0x7f7f5b02d110,
0x7f7f5b02d210<ST4[%retval]> [ORD=1]

0x7f7f5b02db10: i32 = GlobalAddress<%struct.Date* @date> + 8

0x7f7f5b02d210: <multiple use>
0x7f7f5b02d710: i32, ch = load 0x7f7f5b02d310, 0x7f7f5b02db10, 0x7f7f5b02d210
<LD4[getelementptr inbounds (%struct.Date* @date, i32 0, i32 2)]> [ORD=3]

0x7f7f5b02d710: <multiple use>
0x7f7f5b02d710: <multiple use>
0x7f7f5b02d810: i32 = FrameIndex<1> [ORD=4]

0x7f7f5b02d210: <multiple use>
0x7f7f5b02d910: ch = store 0x7f7f5b02d710:1, 0x7f7f5b02d710, 0x7f7f5b02d810,
0x7f7f5b02d210<ST4[%day]> [ORD=4]

0x7f7f5b030010: i32 = GlobalAddress<[3 x i32]* @a> + 4

0x7f7f5b02d210: <multiple use>
0x7f7f5b02de10: i32, ch = load 0x7f7f5b02d910, 0x7f7f5b030010, 0x7f7f5b02d210
<LD4[getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1)]> [ORD=6]

...

```

The output reveals the DAG translation process. As shown, the DAG node for date.day (add GlobalAddress<[3 x i32]* @a> 0, Constant<8>) with three nodes is replaced by a single node GlobalAddress<%struct.Date* @date> + 8. The same applies to a[1].

This replacement occurs because TargetLowering.cpp::isOffsetFoldingLegal(...) returns true in llc -static static addressing mode. In Cpu0, the **ld** instruction format is **ld \$r1, offset(\$r2)**, meaning it loads the value at address(\$r2) + offset into \$r1. To correct this, override isOffsetFoldingLegal(...) as follows:

lib/CodeGen/SelectionDAG/TargetLowering.cpp

```

bool
TargetLowering::isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const {
    // Assume that everything is safe in static mode.
    if (getTargetMachine().getRelocationModel() == Reloc::Static)
        return true;

    // In dynamic-no-pic mode, assume that known defined values are safe.
    if (getTargetMachine().getRelocationModel() == Reloc::DynamicNoPIC &&
        GA &&
        !GA->getGlobal()->isDeclaration() &&
        !GA->getGlobal()->isWeakForLinker())
        return true;

    // Otherwise assume nothing is safe.
    return false;
}

```

Ibdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```
bool
Cpu0TargetLowering::isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const {
    // The Cpu0 target isn't yet aware of offsets.
    return false;
}
```

Additionally, add the following code to Cpu0ISelDAGToDAG.cpp:

When SelectAddr(...) in Cpu0ISelDAGToDAG.cpp is called, Addr represents the DAG node for date.day:

Ibdex/chapters/Chapter7_1/Cpu0ISelDAGToDAG.cpp

```
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {

    // Addresses of the form FI+const or FI|const
    if (CurDAG->isBaseWithConstantOffset(Addr)) {
        ConstantSDNode *CN = dyn_cast<ConstantSDNode>(Addr.getOperand(1));
        if (isInt<16>(CN->getSExtValue())) {

            // If the first operand is a FI, get the TargetFI Node
            if (FrameIndexSDNode *FIN = dyn_cast<FrameIndexSDNode>
                (Addr.getOperand(0)))
                Base = CurDAG->getTargetFrameIndex(FIN->getIndex(), ValTy);
            else
                Base = Addr.getOperand(0);

            Offset = CurDAG->getTargetConstant(CN->getZExtValue(), DL, ValTy);
            return true;
        }
    }
}

...
}
```

Recall that we have translated the DAG list for date.day (add GlobalAddress<[3 x i32]* @a> 0, Constant<8>) into

(add (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)), Constant<8>) by the following code in Cpu0ISelLowering.h.

Ibdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
// This method creates the following nodes, which are necessary for
// computing a symbol's address in non-PIC mode:
//
// (add %hi(sym), %lo(sym))
template<class NodeTy>
SDValue getAddrNonPIC(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
```

(continues on next page)

(continued from previous page)

```

SDLoc DL(N);
SDValue Hi = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_HI);
SDValue Lo = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_LO);
return DAG.getNode(ISD::ADD, DL, Ty,
                   DAG.getNode(Cpu0ISD::Hi, DL, Ty, Hi),
                   DAG.getNode(Cpu0ISD::Lo, DL, Ty, Lo));
}

```

add (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)), Constant<8>

Since Addr.getOpcode() = ISD::ADD, Addr.getOperand(0) = (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)), and Addr.getOperand(1).getOpcode() = ISD::Constant, we set Base to (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)) and Offset to Constant<8>. This ensures ld \$r1, 8(\$r2) is correctly generated in the Instruction Selection stage.

Run Chapter7_1/ with ch7_1_globalstructoffset.cpp to obtain the correct instruction.

```

...
    lui    $2, %hi(date)
    ori    $2, $2, %lo(date)
    ld     $2, 8($2) // correct
...

```

The ch7_1_localarrayinit.cpp is for local variable initialization test. The result as follows,

Ibdex/input/ch7_1_localarrayinit.cpp

```

int main()
{
    int a[3]={0, 1, 2};

    return 0;
}

```

```

118-165-79-206:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_localarrayinit.cpp -emit-llvm -o ch7_1_localarrayinit.bc
118-165-79-206:input Jonathan$ llvm-dis ch7_1_localarrayinit.bc -o -
...

```

```

define i32 @main() nounwind ssp {
entry:
%retval = alloca i32, align 4
%a = alloca [3 x i32], align 4
store i32 0, i32* %retval
%0 = bitcast [3 x i32]* %a to i8*
call void @_ZZ4mainE1a(i8* %0, i8* bitcast ([3 x i32]* @_ZZ4mainE1a to i8*), i32 12, i32 4, i1 false)
ret i32 0
}
; Function Attrs: nounwind
declare void @_ZZ4mainE1a(i8* nocapture, i8* nocapture, i32, i32, i1) #1

```

```
118-165-79-206:input Jonathan$ ~/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_1_localarrayinit.bc -o -
...
# BB#0:                                # %entry
    addiu $sp, $sp, -16
    addiu $2, $zero, 0
    st    $2, 12($fp)
    ld    $2, %got($_ZZ4mainE1a) ($gp)
    ori   $2, $2, %lo($_ZZ4mainE1a)
    ld    $3, 8($2)
    st    $3, 8($fp)
    ld    $3, 4($2)
    st    $3, 4($fp)
    ld    $2, 0($2)
    st    $2, 0($fp)
    addiu $sp, $sp, 16
    ret   $lr
...
.type $_ZZ4mainE1a,@object      # @_ZZ4mainE1a
.section     .rodata,"a",@progbits
.align       2
$_ZZ4mainE1a:
    .4byte    0                      # 0x0
    .4byte    1                      # 0x1
    .4byte    2                      # 0x2
.size $_ZZ4mainE1a, 12
```

7.6 Vector type (SIMD) support

Vector types are used when multiple primitive data are operated in parallel using a single instruction (SIMD)³. Mips supports the following llvm IRs “icmp slt” and “sext” for vector type, Cpu0 supports them either.

Vector types enable multiple primitive data operations in parallel using a single instruction (SIMD)³. MIPS supports **icmp slt** and **sext** LLVM IRs for vector types, which Cpu0 also supports.

Ibdex/input/ch7_1_vector.cpp

```
typedef long   vector8long   __attribute__((__vector_size__(32)));
typedef long   vector8short  __attribute__((__vector_size__(16)));


int test_cmplt_short() {
    volatile vector8short a0 = {0, 1, 2, 3};
    volatile vector8short b0 = {2, 2, 2, 4};
    volatile vector8short c0;
    c0 = a0 < b0; // c0[0] = -1 (since 0 < 2 is true), c0[1] = -1, c0[2] = 0 (since 2 < 2 is false), c0[3] = -1

    return (int)(c0[0]+c0[1]+c0[2]+c0[3]); // -3
```

(continues on next page)

³ <http://llvm.org/docs/LangRef.html#vector-type>

(continued from previous page)

```

}

int test_cmplt_long() {
    volatile vector8long a0 = {2, 2, 2, 2, 1, 1, 1, 1};
    volatile vector8long b0 = {1, 1, 1, 1, 2, 2, 2, 2};
    volatile vector8long c0;
    c0 = a0 < b0; // c0[0..3] = {0, 0, ...}, c0[4..7] = {-1, ...}

    return (c0[0]+c0[1]+c0[2]+c0[3]+c0[4]+c0[5]+c0[6]+c0[7]); // -4
}

```

```

118-165-79-206:118-165-79-206:

```

```

; Function Attrs: nounwind
define i32 @_Z16test_cmplt_shortv() #0 {
    %a0 = alloca <4 x i32>, align 16
    %b0 = alloca <4 x i32>, align 16
    %c0 = alloca <4 x i32>, align 16
    store volatile <4 x i32> <i32 0, i32 1, i32 2, i32 3>, <4 x i32>* %a0, align 16
    store volatile <4 x i32> <i32 2, i32 2, i32 2, i32 2>, <4 x i32>* %b0, align 16
    %1 = load volatile <4 x i32>, <4 x i32>* %a0, align 16
    %2 = load volatile <4 x i32>, <4 x i32>* %b0, align 16
    %3 = icmp slt <4 x i32> %1, %2
    %4 = sext <4 x i1> %3 to <4 x i32>
    store volatile <4 x i32> %4, <4 x i32>* %c0, align 16
    %5 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
    %6 = extractelement <4 x i32> %5, i32 0
    %7 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
    %8 = extractelement <4 x i32> %7, i32 1
    %9 = add nsw i32 %6, %8
    %10 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
    %11 = extractelement <4 x i32> %10, i32 2
    %12 = add nsw i32 %9, %11
    %13 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
    %14 = extractelement <4 x i32> %13, i32 3
    %15 = add nsw i32 %12, %14
    ret i32 %15
}

```

```

118-165-79-206:

```

(continues on next page)

(continued from previous page)

```
.file "ch7_1_vector.bc"
.globl _Z16test_cmplt_shortv
.p2align 2
.type _Z16test_cmplt_shortv,@function
.ent _Z16test_cmplt_shortv  # @_Z16test_cmplt_shortv
_Z16test_cmplt_shortv:
.frame $fp,48,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -48
addiu $2, $zero, 3
st $2, 44($sp)
addiu $2, $zero, 1
st $2, 36($sp)
addiu $2, $zero, 0
st $2, 32($sp)
addiu $2, $zero, 2
st $2, 40($sp)
st $2, 28($sp)
st $2, 24($sp)
st $2, 20($sp)
st $2, 16($sp)
ld $2, 32($sp)
ld $3, 44($sp)
ld $4, 40($sp)
ld $5, 36($sp)
ld $t9, 20($sp)
slt $5, $5, $t9
ld $t9, 24($sp)
slt $4, $4, $t9
ld $t9, 28($sp)
slt $3, $3, $t9
shl $3, $3, 31
sra $3, $3, 31
ld $t9, 16($sp)
st $3, 12($sp)
shl $3, $4, 31
sra $3, $3, 31
st $3, 8($sp)
shl $3, $5, 31
sra $3, $3, 31
st $3, 4($sp)
slt $2, $2, $t9
shl $2, $2, 31
sra $2, $2, 31
st $2, 0($sp)
ld $2, 12($sp)
ld $2, 8($sp)
ld $2, 4($sp)
ld $2, 0($sp)
```

(continues on next page)

(continued from previous page)

```

ld  $3, 4($sp)
addu $2, $2, $3
ld  $3, 12($sp)
ld  $3, 8($sp)
ld  $3, 0($sp)
ld  $3, 8($sp)
addu $2, $2, $3
ld  $3, 12($sp)
ld  $3, 4($sp)
ld  $3, 0($sp)
ld  $3, 12($sp)
addu $2, $2, $3
ld  $3, 8($sp)
ld  $3, 4($sp)
ld  $3, 0($sp)
addiu $sp, $sp, 48
ret $lr
.set macro
.set reorder
.end _Z16test_cmplt_shortv
$func_end0:
.size _Z16test_cmplt_shortv, ($func_end0)-_Z16test_cmplt_shortv

.ident "Apple LLVM version 7.0.0 (clang-700.1.76)"
.section ".note.GNU-stack","",@progbits

```

Since `test_longlong_shift2()` in `ch7_1_vector.cpp` requires `storeRegToStack()` in `Cpu0SEInstInfo.cpp`, it cannot be verified at this point.

Ibdex/chapters/Chapter7_1/Cpu0ISelLowering.h

```

/// getSetCCResultType - get the ISD::SETCC result ValueType
EVT getSetCCResultType(const DataLayout &DL, LLVMContext &Context,
                      EVT VT) const override;

```

Ibdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```

EVT Cpu0TargetLowering::getSetCCResultType(const DataLayout &, LLVMContext &,
                                            EVT VT) const {
    if (!VT.isVector())
        return MVT::i32;
    return VT.changeVectorElementTypeToInteger();
}

```


CONTROL FLOW STATEMENTS

- *Pipeline architecture*
- *Control flow statement*
- *Long branch support*
- *Cpu0 Backend Optimization: Remove Useless JMP*
- *Fill Branch Delay Slot*
- *Conditional Instruction*
- *Phi node*
- *Phi In Optimization*
- *LLVM IR φ -Node Optimization Examples*
- *RISC CPU knowledge*

This chapter illustrates the corresponding IR for control flow statements, such as “**if-else**”, “**while**”, and “**for**” loop statements in C. It also explains how to translate these LLVM IR control flow statements into Cpu0 instructions in *Section I*.

In [section Cpu0 Backend Optimization: Remove Useless JMP](#), a control flow optimization pass for the backend is introduced. It serves as a simple tutorial program to help readers understand how to add and implement a backend optimization pass.

[Section Conditional Instruction](#) includes handling for conditional instructions, since Clang generates specific IR instructions, `select` and `select_cc`, to support control flow optimizations in the backend.

8.1 Pipeline architecture

The following figures are from the book *Computer Architecture: A Quantitative Approach, Fourth Edition*.

- **IF:** Instruction Fetch cycle
- **ID:** Instruction Decode/Register Fetch cycle
- **EX:** Execution/Effective Address cycle
- **MEM:** Memory Access cycle
- **WB:** Write-Back cycle

Multibanked caches (see Fig. 8.3) are used to increase cache bandwidth.

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Figure A.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

Fig. 8.1: 5 stages of pipeline

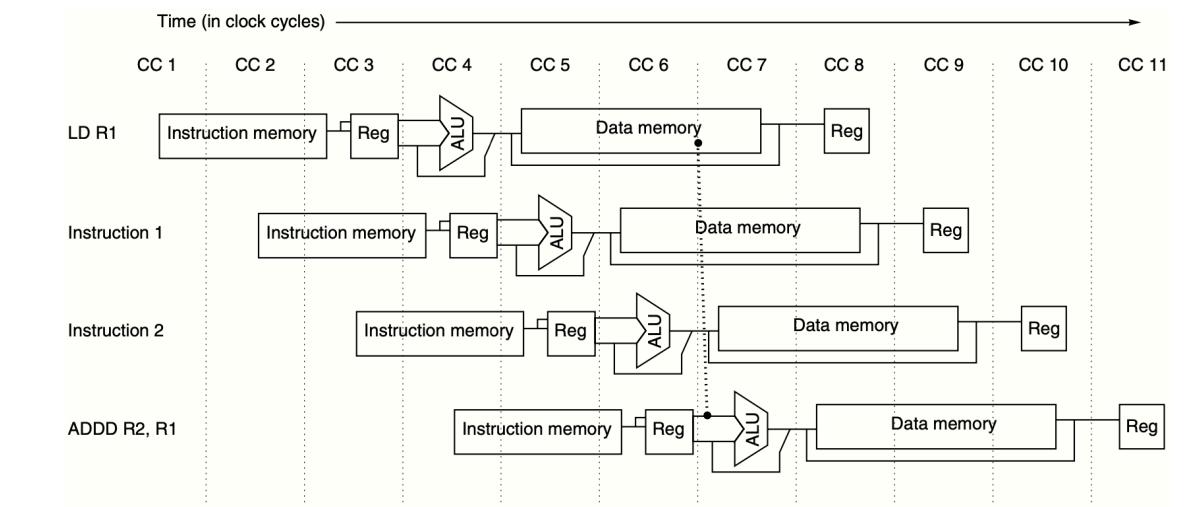


Figure A.38 The structure of the R4000 integer pipeline leads to a 2-cycle load delay. A 2-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

Fig. 8.2: Super pipeline

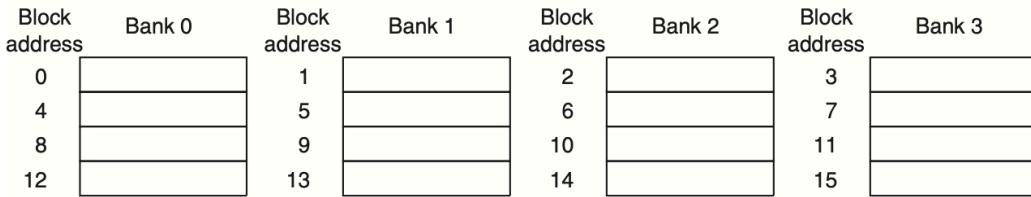


Figure 5.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Fig. 8.3: Interleaved cache banks

The block size in L1 cache is usually between 16 and 256 bytes. Equipped with multibanked caches, a system can support super-pipelined (Fig. 8.2) and superscalar (multi-issue pipeline) architectures, allowing the fetch of $(4 * \text{block_size} / \text{instruction_size})$ instructions per cycle.

8.2 Control flow statement

Running `ch8_1_1.cpp` with Clang will produce the following result:

lbdex/input/ch8_1_1.cpp

```
int test_ifctrl()
{
    unsigned int a = 0;

    if (a == 0) {
        a++; // a = 1
    }

    return a;
}
```

```
...
%0 = load i32* %a, align 4
%cmp = icmp eq i32 %0, 0
br i1 %cmp, label %if.then, label %if.end

; <label>:3:                                ; preds = %0
%1 = load i32* %a, align 4
%inc = add i32 %1, 1
store i32 %inc, i32* %a, align 4
br label %if.end
...
```

The “`icmp ne`” stands for *integer compare NotEqual*, “`slt`” stands for *Set Less Than*, and “`sle`” stands for *Set Less or Equal*.

Run version `Chapter8_1/` with the `llc -view-isel-dags` or `-debug` option. You will see that the `if` statement is

translated into the form:

```
br (brcond (%1, setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01
```

If we ignore `%1`, it simplifies to:

```
br (brcond (setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01
```

For explanation, the corresponding IR DAG is listed as follows:

```
Optimized legalized selection DAG: BB#0 '_Z11test_ifctrlv:entry'
SelectionDAG has 12 nodes:
...
t5: i32,ch = load<Volatile LD4[%a]> t4, FrameIndex:i32<0>, undef:i32
    t16: i32 = setcc t5, Constant:i32<0>, setne:ch
    t11: ch = brcond t5:1, t16, BasicBlock:ch<if.end 0x10305a338>
    t13: ch = br t11, BasicBlock:ch<if.then 0x10305a288>
```

We want to translate them into Cpu0 instruction DAGs as follows:

```
addiu %3, ZERO, Constant<c>
cmp %2, %3
jne BasicBlock_02
jmp BasicBlock_01
```

For the last IR `br`, we translate the unconditional branch (`br BasicBlock_01`) into `jmp BasicBlock_01` using the following pattern definition:

[Index/chapters/Chapter8_1/Cpu0InstrInfo.td](#)

```
// Unconditional branch, such as JMP
let Predicates = [Ch8_1] in {
class UncondBranch<bits<8> op, string instr_asm>:
    FJ<op>, (outs), (ins jmptarget:$addr),
        !strconcat(instr_asm, "\t$addr"), [(br bb:$addr)], IIBranch> {
    let isBranch = 1;
    let isTerminator = 1;
    let isBarrier = 1;
    let hasDelaySlot = 0;
}
}

...
def JMP      : UncondBranch<0x26, "jmp">;
```

The pattern `[(br bb:$imm24)]` in class `UncondBranch` is translated into the `jmp` machine instruction.

The translation for the pair of Cpu0 instructions, `cmp` and `jne`, has not been handled prior to this chapter.

To support this chained IR-to-machine-instruction translation, we define the following pattern:

Ibdex/chapters/Chapter8_1/Cpu0InstrInfo.td

```
// brcond patterns
multiclass BrcondPatsCmp<RegisterClass RC, Instruction JEQOp, Instruction JNEOp,
Instruction JLTop, Instruction JGTop, Instruction JLEOp, Instruction JGEOp,
Instruction CMPOp> {
...
def : Pat<(brcond (i32 (setne RC:$lhs, RC:$rhs)), bb:$dst),
(JNEOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
...
def : Pat<(brcond RC:$cond, bb:$dst),
(JNEOp (CMPOp RC:$cond, ZEROReg), bb:$dst)>;
...
}
```

Since the above *BrcondPats* pattern uses RC (Register Class) as an operand, the following *ADDiu* pattern defined in Chapter 2 will generate the instruction **addiu** before the **cmp** instruction for the first IR, **setcc(%2, Constant<c>, setne)**, as shown above.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
(ADDiu ZERO, imm:$in)>;
```

The definition of *BrcondPats* supports *setne*, *seteq*, *setlt*, and other register operand compares, as well as *setult*, *setugt*, and others for unsigned integer types.

In addition to *seteq* and *setne*, we define *setueq* and *setune* by referring to MIPS code, although we have not found how to generate *setune* IR from C language.

We have tried to define unsigned int types, but Clang still generates *setne* instead of *setune*.

The order of pattern search follows their order of appearance in the context.

The last pattern

```
(brcond RC:$cond, bb:$dst)
```

means branch to *\$dst* if *\$cond != 0*. Therefore, we set the corresponding translation to:

```
(JNEOp (CMPOp RC:$cond, ZEROReg), bb:$dst)
```

The *CMP* instruction sets the result to register *SW*, and then *JNE* checks the condition based on the *SW* status as shown in Fig. 8.4.

Since *SW* belongs to a different register class, the translation remains correct even if an instruction is inserted between *CMP* and *JNE*, as follows:

```
cmp %2, %3
addiu $r1, $r2, 3 // $r1 register never be allocated to $SW because in
// class ArithLogicI, GPROut is the output register
// class and the GPROut is defined without $SW in
// Cpu0RegisterInforGPROutForOther.td
jne BasicBlock_02
```

The reserved registers are set by the following function code that we defined previously:

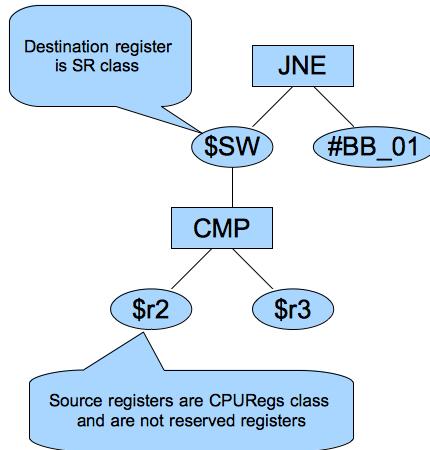


Fig. 8.4: JNE (CMP \$r2, \$r3),

Ibdex/chapters/Chapter3_1/Cpu0RegisterInfo.cpp

```

BitVector Cpu0RegisterInfo::  

getReservedRegs(const MachineFunction &MF) const {  

//@getReservedRegs body {  

    static const uint16_t ReservedCPURegs[] = {  

        Cpu0::ZERO, Cpu0::AT, Cpu0::SP, Cpu0::LR, /*Cpu0::SW, */Cpu0::PC  

    };  

    BitVector Reserved(getNumRegs());  
  

    for (unsigned I = 0; I < array_lengthof(ReservedCPURegs); ++I)  

        Reserved.set(ReservedCPURegs[I]);  
  

    return Reserved;  

}

```

Although the following definition in `Cpu0RegisterInfo.td` has no real effect on reserved registers, it is better to comment the reserved registers in it for readability.

Setting `SW` in both register classes `CPURegs` and `SR` allows access to `SW` by RISC instructions like `andi`, and also lets programmers use traditional assembly instructions like `cmp`.

The `copyPhysReg()` function is called when both `DestReg` and `SrcReg` belong to different register classes.

Ibdex/chapters/Chapter2/Cpu0RegisterInfo.td

```

//===== /  
  

def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add  

    // Reserved  

    ZERO, AT,  

    // Return Values and Arguments  

    V0, V1, A0, A1,  

    // Not preserved across procedure calls  

    T9, T0, T1,

```

(continues on next page)

(continued from previous page)

```
// Callee save
S0, S1,
// Reserved
GP, FP,
SP, LR, SW)>;
```

```
def SR      : RegisterClass<"Cpu0", [i32], 32, (add SW)>;
```

Ibdex/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

```
//=====
// Register Classes
//=====

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPUREgs, SW))>;
```

Chapter8_1/ includes support for control flow statements. Run it along with the following llc option to generate the object file. Dump its content using *gobjdump* or *hexdump* as shown below:

```
118-165-79-206:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
...
ld $4, 36($fp)
cmp $sw, $4, $3
jne $BB0_2
jmp $BB0_1
$BB0_1:                                # %if.then
    ld $4, 36($fp)
    addiu $4, $4, 1
    st $4, 36($fp)
$BB0_2:                                # %if.end
    ld $4, 32($fp)
...
```

```
118-165-79-206:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
-filetype=obj ch8_1_1.bc -o ch8_1_1.cpu0.o

118-165-79-206:input Jonathan$ hexdump ch8_1_1.cpu0.o
    // jmp offset is 0x10=16 bytes which is correct
0000080 ..... 10 43 00 00
0000090 31 00 00 10 36 00 00 00 .....
```

The immediate value of *jne* (opcode 0x31) is 16. The offset between *jne* and *\$BB0_2* is 20 bytes (5 words = 5 * 4 bytes). Suppose the address of *jne* is X, then the label *\$BB0_2* is located at X + 20.

Cpu0's instruction set is designed as a RISC CPU with a 5-stage pipeline, similar to the MIPS architecture. Cpu0 executes branch instructions at the decode stage, just like MIPS.

After the *jne* instruction is fetched, the Program Counter (PC) is updated to $X + 4$, since Cpu0 updates the PC during the fetch stage. The address of *\$BB0_2* is equal to $PC + 16$, because the *jne* instruction executes at the decode stage.

This can be listed and explained again as follows:

```

        // Fetch instruction stage for jne instruction. The fetch stage
        // can be divided into 2 cycles. First cycle fetch the
        // instruction. Second cycle adjust PC = PC+4.
jne $BB0_2 // Do jne compare in decode stage. PC = X+4 at this stage.
        // When jne immediate value is 16, PC = PC+16. It will fetch
        // X+20 which equal to label $BB0_2 instruction, ld $4, 32($sp).
nop
$BB0_1:                                # %if.then
    ld $4, 36($fp)
    addiu $4, $4, 1
    st $4, 36($fp)
$BB0_2:                                # %if.end
    ld $4, 32($fp)

```

If Cpu0 performs the “**jne**” instruction in the execution stage, then we should set $PC = PC + 12$, which equals the offset of $(\$BB0_2 - \text{jne})$ minus 8, in this example.

In reality, conditional branches are critical to CPU performance. According to benchmark data, on average, one branch instruction occurs every seven instructions.

The *cpu032I* requires two instructions for a conditional branch: *jne(cmp...)*. In contrast, *cpu032II* uses a single instruction (*bne*) as follows:

```

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
...
    cmp      $sw, $4, $3
    jne      $sw, $BB0_2
    jmp      $BB0_1
$BB0_1:

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
...
    bne      $4, $zero, $BB0_2
    jmp      $BB0_1
$BB0_1:

```

Besides the *brcond* explained in this section, the code above also includes the DAG opcode **br_jt** and label **JumpTable**, which appear during DAG translation for certain types of programs.

The file *ch8_1_ctrl.cpp* includes examples of control flow statements such as “**nested if**”, “**for loop**”, “**while loop**”, “**continue**”, “**break**”, and “**goto**”.

The file *ch8_1_br_jt.cpp* is used to test **br_jt** and **JumpTable** behavior.

The file *ch8_1_blockaddr.cpp* is for testing **blockaddress** and **indirectbr** instructions.

You can run these examples if you want to explore more control flow features.

List the control flow statements of C, corresponding LLVM IR, DAG nodes, and Cpu0 instructions in the following table.

Table 8.1: Control flow statements of C, IR, DAG and Cpu0 instructions

C	if, else, for, while, goto, switch, break
IR	(icmp + (eq, ne, sgt, sge, slt, sle)0 + br
DAG	(seteq, setne, setgt, setge, setlt, setle) + brcond,
•	(setueq, setune, setugt, setuge, setult, setule) + brcond
cpu032I	CMP + (JEQ, JNE, JGT, JGE, JLT, JLE)
cpu032II	(SLT, SLTu, SLTi, SLTiu) + (BEG, BNE)

8.3 Long branch support

As explained in the last section, *cpu032II* uses *beq* and *bne* instructions to improve performance. However, this change reduces the jump offset from 24 bits to 16 bits. If a program contains a branch that exceeds the 16-bit offset range, *cpu032II* will fail to generate valid code.

The MIPS backend has a solution to this limitation, and *Cpu0* adopts a similar approach.

To support long branches, the following code was added in Chapter8_1.

Ibdex/chapters/Chapter8_2/CMakeLists.txt

```
Cpu0LongBranch.cpp
```

Ibdex/chapters/Chapter8_2/Cpu0.h

```
FunctionPass *createCpu0LongBranchPass(Cpu0TargetMachine &TM);
```

Ibdex/chapters/Chapter8_2/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::  
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,  
                    SmallVectorImpl<MCFixup> &Fixups,  
                    const MCSubtargetInfo &STI) const {  
  
    if (Opcode == Cpu0::JMP || Opcode == Cpu0::BAL)  
  
    ...  
}
```

Ibdex/chapters/Chapter8_2/Cpu0AsmPrinter.h

```
bool isLongBranchPseudo(int Opcode) const;
```

Ibdex/chapters/Chapter8_2/Cpu0AsmPrinter.cpp

```
//- emitInstruction() must exists or will have run time error.  
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {  
  
    if (I->isPseudo() && !isLongBranchPseudo(I->getOpcode()))
```

```
...  
}
```

```
bool Cpu0AsmPrinter::isLongBranchPseudo(int Opcode) const {  
    return (Opcode == Cpu0::LONG_BRANCH_LUI  
            || Opcode == Cpu0::LONG_BRANCH_ADDiu);  
}
```

Ibdex/chapters/Chapter8_2/Cpu0InstrInfo.h

```
virtual unsigned getOppositeBranchOpc(unsigned Opc) const = 0;
```

Ibdex/chapters/Chapter8_2/Cpu0InstrInfo.td

```
let Predicates = [Ch8_2] in {  
// We need these two pseudo instructions to avoid offset calculation for long  
// branches. See the comment in file Cpu0LongBranch.cpp for detailed  
// explanation.  
  
// Expands to: lui $dst, %hi($tgt - $baltgt)  
def LONG_BRANCH_LUI : Cpu0Pseudo<(outs GPROut:$dst),  
    (ins jmptarget:$tgt, jmptarget:$baltgt), "", []>;  
  
// Expands to: addiu $dst, $src, %lo($tgt - $baltgt)  
def LONG_BRANCH_ADDiu : Cpu0Pseudo<(outs GPROut:$dst),  
    (ins GPROut:$src, jmptarget:$tgt, jmptarget:$baltgt), "", []>;  
}
```

```
let isBranch = 1, isTerminator = 1, isBarrier = 1,  
    hasDelaySlot = 0, Defs = [LR] in  
def BAL: FJ<0x3A, (outs), (ins jmptarget:$addr), "bal\t$t$addr", [], IIBranch>,  
    Requires<[HasSlt]>;
```

Ibdex/chapters/Chapter8_2/Cpu0LongBranch.cpp

```
===== Cpu0LongBranch.cpp - Emit long branches =====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// This pass expands a branch or jump instruction into a long branch if its  
// offset is too large to fit into its immediate field.  
//  
// FIXME: Fix pc-region jump instructions which cross 256MB segment boundaries.  
//=====//
```

(continues on next page)

(continued from previous page)

```
// In 9.0.0 rename MipsLongBranch.cpp to MipsBranchExpansion.cpp

#include "Cpu0.h"

#if CH >= CH8_2

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "Cpu0MachineFunction.h"
#include "Cpu0TargetMachine.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/CodeGen/TargetRegisterInfo.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/MathExtras.h"
#include "llvm/Target/TargetMachine.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-long-branch"

STATISTIC(LongBranches, "Number of long branches.");

static cl::opt<bool> ForceLongBranch(
    "force-cpu0-long-branch",
    cl::init(false),
    cl::desc("CPU0: Expand all branches to long format."),
    cl::Hidden);

namespace {
    typedef MachineBasicBlock::iterator Iter;
    typedef MachineBasicBlock::reverse_iterator ReverseIter;

    struct MBBInfo {
        uint64_t Size, Address;
        bool HasLongBranch;
        MachineInstr *Br;

        MBBInfo() : Size(0), HasLongBranch(false), Br(nullptr) {}
    };

    class Cpu0LongBranch : public MachineFunctionPass {

public:
    static char ID;
    Cpu0LongBranch(TargetMachine &tm)
        : MachineFunctionPass(ID), TM(tm), IsPIC(TM.isPositionIndependent()),
          ABI(static_cast<const Cpu0TargetMachine &>(TM).getABI()) {}

        StringRef getPassName() const override {
```

(continues on next page)

(continued from previous page)

```
        return "Cpu0 Long Branch";
    }

    bool runOnMachineFunction(MachineFunction &F) override;

private:
    void splitMBB(MachineBasicBlock *MBB);
    void initMBBInfo();
    int64_t computeOffset(const MachineInstr *Br);
    void replaceBranch(MachineBasicBlock &MBB, Iter Br, const DebugLoc &DL,
                       MachineBasicBlock *MBB0pnd);
    void expandToLongBranch(MBBInfo &Info);

    const TargetMachine &TM;
    MachineFunction *MF;
    SmallVector<MBBInfo, 16> MBBInfos;
    bool IsPIC;
    Cpu0ABIInfo ABI;
    unsigned LongBranchSeqSize;
};

char Cpu0LongBranch::ID = 0;
} // end of anonymous namespace

/// createCpu0LongBranchPass - Returns a pass that converts branches to long
/// branches.
FunctionPass *llvm::createCpu0LongBranchPass(Cpu0TargetMachine &tm) {
    return new Cpu0LongBranch(tm);
}

/// Iterate over list of Br's operands and search for a MachineBasicBlock
/// operand.
static MachineBasicBlock *getTargetMBB(const MachineInstr &Br) {
    for (unsigned I = 0, E = Br.getDesc().getNumOperands(); I < E; ++I) {
        const MachineOperand &MO = Br.getOperand(I);

        if (MO.isMBB())
            return MO.getMBB();
    }

    llvm_unreachable("This instruction does not have an MBB operand.");
}

// Traverse the list of instructions backwards until a non-debug instruction is
// found or it reaches E.
static ReverseIter getNonDebugInstr(ReverseIter B, const ReverseIter &E) {
    for (; B != E; ++B)
        if (!B->isDebugValue())
            return B;

    return E;
}
```

(continues on next page)

(continued from previous page)

```
// Split MBB if it has two direct jumps/branches.
void Cpu0LongBranch::splitMBB(MachineBasicBlock *MBB) {
    ReverseIter End = MBB->rend();
    ReverseIter LastBr = getNonDebugInstr(MBB->rbegin(), End);

    // Return if MBB has no branch instructions.
    if ((LastBr == End) ||
        (!LastBr->isConditionalBranch() && !LastBr->isUnconditionalBranch()))
        return;

    ReverseIter FirstBr = getNonDebugInstr(std::next(LastBr), End);

    // MBB has only one branch instruction if FirstBr is not a branch
    // instruction.
    if ((FirstBr == End) ||
        (!FirstBr->isConditionalBranch() && !FirstBr->isUnconditionalBranch()))
        return;

    assert(!FirstBr->isIndirectBranch() && "Unexpected indirect branch found.");

    // Create a new MBB. Move instructions in MBB to the newly created MBB.
    MachineBasicBlock *NewMBB =
        MF->CreateMachineBasicBlock(MBB->getBasicBlock());

    // Insert NewMBB and fix control flow.
    MachineBasicBlock *Tgt = getTargetMBB(*FirstBr);
    NewMBB->transferSuccessors(MBB);
    if (Tgt != getTargetMBB(*LastBr))
        NewMBB->removeSuccessor(Tgt, true);
    MBB->addSuccessor(NewMBB);
    MBB->addSuccessor(Tgt);
    MF->insert(std::next(MachineFunction::iterator(MBB)), NewMBB);

    NewMBB->splice(NewMBB->end(), MBB, LastBr.getReverse(), MBB->end());
}

// Fill MBBInfos.
void Cpu0LongBranch::initMBBInfo() {
    // Split the MBBs if they have two branches. Each basic block should have at
    // most one branch after this loop is executed.
    for (auto &MBB : *MF)
        splitMBB(&MBB);

    MF->RenumberBlocks();
    MBBInfos.clear();
    MBBInfos.resize(MF->size());

    const Cpu0InstrInfo *TII =
        static_cast<const Cpu0InstrInfo *>(MF->getSubtarget().getInstrInfo());
    for (unsigned I = 0, E = MBBInfos.size(); I < E; ++I) {
        MachineBasicBlock *MBB = MF->getBlockNumbered(I);
```

(continues on next page)

(continued from previous page)

```
// Compute size of MBB.
for (MachineBasicBlock::instr_iterator MI = MBB->instr_begin();
     MI != MBB->instr_end(); ++MI)
    MBBInfos[I].Size += TII->GetInstSizeInBytes(*MI);

// Search for MBB's branch instruction.
ReverseIter End = MBB->rend();
ReverseIter Br = getNonDebugInstr(MBB->rbegin(), End);

if ((Br != End) && !Br->isIndirectBranch() &&
    (Br->isConditionalBranch() || (Br->isUnconditionalBranch() && IsPIC)))
    MBBInfos[I].Br = &(*Br.getReverse());
}

}

// Compute offset of branch in number of bytes.
int64_t Cpu0LongBranch::computeOffset(const MachineInstr *Br) {
    int64_t Offset = 0;
    int ThisMBB = Br->getParent()->getNumber();
    int TargetMBB = getTargetMBB(*Br)->getNumber();

    // Compute offset of a forward branch.
    if (ThisMBB < TargetMBB) {
        for (int N = ThisMBB + 1; N < TargetMBB; ++N)
            Offset += MBBInfos[N].Size;

        return Offset + 4;
    }

    // Compute offset of a backward branch.
    for (int N = ThisMBB; N >= TargetMBB; --N)
        Offset += MBBInfos[N].Size;

    return -Offset + 4;
}

// Replace Br with a branch which has the opposite condition code and a
// MachineBasicBlock operand MBBOpnd.
void Cpu0LongBranch::replaceBranch(MachineBasicBlock &MBB, Iter Br,
                                    const DebugLoc &DL,
                                    MachineBasicBlock *MBBOpnd) {
    const Cpu0InstrInfo *TII = static_cast<const Cpu0InstrInfo *>(
        MBB.getParent()->getSubtarget().getInstrInfo());
    unsigned NewOpc = TII->getOppositeBranchOpc(Br->getOpcode());
    const MCInstrDesc &NewDesc = TII->get(NewOpc);

    MachineInstrBuilder MIB = BuildMI(MBB, Br, DL, NewDesc);

    for (unsigned I = 0, E = Br->getDesc().getNumOperands(); I < E; ++I) {
        MachineOperand &MO = Br->getOperand(I);
```

(continues on next page)

(continued from previous page)

```

if (!MO.isReg()) {
    assert(MO.isMBB() && "MBB operand expected.");
    break;
}

MIB.addReg(MO.getReg());
}

MIB.addMBB(MBBOpnd);

if (Br->hasDelaySlot()) {
    // Bundle the instruction in the delay slot to the newly created branch
    // and erase the original branch.
    assert(Br->isBundledWithSucc());
    MachineBasicBlock::instr_iterator II = Br.getInstrIterator();
    MIBundleBuilder(&*MIB).append((++II)->removeFromBundle());
}
Br->eraseFromParent();
}

// Expand branch instructions to long branches.
// TODO: This function has to be fixed for beqz16 and bnez16, because it
// currently assumes that all branches have 16-bit offsets, and will produce
// wrong code if branches whose allowed offsets are [-128, -126, ..., 126]
// are present.
void Cpu0LongBranch::expandToLongBranch(MBBInfo &I) {
    MachineBasicBlock::iterator Pos;
    MachineBasicBlock *MBB = I.Br->getParent(), *TgtMBB = getTargetMBB(*I.Br);
    DebugLoc DL = I.Br->getDebugLoc();
    const BasicBlock *BB = MBB->getBasicBlock();
    MachineFunction::iterator FallThroughMBB = ++MachineFunction::iterator(MBB);
    MachineBasicBlock *LongBrMBB = MF->CreateMachineBasicBlock(BB);
    const Cpu0Subtarget &Subtarget =
        static_cast<const Cpu0Subtarget &>(MF->getSubtarget());
    const Cpu0InstrInfo *TII =
        static_cast<const Cpu0InstrInfo *>(Subtarget.getInstrInfo());

    MF->insert(FallThroughMBB, LongBrMBB);
    MBB->replaceSuccessor(TgtMBB, LongBrMBB);

    if (IsPIC) {
        MachineBasicBlock *BalTgtMBB = MF->CreateMachineBasicBlock(BB);
        MF->insert(FallThroughMBB, BalTgtMBB);
        LongBrMBB->addSuccessor(BalTgtMBB);
        BalTgtMBB->addSuccessor(TgtMBB);

        unsigned BalOp = Cpu0::BAL;

        // $longbr:
        // addiu $sp, $sp, -8
        // st $lr, 0($sp)
        // lui $at, %hi($tgt - $baltgt)
    }
}

```

(continues on next page)

(continued from previous page)

```
// addiu $lr, $lr, %lo($tgt - $baltgt)
// bal $baltgt
// nop
// $baltgt:
// addu $at, $lr, $at
// addiu $sp, $sp, 8
// ld $lr, 0($sp)
// jr $at
// nop
// $fallthrough:
//

Pos = LongBrMBB->begin();

BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::ADDiu), Cpu0::SP)
    .addReg(Cpu0::SP).addImm(-8);
BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::ST)).addReg(Cpu0::LR)
    .addReg(Cpu0::SP).addImm(0);

// LUi and ADDiu instructions create 32-bit offset of the target basic
// block from the target of BAL instruction. We cannot use immediate
// value for this offset because it cannot be determined accurately when
// the program has inline assembly statements. We therefore use the
// relocation expressions %hi($tgt-$baltgt) and %lo($tgt-$baltgt) which
// are resolved during the fixup, so the values will always be correct.
//
// Since we cannot create %hi($tgt-$baltgt) and %lo($tgt-$baltgt)
// expressions at this point (it is possible only at the MC layer),
// we replace LUi and ADDiu with pseudo instructions
// LONG_BRANCH_LUi and LONG_BRANCH_ADDiu, and add both basic
// blocks as operands to these instructions. When lowering these pseudo
// instructions to LUi and ADDiu in the MC layer, we will create
// %hi($tgt-$baltgt) and %lo($tgt-$baltgt) expressions and add them as
// operands to lowered instructions.

BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::LONG_BRANCH_LUi), Cpu0::AT)
    .addMBB(TgtMBB).addMBB(BalTgtMBB);
BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::LONG_BRANCH_ADDiu), Cpu0::AT)
    .addReg(Cpu0::AT).addMBB(TgtMBB).addMBB(BalTgtMBB);
MIBundleBuilder(*LongBrMBB, Pos)
    .append(BuildMI(*MF, DL, TII->get(BalOp)).addMBB(BalTgtMBB));

Pos = BalTgtMBB->begin();

BuildMI(*BalTgtMBB, Pos, DL, TII->get(Cpu0::ADDu), Cpu0::AT)
    .addReg(Cpu0::LR).addReg(Cpu0::AT);
BuildMI(*BalTgtMBB, Pos, DL, TII->get(Cpu0::LD), Cpu0::LR)
    .addReg(Cpu0::SP).addImm(0);
BuildMI(*BalTgtMBB, Pos, DL, TII->get(Cpu0::ADDiu), Cpu0::SP)
    .addReg(Cpu0::SP).addImm(8);

MIBundleBuilder(*BalTgtMBB, Pos)
```

(continues on next page)

(continued from previous page)

```

.append(BuildMI(*MF, DL, TII->get(Cpu0::JR)).addReg(Cpu0::AT))
.append(BuildMI(*MF, DL, TII->get(Cpu0::NOP))));

assert(LongBrMBB->size() + BalttgtMBB->size() == LongBranchSeqSize);
} else {
    // $longbr:
    // jmp $tgt
    // nop
    // $fallthrough:
    //
    Pos = LongBrMBB->begin();
    LongBrMBB->addSuccessor(TgtMBB);
    MIBundleBuilder(*LongBrMBB, Pos)
        .append(BuildMI(*MF, DL, TII->get(Cpu0::JMP)).addMBB(TgtMBB))
        .append(BuildMI(*MF, DL, TII->get(Cpu0::NOP)));

    assert(LongBrMBB->size() == LongBranchSeqSize);
}

if (I.Br->isUnconditionalBranch()) {
    // Change branch destination.
    assert(I.Br->getDesc().getNumOperands() == 1);
    I.Br->RemoveOperand(0);
    I.Br->addOperand(MachineOperand::CreateMBB(LongBrMBB));
} else
    // Change branch destination and reverse condition.
    replaceBranch(*MBB, I.Br, DL, &*FallThroughMBB);
}

static void emitGPDisp(MachineFunction &F, const Cpu0InstrInfo *TII) {
    MachineBasicBlock &MBB = F.front();
    MachineBasicBlock::iterator I = MBB.begin();
    DebugLoc DL = MBB.findDebugLoc(MBB.begin());
    BuildMI(MBB, I, DL, TII->get(Cpu0::LUI), Cpu0::V0)
        .addExternalSymbol("_gp_disp", Cpu0II::MO_ABS_HI);
    BuildMI(MBB, I, DL, TII->get(Cpu0::ADDiu), Cpu0::V0)
        .addReg(Cpu0::V0).addExternalSymbol("_gp_disp", Cpu0II::MO_ABS_LO);
    MBB.removeLiveIn(Cpu0::V0);
}

bool Cpu0LongBranch::runOnMachineFunction(MachineFunction &F) {
    const Cpu0Subtarget &STI =
        static_cast<const Cpu0Subtarget &>(F.getSubtarget());
    const Cpu0InstrInfo *TII =
        static_cast<const Cpu0InstrInfo *>(STI.getInstrInfo());
    LongBranchSeqSize =
        !IsPIC ? 2 : 10;

    if (!STI.enableLongBranchPass())
        return false;
    if (IsPIC && static_cast<const Cpu0TargetMachine &>(TM).getABI().IsO32() &&
        F.getInfo<Cpu0FunctionInfo>()->globalBaseRegSet())

```

(continues on next page)

(continued from previous page)

```
emitGPDisp(F, TII);

MF = &F;
initMBBInfo();

SmallVectorImpl<MBBInfo>::iterator I, E = MBBInfos.end();
bool EverMadeChange = false, MadeChange = true;

while (MadeChange) {
    MadeChange = false;

    for (I = MBBInfos.begin(); I != E; ++I) {
        // Skip if this MBB doesn't have a branch or the branch has already been
        // converted to a long branch.
        if (!I->Br || I->HasLongBranch)
            continue;

        int ShVal = 4;
        int64_t Offset = computeOffset(I->Br) / ShVal;

        // Check if offset fits into 16-bit immediate field of branches.
        if (!ForceLongBranch && isInt<16>(Offset))
            continue;

        I->HasLongBranch = true;
        I->Size += LongBranchSeqSize * 4;
        ++LongBranches;
        EverMadeChange = MadeChange = true;
    }
}

if (!EverMadeChange)
    return true;

// Compute basic block addresses.
if (IsPIC) {
    uint64_t Address = 0;

    for (I = MBBInfos.begin(); I != E; Address += I->Size, ++I)
        I->Address = Address;
}

// Do the expansion.
for (I = MBBInfos.begin(); I != E; ++I)
    if (I->HasLongBranch)
        expandToLongBranch(*I);

MF->RenumberBlocks();

return true;
}
```

(continues on next page)

(continued from previous page)

```
#endif // #if CH >= CH8_2
```

Ibdex/chapters/Chapter8_2/Cpu0MCInstLower.h

```
MCOOperand createSub(MachineBasicBlock *BB1, MachineBasicBlock *BB2,
                      Cpu0MCEExpr::Cpu0ExprKind Kind) const;
void lowerLongBranchLUI(const MachineInstr *MI, MCInst &OutMI) const;
void lowerLongBranchADDiu(const MachineInstr *MI, MCInst &OutMI,
                          int Opcode,
                          Cpu0MCEExpr::Cpu0ExprKind Kind) const;
bool lowerLongBranch(const MachineInstr *MI, MCInst &OutMI) const;
```

Ibdex/chapters/Chapter8_2/Cpu0MCInstLower.cpp

```
MCOOperand Cpu0MCInstLower::createSub(MachineBasicBlock *BB1,
                                       MachineBasicBlock *BB2,
                                       Cpu0MCEExpr::Cpu0ExprKind Kind) const {
    const MCSymbolRefExpr *Sym1 = MCSymbolRefExpr::create(BB1->getSymbol(), *Ctx);
    const MCSymbolRefExpr *Sym2 = MCSymbolRefExpr::create(BB2->getSymbol(), *Ctx);
    const MCBinaryExpr *Sub = MCBinaryExpr::createSub(Sym1, Sym2, *Ctx);

    return MCOOperand::createExpr(Cpu0MCEExpr::create(Kind, Sub, *Ctx));
}

void Cpu0MCInstLower::
lowerLongBranchLUI(const MachineInstr *MI, MCInst &OutMI) const {
    OutMI.setOpcode(Cpu0::LUI);

    // Lower register operand.
    OutMI.addOperand(LowerOperand(MI->getOperand(0)));

    // Create %hi($tgt-$baltgt).
    OutMI.addOperand(createSub(MI->getOperand(1).getMBB(),
                               MI->getOperand(2).getMBB(),
                               Cpu0MCEExpr::CEK_ABS_HI));
}

void Cpu0MCInstLower::
lowerLongBranchADDiu(const MachineInstr *MI, MCInst &OutMI, int Opcode,
                     Cpu0MCEExpr::Cpu0ExprKind Kind) const {
    OutMI.setOpcode(Opcode);

    // Lower two register operands.
    for (unsigned I = 0, E = 2; I != E; ++I) {
        const MachineOperand &MO = MI->getOperand(I);
        OutMI.addOperand(LowerOperand(MO));
    }

    // Create %lo($tgt-$baltgt) or %hi($tgt-$baltgt).
    OutMI.addOperand(createSub(MI->getOperand(2).getMBB(),
                               MI->getOperand(3).getMBB(), Kind));
}
```

(continues on next page)

(continued from previous page)

```
}
```



```
bool Cpu0MCInstLower::lowerLongBranch(const MachineInstr *MI,
                                      MCInst &OutMI) const {
    switch (MI->getOpcode()) {
        default:
            return false;
        case Cpu0::LONG_BRANCH_LUI:
            lowerLongBranchLUI(MI, OutMI);
            return true;
        case Cpu0::LONG_BRANCH_ADDiu:
            lowerLongBranchADDiu(MI, OutMI, Cpu0::ADDiu,
                                 Cpu0MCEExpr::CEK_ABS_LO);
            return true;
    }
}
```

```
void Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) const {
```

```
    if (lowerLongBranch(MI, OutMI))
        return;
```

```
    ...  
}
```

Ibdex/chapters/Chapter8_2/Cpu0SEInstrInfo.h

```
unsigned getOppositeBranchOpc(unsigned Opc) const override;
```

Ibdex/chapters/Chapter8_2/Cpu0SEInstrInfo.cpp

```
/// getOppositeBranchOpc - Return the inverse of the specified
/// opcode, e.g. turning BEQ to BNE.
unsigned Cpu0SEInstrInfo::getOppositeBranchOpc(unsigned Opc) const {
    switch (Opc) {
        default:           llvm_unreachable("Illegal opcode!");
        case Cpu0::BEQ:   return Cpu0::BNE;
        case Cpu0::BNE:   return Cpu0::BEQ;
    }
}
```

Ibdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
void addPreEmitPass() override;
```

```
// Implemented by targets that want to run passes immediately before
// machine code is emitted. return true if -print-machineinstrs should
// print out the code after the passes.
void Cpu0PassConfig::addPreEmitPass() {
    Cpu0TargetMachine &TM = getCpu0TargetMachine();
```

```

addPass(createCpu0LongBranchPass(TM));
return;
}

```

The code of Chapter8_2 will compile the following example as follows:

Ibdex/input/ch8_2_longbranch.cpp

```

int test_longbranch()
{
    volatile int a = 2;
    volatile int b = 1;
    int result = 0;

    if (a < b)
        result = 1;
    return result;
}

```

```

118-165-78-10:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
-force-cpu0-long-branch ch8_2_longbranch.bc -o -
...
.text
.section .mdebug.abi032
.previous
.file "ch8_2_longbranch.bc"
.globl _Z15test_longbranchv
.align 2
.type _Z15test_longbranchv,@function
.ent _Z15test_longbranchv # @_Z15test_longbranchv
_Z15test_longbranchv:
.frame $fp,16,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
    addiu $sp, $sp, -16
    st $fp, 12($sp)          # 4-byte Folded Spill
    move $fp, $sp
    addiu $2, $zero, 1
    st $2, 8($fp)
    addiu $3, $zero, 2
    st $3, 4($fp)
    addiu $3, $zero, 0
    st $3, 0($fp)
    ld $3, 8($fp)
    ld $4, 4($fp)
    slt $3, $3, $4
    bne $3, $zero, .LBB0_3
    nop

```

(continues on next page)

(continued from previous page)

```
# BB#1:
    addiu $sp, $sp, -8
    st    $lr, 0($sp)
    lui   $1, %hi(.LBB0_4-.LBB0_2)
    addiu $1, $1, %lo(.LBB0_4-.LBB0_2)
    bal   .LBB0_2

.LBB0_2:
    addu $1, $lr, $1
    ld   $lr, 0($sp)
    addiu $sp, $sp, 8
    jr   $1
    nop

.LBB0_3:
    st   $2, 0($fp)

.LBB0_4:
    ld   $2, 0($fp)
    move $sp, $fp
    ld   $fp, 12($sp)           # 4-byte Folded Reload
    addiu $sp, $sp, 16
    ret  $lr
    nop
    .set macro
    .set reorder
    .end _Z15test_longbranchv

$func_end0:
    .size _Z15test_longbranchv, ($func_end0)-_Z15test_longbranchv
```

8.4 Cpu0 Backend Optimization: Remove Useless JMP

LLVM uses functional passes in both code generation and optimization. Following the three-tier architecture of a compiler, LLVM performs most optimizations in the middle tier, working on the LLVM IR in SSA form.

Beyond middle-tier optimizations, there are opportunities for backend-specific optimizations that depend on target architecture features. For example, “fill delay slot” in the Mips backend is a backend optimization used in pipelined RISC machines. You can port this technique from Mips if your backend also supports pipeline RISC with delay slots.

In this section, we implement the “remove useless jmp” optimization in the Cpu0 backend. This algorithm is simple and effective, making it a perfect tutorial for learning how to add an optimization pass. Through this example, you can understand how to introduce an optimization pass and implement a more complex optimization algorithm tailored for your backend in a real-world project.

The *Chapter8_2/* directory supports the “remove useless jmp” optimization algorithm and includes the added code as follows:

Ibdex/chapters/Chapter8_2/CMakeLists.txt

```
Cpu0DelUselessJMP.cpp
```

Ibdex/chapters/Chapter8_2/Cpu0.h

```
FunctionPass *createCpu0DelJmpPass(Cpu0TargetMachine &TM);
```

Ibdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
// Implemented by targets that want to run passes immediately before
// machine code is emitted. return true if -print-machineinstrs should
// print out the code after the passes.
void Cpu0PassConfig::addPreEmitPass() {
    Cpu0TargetMachine &TM = getCpu0TargetMachine();
```

```
    addPass(createCpu0DelJmpPass(TM));
```

```
}
```

Ibdex/chapters/Chapter8_2/Cpu0DelUselessJMP.cpp

```
===== Cpu0DelUselessJMP.cpp - Cpu0 DelJmp =====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// Simple pass to fills delay slots with useful instructions.
//
//=====

#include "Cpu0.h"
#if CH >= CH8_2

#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/Statistic.h"

using namespace llvm;

#define DEBUG_TYPE "del-jmp"

STATISTIC(NumDelJmp, "Number of useless jmp deleted");

```

(continues on next page)

(continued from previous page)

```
static cl::opt<bool> EnableDelJmp(
    "enable-cpu0-del-useless-jmp",
    cl::init(true),
    cl::desc("Delete useless jmp instructions: jmp 0."),
    cl::Hidden);

namespace {
    struct DelJmp : public MachineFunctionPass {
        static char ID;
        DelJmp(TargetMachine &tm)
            : MachineFunctionPass(ID) { }

        StringRef getPassName() const override {
            return "Cpu0 Del Useless jmp";
        }

        bool runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN);
        bool runOnMachineFunction(MachineFunction &F) override {
            bool Changed = false;
            if (EnableDelJmp) {
                MachineFunction::iterator FJ = F.begin();
                if (FJ != F.end())
                    FJ++;
                if (FJ == F.end())
                    return Changed;
                for (MachineFunction::iterator FI = F.begin(), FE = F.end();
                     FJ != FE; ++FI, ++FJ)
                    // In STL style, F.end() is the dummy BasicBlock() like '\0' in
                    // C string.
                    // FJ is the next BasicBlock of FI; When FI range from F.begin() to
                    // the PreviousBasicBlock of F.end() call runOnMachineBasicBlock().
                    Changed |= runOnMachineBasicBlock(*FI, *FJ);
            }
            return Changed;
        }
    };
    char DelJmp::ID = 0;
} // end of anonymous namespace

bool DelJmp::
runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN) {
    bool Changed = false;

    MachineBasicBlock::iterator I = MBB.end();
    if (I != MBB.begin())
        I--;           // set I to the last instruction
    else
        return Changed;

    if (I->getOpcode() == Cpu0::JMP && I->getOperand(0).getMBB() == &MBBN) {
```

(continues on next page)

(continued from previous page)

```

// I is the instruction of "jmp #offset=0", as follows,
//     jmp  $BB0_3
// $BB0_3:
//     ld  $4, 28($sp)
++NumDelJmp;
MBB.erase(I);           // delete the "JMP 0" instruction
Changed = true;         // Notify LLVM kernel Changed
}
return Changed;

}

/// createCpu0DelJmpPass - Returns a pass that DelJmp in Cpu0 MachineFunctions
FunctionPass *llvm::createCpu0DelJmpPass(Cpu0TargetMachine &tm) {
    return new DelJmp(tm);
}

#endif

```

As the above code shows, except for Cpu0DelUselessJMP.cpp, other files are changed to register the class DelJmp as a functional pass.

As the comment in the code explains, MBB is the current basic block, and MBBN is the next block. For each last instruction of every MBB, we check whether it is a JMP instruction and if its operand is the next basic block.

Using getMBB() in MachineOperand, you can get the address of the MBB. For more information about the member functions of MachineOperand, please check the file include/llvm/CodeGen/MachineOperand.h.

Now, let's run Chapter8_2/ with ch8_2_deluselessjmp.cpp for further explanation.

Ibdex/input/ch8_2_deluselessjmp.cpp

```

int test_DelUselessJMP()
{
    int a = 1; int b = -2; int c = 3;

    if (a == 0) {
        a++;
    }
    if (b == 0) {
        a = a + 3;
        b++;
    } else if (b < 0) {
        a = a + b;
        b--;
    }
    if (c > 0) {
        a = a + c;
        c++;
    }

    return a;
}

```

```

118-165-78-10:input Jonathan$ clang -target mips-unknown-linux-gnu
-c ch8_2_deluselessjmp.cpp -emit-llvm -o ch8_2_deluselessjmp.bc
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm -stats
ch8_2_deluselessjmp.bc -o -
...
    cmp    $sw, $4, $3
    jne    $sw, $BB0_2
    nop
# BB#1:
...
    cmp    $sw, $3, $2
    jlt    $sw, $BB0_8
    nop
# BB#7:
...
===== ... Statistics Collected ...
=====
...
2 del-jmp      - Number of useless jmp deleted
...

```

The terminal displays “Number of useless jmp deleted” when running with the `llc -stats` option, because we set the `STATISTIC(NumDelJmp, "Number of useless jmp deleted")` in the code. It deletes 2 jmp instructions from block # `BB#0` and `$BB0_6`.

You can verify this by using the `llc -enable-cpu0-del-useless-jmp=false` option to compare with the non-optimized version.

If you run with `ch8_1_1.cpp`, you will find 10 jmp instructions are deleted from 120 lines of assembly code. This implies about 8% improvement in both speed and code size¹.

8.5 Fill Branch Delay Slot

Cpu0 instruction set is designed to be a classical RISC pipeline machine. Classical RISC machines have many ideal features³⁴.

I modified the Cpu0 backend to support a 5-stage classical RISC pipeline with one delay slot, similar to some Mips models. (The original Cpu0 backend from its author used a 3-stage pipeline.)

With this change, the backend needs to insert a NOP instruction in the branch delay slot.

To keep this tutorial simple, the Cpu0 backend does not attempt to fill the delay slot with useful instructions for optimization. Readers can refer to `MipsDelaySlotFiller.cpp` for an example of how to do this kind of backend optimization.

The following code was added in `Chapter8_2` for NOP insertion in the branch delay slot.

¹ On a platform with cache and DRAM, the cache miss costs several tens of instruction cycles. Usually, the compiler engineers who work in the vendor of platform solution are spending much effort of trying to reduce the cache miss for speed. Reducing code size will decrease the cache miss frequency too.

³ See book Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

⁴ http://en.wikipedia.org/wiki/Classic_RISC_pipeline

Ibdex/chapters/Chapter8_2/CMakeLists.txt

```
Cpu0DelaySlotFiller.cpp
```

Ibdex/chapters/Chapter8_2/Cpu0.h

```
FunctionPass *createCpu0DelaySlotFillerPass (Cpu0TargetMachine &TM) ;
```

Ibdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
// Implemented by targets that want to run passes immediately before  
// machine code is emitted. return true if -print-machineinstrs should  
// print out the code after the passes.  
void Cpu0PassConfig::addPreEmitPass () {  
    Cpu0TargetMachine &TM = getCpu0TargetMachine ();
```

```
    addPass (createCpu0DelaySlotFillerPass (TM)) ;
```

```
}
```

Ibdex/chapters/Chapter8_2/Cpu0DelaySlotFiller.cpp

```
===== Cpu0DelaySlotFiller.cpp - Cpu0 Delay Slot Filler =====//  
//  
//           The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// Simple pass to fill delay slots with useful instructions.  
//  
//=====//  
  
#include "Cpu0.h"  
#if CH >= CH8_2  
  
#include "Cpu0InstrInfo.h"  
#include "Cpu0TargetMachine.h"  
#include "llvm/ADT/BitVector.h"  
#include "llvm/ADT/SmallPtrSet.h"  
#include "llvm/ADT/Statistic.h"  
#include "llvm/Analysis/AliasAnalysis.h"  
#include "llvm/Analysis/ValueTracking.h"  
#include "llvm/CodeGen/MachineBranchProbabilityInfo.h"  
#include "llvm/CodeGen/MachineFunctionPass.h"  
#include "llvm/CodeGen/MachineInstrBuilder.h"  
#include "llvm/CodeGen/PseudoSourceValue.h"  
#include "llvm/CodeGen/TargetInstrInfo.h"  
#include "llvm/Support/CommandLine.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/Target/TargetMachine.h"
#include "llvm/CodeGen/TargetRegisterInfo.h"

using namespace llvm;

#define DEBUG_TYPE "delay-slot-filler"

STATISTIC(FilledSlots, "Number of delay slots filled");

namespace {

    typedef MachineBasicBlock::iterator Iter;
    typedef MachineBasicBlock::reverse_iterator ReverseIter;

    class Filler : public MachineFunctionPass {
public:
    Filler(TargetMachine &tm)
        : MachineFunctionPass(ID) { }

    StringRef getPassName() const override {
        return "Cpu0 Delay Slot Filler";
    }

    bool runOnMachineFunction(MachineFunction &F) override {
        bool Changed = false;
        for (MachineFunction::iterator FI = F.begin(), FE = F.end();
             FI != FE; ++FI)
            Changed |= runOnMachineBasicBlock(*FI);
        return Changed;
    }
private:
    bool runOnMachineBasicBlock(MachineBasicBlock &MBB);

    static char ID;
};

char Filler::ID = 0;
} // end of anonymous namespace

static bool hasUnoccupiedSlot(const MachineInstr *MI) {
    return MI->hasDelaySlot() && !MI->isBundledWithSucc();
}

/// runOnMachineBasicBlock - Fill in delay slots for the given basic block.
/// We assume there is only one delay slot per delayed instruction.
bool Filler::runOnMachineBasicBlock(MachineBasicBlock &MBB) {
    bool Changed = false;
    const Cpu0Subtarget &STI = MBB.getParent()->getSubtarget<Cpu0Subtarget>();
    const Cpu0InstrInfo *TII = STI.getInstrInfo();

    for (Iter I = MBB.begin(); I != MBB.end(); ++I) {
        if (!hasUnoccupiedSlot(&*I))
            continue;
```

(continues on next page)

(continued from previous page)

```

++FilledSlots;
Changed = true;

// Bundle the NOP to the instruction with the delay slot.
BuildMI(MBB, std::next(I), I->getDebugLoc(), TII->get(Cpu0::NOP));
MIBundleBuilder(MBB, I, std::next(I, 2));
}

return Changed;
}

/// createCpu0DelaySlotFillerPass - Returns a pass that fills in delay
/// slots in Cpu0 MachineFunctions
FunctionPass *llvm::createCpu0DelaySlotFillerPass(Cpu0TargetMachine &tm) {
    return new Filler(tm);
}

#endif

```

To ensure the basic block label remains unchanged, the statement `MIBundleBuilder()` must be inserted after the `BuildMI(..., NOP)` statement in `Cpu0DelaySlotFiller.cpp`.

`MIBundleBuilder()` bundles the branch instruction and the NOP into a single instruction unit. The first part is the branch instruction, and the second part is the NOP.

Ibdex/chapters/Chapter3_2/Cpu0AsmPrinter.cpp

```

// emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {

    // Print out both ordinary instruction and boudle instruction
    MachineBasicBlock::const_instr_iterator I = MI->getIterator();
    MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

    do {

        if (I->isPseudo() && !isLongBranchPseudo(I->getOpcode()))

            llvm_unreachable("Pseudo opcode found in emitInstruction()");

        MCInst TmpInst0;
        // Call Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) to
        // extracts MCInst from MachineInstr.
        MCInstLowering.Lower(&*I, TmpInst0);
        OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
    } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}

```

In order to print the NOP, the `Cpu0AsmPrinter.cpp` of Chapter3_2 prints all bundle instructions in a loop. Without the loop, only the first part of the bundle instruction (branch instruction only) is printed.

In LLVM 3.1, the basic block label remains the same even if you do not bundle after it. But for some reason, this behavior changed in a later LLVM version, and now you need to perform the “bundle” to keep the block label unchanged in later

LLVM phases.

8.6 Conditional Instruction

[lbdex/input/ch8_2_select.cpp](#)

```
// The following files will generate IR select even compile with clang -O0.
int test_movx_1()
{
    volatile int a = 1;
    int c = 0;

    c = !a ? 1:3;

    return c;
}

int test_movx_2()
{
    volatile int a = 1;
    int c = 0;

    c = a ? 1:3;

    return c;
}
```

Run Chapter8_1 with ch8_2_select.cpp will get the following result.

```
114-37-150-209:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu
-c ch8_2_select.cpp -emit-llvm -o ch8_2_select.bc
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch8_2_select.bc -o -
...
; Function Attrs: nounwind uwtable
define i32 @_Z11test_movx_1v() #0 {
    %a = alloca i32, align 4
    %c = alloca i32, align 4
    store volatile i32 1, i32* %a, align 4
    store i32 0, i32* %c, align 4
    %1 = load volatile i32* %a, align 4
    %2 = icmp ne i32 %1, 0
    %3 = xor i1 %2, true
    %4 = select i1 %3, i32 1, i32 3
    store i32 %4, i32* %c, align 4
    %5 = load i32* %c, align 4
    ret i32 %5
}

; Function Attrs: nounwind uwtable
define i32 @_Z11test_movx_2v() #0 {
```

(continues on next page)

(continued from previous page)

```
%a = alloca i32, align 4
%c = alloca i32, align 4
store volatile i32 1, i32* %a, align 4
store i32 0, i32* %c, align 4
%1 = load volatile i32* %a, align 4
%2 = icmp ne i32 %1, 0
%3 = select i1 %2, i32 1, i32 3
store i32 %3, i32* %c, align 4
%4 = load i32* %c, align 4
ret i32 %4
}
...
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch8_2_select.bc -o -
...
LLVM ERROR: Cannot select: 0x39f47c0: i32 = select_cc ...
```

As shown in the above LLVM IR from ch8_2_select.bc, Clang generates a **select** IR for small control blocks (e.g., an if-statement containing only one assignment). This **select** IR is the result of an optimization for CPUs that support conditional instructions.

From the error message above, it is clear that the IR **select** is transformed into **select_cc** during the DAG (Directed Acyclic Graph) optimization stage.

Chapter8_2 supports **select** with the following code added and changed.

Ibdex/chapters/Chapter8_2/Cpu0InstrInfo.td

```
let Predicates = [Ch8_2] in {
include "Cpu0CondMov.td"
} // let Predicates = [Ch8_2]
```

Ibdex/chapters/Chapter8_2/Cpu0CondMov.td

```
===== Cpu0CondMov.td - Describe Cpu0 Conditional Moves --*- tablegen -*-=====
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//-----
// This is the Conditional Moves implementation.
//
//-----
// Conditional moves:
// These instructions are expanded in
// Cpu0ISelLowering::EmitInstrWithCustomInserter if target does not have
```

(continues on next page)

(continued from previous page)

```
// conditional move instructions.
// cond:int, data:int
class CondMovIntInt<RegisterClass CRC, RegisterClass DRC, bits<8> op,
    string instr_asm> :
FA<op, (outs DRC:$ra), (ins DRC:$rb, CRC:$rc, DRC:$F),
    !strconcat(instr_asm, "\t$ra, $rb, $rc"), [], IIAlu> {
let shamt = 0;
let Constraints = "$F = $ra";
}

// select patterns
multiclass MovzPats0Slt<RegisterClass CRC, RegisterClass DRC,
    Instruction MOVZInst, Instruction SLTop,
    Instruction SLTuOp, Instruction SLTiOp,
    Instruction SLTiuOp> {
def : Pat<(select (i32 (setge CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (SLTop CRC:$lhs, CRC:$rhs), DRC:$F)>;
def : Pat<(select (i32 (setuge CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (SLTuOp CRC:$lhs, CRC:$rhs), DRC:$F)>;
def : Pat<(select (i32 (setge CRC:$lhs, immSExt16:$rhs)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (SLTiOp CRC:$lhs, immSExt16:$rhs), DRC:$F)>;
def : Pat<(select (i32 (setuge CRC:$lh, immSExt16:$rh)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (SLTiuOp CRC:$lh, immSExt16:$rh), DRC:$F)>;
def : Pat<(select (i32 (setle CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (SLTop CRC:$rhs, CRC:$lhs), DRC:$F)>;
def : Pat<(select (i32 (setule CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (SLTuOp CRC:$rhs, CRC:$lhs), DRC:$F)>;
}

multiclass MovzPats1<RegisterClass CRC, RegisterClass DRC,
    Instruction MOVZInst, Instruction XOROp> {
def : Pat<(select (i32 (seteq CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, (XOROp CRC:$lhs, CRC:$rhs), DRC:$F)>;
def : Pat<(select (i32 (seteq CRC:$lhs, 0)), DRC:$T, DRC:$F),
    (MOVZInst DRC:$T, CRC:$lhs, DRC:$F)>;
}

multiclass MovnPats<RegisterClass CRC, RegisterClass DRC, Instruction MOVNInst,
    Instruction XOROp> {
def : Pat<(select (i32 (setne CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
    (MOVNInst DRC:$T, (XOROp CRC:$lhs, CRC:$rhs), DRC:$F)>;
def : Pat<(select CRC:$cond, DRC:$T, DRC:$F),
    (MOVNInst DRC:$T, CRC:$cond, DRC:$F)>;
def : Pat<(select (i32 (setne CRC:$lhs, 0)), DRC:$T, DRC:$F),
    (MOVNInst DRC:$T, CRC:$lhs, DRC:$F)>;
}

// Instantiation of instructions.
def MOVZ_I_I      : CondMovIntInt<CPUREgs, CPUREgs, 0x0a, "movz">;
def MOVN_I_I      : CondMovIntInt<CPUREgs, CPUREgs, 0x0b, "movn">;
```

(continues on next page)

(continued from previous page)

```
// Instantiation of conditional move patterns.
let Predicates = [HasSlt] in {
defm : MovzPats0Slt<CPUREgs, CPUREgs, MOVZ_I_I, SLT, SLTu, SLTi, SLTi>;
}

defm : MovzPats1<CPUREgs, CPUREgs, MOVZ_I_I, XOR>;

defm : MovnPats<CPUREgs, CPUREgs, MOVN_I_I, XOR>;
```

Ibdex/chapters/Chapter8_2/Cpu0ISelLowering.h

```
SDValue lowerSELECT(SDValue Op, SelectionDAG &DAG) const;
```

Ibdex/chapters/Chapter8_2/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                         const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
    setOperationAction(ISD::SELECT, MVT::i32, Custom);
```

```
    setOperationAction(ISD::SELECT_CC, MVT::i32, Expand);
    setOperationAction(ISD::SELECT_CC, MVT::Other, Expand);
```

```
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    switch (Op.getOpcode())
    {
```

```
        case ISD::SELECT: return lowerSELECT(Op, DAG);
```

```
    }
    ...
}
```

```
SDValue Cpu0TargetLowering::
lowerSELECT(SDValue Op, SelectionDAG &DAG) const
{
    return Op;
}
```

Setting *ISD::SELECT_CC* to “Expand” prevents LLVM from optimizing by merging *setcc* and *select* into a single IR instruction *select_cc*².

² <http://llvm.org/docs/WritingAnLLVMBackend.html#expand>

Next, the `LowerOperation()` function directly returns the opcode for `ISD::SELECT`. Finally, the pattern defined in `Cpu0CondMov.td` translates the `select` IR into conditional instructions, `movz` or `movn`.

Let's run Chapter8_2 with `ch8_2_select.cpp` to get the following result.

Again, `cpu032II` uses `slt` instead of `cmp`, resulting in a slight reduction in instruction count.

```
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm ch8_2_select.bc -o -
...
.type _Z11test_movx_1v,@function
...
addiu $2, $zero, 3
movz $2, $3, $4
...
.type _Z11test_movx_2v,@function
...
addiu $2, $zero, 3
movn $2, $3, $4
...
```

Clang uses the `select` IR in small basic blocks to reduce branch costs in pipeline machines, as branches can cause pipeline stalls. However, this optimization requires support for conditional instructions [Page 370, 3](#).

If your backend lacks conditional instruction support but still needs to compile with Clang optimization level `-O1` or higher, you can modify Clang to force it to generate traditional branching blocks instead of the `select` IR.

RISC CPUs were originally designed to take advantage of pipelining, and over time, they have incorporated more instructions. For example, MIPS provides only `movz` and `movn`, while ARM supports a wider range of conditional instructions.

We created the Cpu0 instruction set as a simple RISC pipeline machine for educational use in compiler toolchain tutorials. Although Cpu0 includes a `cmp` instruction, which many programmers are familiar with (as used in ARM), the `slt` instruction is more efficient in a RISC pipeline.

If you are designing a backend for high-level languages like C/C++, it may be better to use `slt` instead of `cmp`, since assembly is rarely used directly, and professional assembly programmers can easily adapt to `slt`.

File `ch8_2_select2.cpp` will generate the `select` IR if compiled with `clang -O1`.

Index/input/ch8_2_select2.cpp

```
// The following files will generate IR select when compile with clang -O1 but
// ~/llvm/debug/build/bin/clang -O0 won't generate IR select.
volatile int a = 1;
volatile int b = 2;

int test_movx_3()
{
    int c = 0;

    if (a < b)
        return 1;
    else
        return 2;
}
```

(continues on next page)

(continued from previous page)

```

int test_movx_4()
{
    int c = 0;

    if (a)
        c = 1;
    else
        c = 3;

    return c;
}

```

List the conditional statements of C, IR, DAG and Cpu0 instructions as the following table.

Table 8.2: Conditional statements of C, IR, DAG and Cpu0 instructions

C	if (a < b) c = 1; else c = 3;
•	c = a ? 1:3;
IR	icmp + (eq, ne, sgt, sge, slt, sle) + br
DAG	((seteq, setne, setgt, setge, setlt, setle) + setcc) + select
Cpu0	movz, movn

The file *ch8_2_select_global_pic.cpp*, mentioned in the chapter “Global Variables”, can now be tested as follows:

Ibdex/input/ch8_2_select_global_pic.cpp

```

volatile int a1 = 1;
volatile int b1 = 2;

int gI1 = 100;
int gJ1 = 50;

int test_select_global_pic()
{
    if (a1 < b1)
        return gI1;
    else
        return gJ1;
}

```

```

Jonathan@ekiiMac:~/input$ clang -O1 -target mips-unknown-linux-gnu
-c ch8_2_select_global_pic.cpp -emit-llvm -o ch8_2_select_global_pic.bc
Jonathan@ekiiMac:~/input$ ~/llvm/test/build/bin/
 llvm-dis ch8_2_select_global_pic.bc -o -
...
@a1 = global i32 1, align 4
@b1 = global i32 2, align 4
@gI1 = global i32 100, align 4
@gJ1 = global i32 50, align 4

```

(continues on next page)

(continued from previous page)

```
; Function Attrs: nounwind
define i32 @_Z18test_select_globalv() #0 {
    %1 = load volatile i32* @a1, align 4, !tbaa !1
    %2 = load volatile i32* @b1, align 4, !tbaa !1
    %3 = icmp slt i32 %1, %2
    %gI1.val = load i32* @gI1, align 4
    %gJ1.val = load i32* @gJ1, align 4
    %.0 = select i1 %3, i32 %gI1.val, i32 %gJ1.val
    ret i32 %.0
}
...
Jonathan@tekiMac:~/input Jonathan$ ~/llvm/test/build/bin/
llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm ch8_2_select_global_
→pic.bc -o -
.section .mdebug.abi32
.previous
.file "ch8_2_select_global_pic.bc"
.text
.globl _Z18test_select_globalv
.align 2
.type _Z18test_select_globalv,@function
.ent _Z18test_select_globalv # @_Z18test_select_globalv
_Z18test_select_globalv:
.frame $sp,0,$lr
.mask 0x00000000,0
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
    lui $2, %got_hi(a1)
    addu $2, $2, $gp
    ld $2, %got_lo(a1)($2)
    ld $2, 0($2)
    lui $3, %got_hi(b1)
    addu $3, $3, $gp
    ld $3, %got_lo(b1)($3)
    ld $3, 0($3)
    cmp $sw, $2, $3
    andi $2, $sw, 1
    lui $3, %got_hi(gJ1)
    addu $3, $3, $gp
    ori $3, $3, %got_lo(gJ1)
    lui $4, %got_hi(gI1)
    addu $4, $4, $gp
    ori $4, $4, %got_lo(gI1)
    movn $3, $4, $2
    ld $2, 0($3)
    ld $2, 0($2)
    ret $lr
.set macro
.set reorder
```

(continues on next page)

(continued from previous page)

```

.end _Z18test_select_globalv
$tmp0:
.size _Z18test_select_globalv, ($tmp0)-_Z18test_select_globalv

.type a1,@object          # @a1
.data
.globl a1
.align 2
a1:
.4byte 1                  # 0x1
.size a1, 4

.type b1,@object          # @b1
.globl b1
.align 2
b1:
.4byte 2                  # 0x2
.size b1, 4

.type gI1,@object          # @gI1
.globl gI1
.align 2
gI1:
.4byte 100                 # 0x64
.size gI1, 4

.type gJ1,@object          # @gJ1
.globl gJ1
.align 2
gJ1:
.4byte 50                  # 0x32
.size gJ1, 4

```

8.7 Phi node

Since the phi node is widely used in SSA form⁵, LLVM applies phi nodes in its IR for optimization purposes as well. Phi nodes are used for **live variable analysis**. An example in C can be found here⁶.

As mentioned on the referenced wiki page, the compiler determines where to insert phi functions by computing **dominance frontiers**.

The following input demonstrates the benefits of using phi nodes:

Ibdex/input/ch8_2_phinode.cpp

```

int test_phinode(int a , int b, int c)
{
    int d = 2;

```

(continues on next page)

⁵ https://en.wikipedia.org/wiki/Static_single_assignment_form

⁶ <http://stackoverflow.com/questions/11485531/what-exactly-phi-instruction-does-and-how-to-use-it-in-llvm>

(continued from previous page)

```

if (a == 0) {
    a = a+1; // a = 1
}
else if (b != 0) {
    a = a-1;
}
else if (c == 0) {
    a = a+2;
}
d = a + b;

return d;
}

```

Compile it with a debug build of Clang using the `-O3` optimization level as follows:

```

114-43-212-251:input Jonathan$ ~/llvm/debug/build/bin/clang
-O3 -target mips-unknown-linux-gnu -c ch8_2_phinode.cpp -emit-llvm -o
ch8_2_phinode.bc
114-43-212-251:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch8_2_phinode.bc -o -
...
define i32 @_Z12test_phinodeiii(i32 signext %a, i32 signext %b, i32 signext %c) local_
↳unnamed_addr #0 {
entry:
    %cmp = icmp eq i32 %a, 0
    br i1 %cmp, label %if.end7, label %if.else

if.else:                                ; preds = %entry
    %cmp1 = icmp eq i32 %b, 0
    br i1 %cmp1, label %if.else3, label %if.then2

if.then2:                                ; preds = %if.else
    %dec = add nsw i32 %a, -1
    br label %if.end7

if.else3:                                ; preds = %if.else
    %cmp4 = icmp eq i32 %c, 0
    %add = add nsw i32 %a, 2
    %add.a = select i1 %cmp4, i32 %add, i32 %a
    br label %if.end7

if.end7:                                ; preds = %entry, %if.else3, %if.
↳then2
    %a.addr.0 = phi i32 [ %dec, %if.then2 ], [ %add.a, %if.else3 ], [ 1, %entry ]
    %add8 = add nsw i32 %a.addr.0, %b
    ret i32 %add8
}

```

Because of SSA form, the LLVM IR must use different names for the destination variable *a* in different basic blocks (e.g., *if-then*, *else*). But how is the source variable *a* in the statement *d = a + b*; named?

In SSA form, the basic block containing *a = a - 1*; uses *%dec* as the destination variable, and the one containing *a = a +*

2; uses `%add`. Since the source value for a in $d = a + b$; comes from different basic blocks, a *phi* node is used to resolve this.

The *phi* structure merges values from different control flow paths to produce a single value for use in subsequent instructions.

When compiled with optimization level `-O0`, Clang does not apply *phi* nodes. Instead, it uses memory operations like *store* and *load* to handle variable values across different basic blocks.

```
114-43-212-251:input Jonathan$ ~/llvm/debug/build/bin/clang
-O0 -target mips-unknown-linux-gnu -c ch8_2_phinode.cpp -emit-llvm -o
ch8_2_phinode.bc
114-43-212-251:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch8_2_phinode.bc -o -
...
define i32 @_Z12test_phinodeiii(i32 signext %a, i32 signext %b, i32 signext %c) #0 {
entry:
%a.addr = alloca i32, align 4
%b.addr = alloca i32, align 4
%c.addr = alloca i32, align 4
%d = alloca i32, align 4
store i32 %a, i32* %a.addr, align 4
store i32 %b, i32* %b.addr, align 4
store i32 %c, i32* %c.addr, align 4
store i32 2, i32* %d, align 4
%0 = load i32, i32* %a.addr, align 4
%cmp = icmp eq i32 %0, 0
br i1 %cmp, label %if.then, label %if.else

if.then:                                ; preds = %entry
%1 = load i32, i32* %a.addr, align 4
%inc = add nsw i32 %1, 1
store i32 %inc, i32* %a.addr, align 4
br label %if.end7

if.else:                                 ; preds = %entry
%2 = load i32, i32* %b.addr, align 4
%cmp1 = icmp ne i32 %2, 0
br i1 %cmp1, label %if.then2, label %if.else3

if.then2:                                ; preds = %if.else
%3 = load i32, i32* %a.addr, align 4
%dec = add nsw i32 %3, -1
store i32 %dec, i32* %a.addr, align 4
br label %if.end6

if.else3:                                ; preds = %if.else
%4 = load i32, i32* %c.addr, align 4
%cmp4 = icmp eq i32 %4, 0
br i1 %cmp4, label %if.then5, label %if.end

if.then5:                                ; preds = %if.else3
%5 = load i32, i32* %a.addr, align 4
%add = add nsw i32 %5, 2
```

(continues on next page)

(continued from previous page)

```

store i32 %add, i32* %a.addr, align 4
br label %if.end

if.end:                                ; preds = %if.then5, %if.else3
  br label %if.end6

if.end6:                                ; preds = %if.end, %if.then2
  br label %if.end7

if.end7:                                ; preds = %if.end6, %if.then
  %6 = load i32, i32* %a.addr, align 4
  %7 = load i32, i32* %b.addr, align 4
  %add8 = add nsw i32 %6, %7
  store i32 %add8, i32* %d, align 4
  %8 = load i32, i32* %d, align 4
  ret i32 %8
}

```

When compiled with `clang -O3`, the compiler generates *phi* functions. The *phi* function assigns a virtual register value directly from multiple basic blocks.

In contrast, compiling with `clang -O0` does not generate *phi* nodes. Instead, it assigns the virtual register value by loading from a stack slot, which is written in each of the multiple basic blocks.

In this example, the pointer `%a.addr` points to the stack slot. The store instructions:

- `store i32 %inc, i32* %a.addr, align 4` in label `if.then`:
- `store i32 %dec, i32* %a.addr, align 4` in label `if.then2`:
- `store i32 %add, i32* %a.addr, align 4` in label `if.then5`:

These instructions show that three *store*'s are needed in the '`-O0`' version.

With optimization, the compiler may determine that the condition `a == 0` is always true. In such cases, the *phi* node version may produce better results, since the `-O0` version uses *load* and *store* with the pointer `%a.addr`, which can hinder further optimization.

Compiler textbooks discuss how the Control Flow Graph (CFG) analysis uses dominance frontier calculations to determine where to insert *phi* nodes. After inserting *phi* nodes, the compiler performs global optimization on the CFG and eventually removes the *phi* nodes by replacing them with *load* and *store* instructions.

If you are interested in more technical details beyond what's provided in the wiki reference, see⁷ for *phi* node coverage, or refer to⁸ for dominator tree analysis if you have the book.

8.8 Phi In Optimization

Optimization in SSA φ-Nodes and LLVM Passes

This section introduces the theoretical foundations of φ-nodes in SSA form, explains how φ-nodes enable compiler optimizations, and surveys LLVM passes that manipulate, simplify, or rely on φ-nodes. The material is suitable for compiler courses, backend engineering documentation, or LLVM-based curriculum modules.

1. Introduction

⁷ Section 8.11 of Muchnick, Steven S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann. ISBN 1-55860-320-4.

⁸ Refer chapter 9 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

Static Single Assignment (SSA) form is one of the most influential intermediate representations in modern compilers. By enforcing that each variable is assigned exactly once, SSA enables precise data-flow reasoning and exposes opportunities for aggressive optimization.

At the center of SSA lies the **φ -node**, a pseudo-instruction that merges values coming from different control-flow paths. Although simple in concept, φ -nodes are essential for:

- constant propagation
- redundancy elimination
- control-flow simplification
- loop optimization
- value numbering and equivalence reasoning

This chapter covers:

- the theory behind φ -nodes and SSA construction
 - optimizations enabled by φ -nodes
 - LLVM passes that manipulate or depend on φ -nodes
 - practical considerations for implementing φ -aware transformations
2. Theoretical Foundations of φ -Nodes

2.1 SSA Form and the Need for φ -Nodes

SSA requires that each variable have a single static assignment. When control flow merges, multiple reaching definitions may exist. A φ -node resolves this ambiguity:

```
%x3 = phi i32 [ %x1, %P1 ], [ %x2, %P2 ]
```

This means:

- if control arrives from predecessor $P_1 \rightarrow$ use x_1
- if control arrives from predecessor $P_2 \rightarrow$ use x_2

The φ -node is a compile-time abstraction, not a runtime instruction.

2.2 Dominance and Placement of φ -Nodes

The classical Cytron et al. algorithm inserts φ -nodes at dominance frontiers. A φ -node for variable v is placed in block B if:

- B is in the dominance frontier of a block that assigns v
- B is reachable from multiple definitions of v

This produces *minimal SSA form*.

2.3 Properties of φ -Nodes Relevant to Optimization

Key properties include:

- **Value identity**: each φ defines a new SSA name.
- **Equivalence exposure**: identical incoming values imply redundancy.
- **Unreachable path detection**: dead predecessors simplify φ -nodes.
- **Loop structure representation**: induction variables naturally use φ -nodes.

3. Optimizations Enabled by φ -Nodes

3.1 Constant Propagation and Folding

If all incoming values are constants, the φ -node collapses:

```
%x = phi i32 [ 3, %A ], [ 3, %B ];    →    x = 3
```

This enables:

- dead code elimination
- branch simplification
- further constant folding

3.2 Copy and Redundancy Elimination

If all incoming SSA names are identical:

```
%x = phi i32 [ %a, %A ], [ %a, %B ];    →    x = a
```

This reduces renaming noise and simplifies data flow.

3.3 Dead φ -Node Elimination

A φ -node is dead if:

- its result is unused, or
- all incoming edges are dead

Dead φ -nodes often cascade, enabling further cleanup.

3.4 Control-Flow Simplification

When branches become constant, φ -nodes collapse:

```
%x = phi i32 [ %a, %A ], [ %b, %B ];    →    x = a
```

This supports:

- jump threading
- block merging
- CFG simplification

3.5 Loop Optimizations

φ -nodes identify:

- induction variables
- loop-carried dependencies
- reduction patterns

This enables:

- Loop-Invariant Code Motion (LICM)
- strength reduction
- induction variable simplification
- loop unrolling and vectorization

4. LLVM Passes Related to φ -Nodes

LLVM contains many passes that either manipulate φ -nodes directly or depend on them for analysis.

4.1 Passes That Directly Modify or Simplify φ -Nodes

mem2reg (PromoteMemoryToRegisterPass)

- Converts stack variables to SSA registers.
- Inserts φ -nodes using dominance frontiers.

SROA (Scalar Replacement of Aggregates)

- Breaks aggregates into scalars.
- Introduces new φ -nodes for scalarized fields.

InstCombine

- Simplifies φ -nodes.
- Removes redundant φ -nodes.
- Folds constant φ -nodes.

SimplifyCFG

- Removes unreachable predecessors.
- Updates φ -nodes accordingly.
- Merges blocks and merges φ -nodes.

CorrelatedValuePropagation

- Uses branch correlation to simplify φ -nodes.

4.2 Passes That Use φ -Nodes for Analysis

GVN (Global Value Numbering)

- Uses φ -translation to compare values across blocks.
- Detects equivalent expressions.

SCCP (Sparse Conditional Constant Propagation)

- Tracks constant values through φ -nodes.
- Eliminates dead branches and simplifies φ -nodes.

LICM (Loop-Invariant Code Motion)

- Identifies loop invariants via φ -nodes.
- Moves invariant code out of loops.

LoopSimplify

- Normalizes loop structure.
- Ensures canonical induction φ -nodes exist.

IndVarSimplify

- Rewrites induction φ -nodes into canonical form.
- Enables strength reduction and vectorization.

4.3 Passes That Remove or Rewrite φ -Nodes

DemotePHIToStack

- Converts φ -nodes back to memory operations.
- Used in lowering or debugging.

SSAUpdater (utility)

- Rewrites SSA form during transformations.
 - Inserts or removes φ -nodes as needed.
5. Practical Considerations for φ -Aware Optimizations

5.1 Maintaining SSA Consistency

Transformations must ensure:

- each φ -node has one incoming value per predecessor
- predecessor order matches CFG edge order
- removing edges updates φ -nodes

5.2 Avoiding Critical Edges

Some optimizations require splitting critical edges to maintain valid φ -node semantics.

5.3 Handling Cyclic φ -Nodes

Loops often produce mutually recursive φ -nodes. Optimizations must detect:

- induction variables
- loop-carried dependencies
- reduction patterns

5.4 φ -Translation

Analyses such as GVN and PRE require translating values across control flow by following φ -node edges. This is essential for correctness.

6. Summary

φ -nodes are a structural backbone of SSA form. They:

- represent merged values across control flow
- enable precise data-flow reasoning
- support loop analysis and transformation
- are manipulated by many LLVM optimization passes

Understanding φ -nodes is essential for designing compiler optimizations, implementing SSA-based transformations, and working effectively with LLVM IR.

8.9 LLVM IR φ -Node Optimization Examples

This section presents LLVM IR examples *before* and *after* key φ -node optimizations. Each transformation illustrates how SSA form enables simplification, redundancy elimination, and loop optimization.

All code blocks use LLVM IR syntax.

1. Constant Propagation and Folding

1.1 Constant Folding of a φ with Identical Constants

Before:

```
define i32 @example_const_phi(i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  %x = phi i32 [ 3, %then ], [ 3, %else ]
  ret i32 %x
}
```

After:

```
define i32 @example_const_phi(i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  ret i32 3
}
```

1.2 Constant Propagation into a φ

Before:

```
define i32 @example_sccp(i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  %x = phi i32 [ 42, %then ], [ 13, %else ]
  %y = add i32 %x, 1
  ret i32 %y
}
```

After SCCP (assuming %cond is always true):

```
define i32 @example_sccp(i1 %cond) {
entry:
  br label %merge

merge:
  ret i32 43
}
```

2. Redundant φ Elimination

2.1 φ with Identical SSA Names

Before:

```
define i32 @example_redundant_phi(i32 %a, i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  %v_then = add i32 %a, 0
  br label %merge

else:
  %v_else = add i32 %a, 0
  br label %merge

merge:
  %v = phi i32 [ %v_then, %then ], [ %v_else, %else ]
  ret i32 %v
}
```

After InstCombine:

```
define i32 @example_redundant_phi(i32 %a, i1 %cond) {
entry:
  br label %merge

merge:
  ret i32 %a
}
```

2.2 Collapsing Copy φ -Chains

Before:

```
define i32 @example_copy_phi(i32 %a, i32 %b, i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  %x = phi i32 [ %a, %entry ]
  br label %merge

else:
  %x2 = phi i32 [ %b, %entry ]
```

(continues on next page)

(continued from previous page)

```

br label %merge

merge:
  %y = phi i32 [ %x, %then ], [ %x2, %else ]
  ret i32 %y
}

```

After Simplification:

```

define i32 @example_copy_phi(i32 %a, i32 %b, i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  %y = phi i32 [ %a, %then ], [ %b, %else ]
  ret i32 %y
}

```

3. Dead φ -Node Elimination

3.1 Unused φ

Before:

```

define i32 @example_dead_phi(i32 %a, i32 %b, i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  br label %merge

else:
  br label %merge

merge:
  %x = phi i32 [ %a, %then ], [ %b, %else ]
  ret i32 0
}

```

After DCE:

```

define i32 @example_dead_phi(i32 %a, i32 %b, i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  br label %merge

```

(continues on next page)

(continued from previous page)

```

else:
    br label %merge

merge:
    ret i32 0
}

```

3.2 Cascading Dead φs

Before:

```

define i32 @example_cascade_dead_phi(i32 %a, i32 %b, i1 %cond) {
entry:
    br i1 %cond, label %then, label %else

then:
    br label %merge

else:
    br label %merge

merge:
    %x = phi i32 [ %a, %then ], [ %b, %else ]
    %y = add i32 %x, 0
    ret i32 1
}

```

After DCE + InstCombine:

```

define i32 @example_cascade_dead_phi(i32 %a, i32 %b, i1 %cond) {
entry:
    br i1 %cond, label %then, label %else

then:
    br label %merge

else:
    br label %merge

merge:
    ret i32 1
}

```

4. Control-Flow Simplification and φ

4.1 Branch Known Constant → φ Collapse

Before:

```

define i32 @example_simplifycfg(i32 %a) {
entry:
    %cmp = icmp eq i32 %a, 0
    br i1 %cmp, label %zero, label %nonzero
}

```

(continues on next page)

(continued from previous page)

```

zero:
  br label %merge

nonzero:
  br label %merge

merge:
  %x = phi i32 [ 0, %zero ], [ 1, %nonzero ]
  ret i32 %x
}

```

After SimplifyCFG:

```

define i32 @example_simplifycfg(i32 %a) {
entry:
  br label %merge

merge:
  ret i32 0
}

```

4.2 Block Merging

Before:

```

define i32 @example_block_merge(i32 %a, i1 %cond) {
entry:
  br i1 %cond, label %then, label %else

then:
  %x1 = add i32 %a, 1
  br label %merge

else:
  %x2 = add i32 %a, 1
  br label %merge

merge:
  %x = phi i32 [ %x1, %then ], [ %x2, %else ]
  ret i32 %x
}

```

After SimplifyCFG + InstCombine:

```

define i32 @example_block_merge(i32 %a, i1 %cond) {
entry:
  %x = add i32 %a, 1
  ret i32 %x
}

```

5. Loop Optimizations Using φ -Nodes

5.1 Canonical Induction Variable

Before:

```

define i32 @sum(i32* %arr, i32 %n) {
entry:
  br label %loop

loop:
  %i = phi i32 [ 0, %entry ], [ %i.next, %loop ]
  %sum = phi i32 [ 0, %entry ], [ %sum.next, %loop ]

  %inbounds = icmp slt i32 %i, %n
  br i1 %inbounds, label %body, label %exit

body:
  %ptr = getelementptr inbounds i32, i32* %arr, i32 %i
  %val = load i32, i32* %ptr
  %sum.next = add i32 %sum, %val
  %i.next = add i32 %i, 1
  br label %loop

exit:
  ret i32 %sum
}

```

After IndVarSimplify (conceptual):

```

define i32 @sum(i32* %arr, i32 %n) {
entry:
  br label %loop

loop:
  %iv = phi i64 [ 0, %entry ], [ %iv.next, %loop ]
  %sum = phi i32 [ 0, %entry ], [ %sum.next, %loop ]

  %iv.trunc = trunc i64 %iv to i32
  %inbounds = icmp slt i32 %iv.trunc, %n
  br i1 %inbounds, label %body, label %exit

body:
  %ptr = getelementptr inbounds i32, i32* %arr, i64 %iv
  %val = load i32, i32* %ptr
  %sum.next = add i32 %sum, %val
  %iv.next = add i64 %iv, 1
  br label %loop

exit:
  ret i32 %sum
}

```

5.2 Loop-Invariant Code Motion (LICM)

Before:

```

define void @example_licm(i32* %p, i32* %q, i32 %n) {
entry:
  br label %loop

```

(continues on next page)

(continued from previous page)

```

loop:
    %i = phi i32 [ 0, %entry ], [ %i.next, %loop ]
    %inb = icmp slt i32 %i, %n
    br i1 %inb, label %body, label %exit

body:
    %c = load i32, i32* %q
    %ptr = getelementptr inbounds i32, i32* %p, i32 %i
    store i32 %c, i32* %ptr
    %i.next = add i32 %i, 1
    br label %loop

exit:
    ret void
}

```

After LICM:

```

define void @example_licm(i32* %p, i32* %q, i32 %n) {
entry:
    %c = load i32, i32* %q
    br label %loop

loop:
    %i = phi i32 [ 0, %entry ], [ %i.next, %loop ]
    %inb = icmp slt i32 %i, %n
    br i1 %inb, label %body, label %exit

body:
    %ptr = getelementptr inbounds i32, i32* %p, i32 %i
    store i32 %c, i32* %ptr
    %i.next = add i32 %i, 1
    br label %loop

exit:
    ret void
}

```

6. mem2reg: Introducing φ-Nodes

6.1 Before mem2reg

```

define i32 @example_mem2reg(i1 %cond) {
entry:
    %x = alloca i32, align 4
    store i32 0, i32* %x
    br i1 %cond, label %then, label %else

then:
    store i32 1, i32* %x
    br label %merge

```

(continues on next page)

(continued from previous page)

```

else:
    store i32 2, i32* %x
    br label %merge

merge:
    %x.load = load i32, i32* %x
    ret i32 %x.load
}

```

6.2 After mem2reg

```

define i32 @example_mem2reg(i1 %cond) {
entry:
    br i1 %cond, label %then, label %else

then:
    br label %merge

else:
    br label %merge

merge:
    %x.promoted = phi i32 [ 1, %then ], [ 2, %else ]
    ret i32 %x.promoted
}

```

8.10 RISC CPU knowledge

As mentioned in the previous section, Cpu0 is a RISC (Reduced Instruction Set Computer) CPU with a 5-stage pipeline. RISC CPUs dominate the world today. Even the x86 architecture, traditionally classified as CISC (Complex Instruction Set Computer), internally translates its complex instructions into micro-instructions that are pipelined like RISC.

Understanding RISC concepts can be very helpful and rewarding for compiler design.

Below are two excellent and popular books we have read and recommend for reference. Although there are many books on computer architecture, these two stand out for their clarity and depth:

- *Computer Organization and Design: The Hardware/Software Interface* (The Morgan Kaufmann Series in Computer Architecture and Design)
- *Computer Architecture: A Quantitative Approach* (The Morgan Kaufmann Series in Computer Architecture and Design)

The book *Computer Organization and Design* (available in 4 editions at the time of writing) serves as an introductory text, while *Computer Architecture: A Quantitative Approach* (with 5 editions) explores more advanced and in-depth topics in CPU architecture.

Both books use the MIPS CPU as a reference example, since MIPS is more RISC-like than many other commercial CPUs.

FUNCTION CALL

- *MIPS Stack Frame*
- *Load Incoming Arguments from Stack Frame*
- *Store Outgoing Arguments to Stack Frame*
 - *Pseudo Hook Instructions ADJCALLSTACKDOWN and ADJCALLSTACKUP*
 - *Read LowerCall() with Graphviz's Help*
 - *Long and Short String Initialization*
- *Structure Type Support*
 - *Ordinary Struct Type*
 - *Byval Struct Type*
- *Function Call Optimization*
 - *Tail Call Optimization*
 - *Recursion optimization*
- *Other Features Supported*
 - *Global Variables Accessing In PIC Addressing Mode*
 - *Variable number of arguments*
 - *Dynamic stack allocation support*
 - *Variable sized array support*
 - *Function related Intrinsics support*
 - * *frameaddress and returnaddress intrinsics*
 - * *eh.return intrinsic*
 - * *eh.dwarf intrinsic*
 - * *bswap intrinsic*
 - *Add specific backend intrinsic function*
- *Summary*

This chapter introduces support for subroutine and function calls in backend translation. A significant amount of code is required to support function calls, and it is organized using LLVM-supplied interfaces for clarity.

The chapter begins by introducing the MIPS stack frame structure, as many parts of the ABI are borrowed from it. Although each CPU has its own ABI, most RISC CPU ABIs share similar characteristics.

Section “4.5 DAG Lowering” of *tricore_llvm.pdf* provides insight into the lowering process. Section “4.5.1 Calling Conventions” in the same document is also a helpful reference for further understanding.

If you have difficulty understanding the stack frame illustrated in the first three sections of this chapter, you may consult the following resources: Appendix B, “Procedure Call Convention,” in *Computer Organization and Design, 1st Edition*¹; “Run Time Memory” in a compiler textbook; or “Function Call Sequence” and “Stack Frame” in the MIPS ABI³.

9.1 MIPS Stack Frame

The first step in designing Cpu0 function calls is deciding how to pass arguments. There are two options:

1. Pass all arguments on the stack.
2. Pass arguments using registers reserved for function arguments, and place any remaining arguments on the stack once the registers are full.

For example, MIPS passes the first four arguments in registers *\$a0*, *\$a1*, *\$a2*, and *\$a3*. Any additional arguments are passed on the stack. Fig. 9.1 illustrates the MIPS stack frame.

Base	Offset	Contents	Frame
		unspecified ...	<i>High addresses</i>
	+16	variable size (if present) incoming arguments passed in stack frame	Previous
old \$sp	+0	space for incoming arguments 1-4	
		locals and temporaries	
		general register save area	
		floating-point register save area	
\$sp	+0	argument build area	<i>Low addresses</i>

Fig. 9.1: Mips stack frame

Run `llc -march=mips` on `ch9_1.bc`, and you will get the following result. See the comments marked with “//”.

Ibdex/input/ch9_1.cpp

```
int gI = 100;

int sum_i(int x1, int x2, int x3, int x4, int x5, int x6)
{
    int sum = gI + x1 + x2 + x3 + x4 + x5 + x6;
```

(continues on next page)

¹ Computer Organization and Design: The Hardware/Software Interface 1st edition (The Morgan Kaufmann Series in Computer Architecture and Design)

³ <http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf>

(continued from previous page)

```

    return sum;
}

int main()
{
    int a = sum_i(1, 2, 3, 4, 5, 6);

    return a;
}

```

```

118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_1.cpp -emit-llvm -o ch9_1.bc
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=mips -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.mips.s
118-165-78-230:input Jonathan$ cat ch9_1.mips.s
.section .mdebug.abi32
.previous
.file "ch9_1.bc"
.text
.globl _Z5sum_iiiiiii
.align 2
.type _Z5sum_iiiiiii,@function
.set nomips16           # @_Z5sum_iiiiiii
.ent _Z5sum_iiiiiii
_Z5sum_iiiiiii:
.cfi_startproc
.frame $sp,32,$ra
.mask 0x00000000,0
.fmask 0x00000000,0
.set noreorder
.set nomacro
.set noat
# BB#0:
addiu $sp, $sp, -32
$tmp1:
.cfi_def_cfa_offset 32
sw $4, 28($sp)
sw $5, 24($sp)
sw $t9, 20($sp)
sw $7, 16($sp)
lw $1, 48($sp) // load argument 5
sw $1, 12($sp)
lw $1, 52($sp) // load argument 6
sw $1, 8($sp)
lw $2, 24($sp)
lw $3, 28($sp)
addu $2, $3, $2
lw $3, 20($sp)
addu $2, $2, $3
lw $3, 16($sp)
addu $2, $2, $3

```

(continues on next page)

(continued from previous page)

```
lw $3, 12($sp)
addu $2, $2, $3
addu $2, $2, $1
sw $2, 4($sp)
jr $ra
addiu $sp, $sp, 32
.set at
.set macro
.set reorder
.end _Z5sum_iiiiii
$tmp2:
.size _Z5sum_iiiiii, ($tmp2)-_Z5sum_iiiiii
.cfi_endproc

.globl main
.align 2
.type main,@function
.set nomips16           # @main
.ent main
main:
.cfi_startproc
.frame $sp,40,$ra
.mask 0x80000000,-4
.fmask 0x00000000,0
.set noreorder
.set nomacro
.set noat
# BB#0:
lui $2, %hi(_gp_disp)
ori $2, $2, %lo(_gp_disp)
addiu $sp, $sp, -40
$tmp5:
.cfi_def_cfa_offset 40
sw $ra, 36($sp)          # 4-byte Folded Spill
$tmp6:
.cfi_offset 31, -4
addu $gp, $2, $25
sw $zero, 32($sp)
addiu $1, $zero, 6
sw $1, 20($sp) // Save argument 6 to 20($sp)
addiu $1, $zero, 5
sw $1, 16($sp) // Save argument 5 to 16($sp)
lw $25, %call16(_Z5sum_iiiiii)($gp)
addiu $4, $zero, 1      // Pass argument 1 to $4 (=a0)
addiu $5, $zero, 2      // Pass argument 2 to $5 (=a1)
addiu $t9, $zero, 3
jalr $25
addiu $7, $zero, 4
sw $2, 28($sp)
lw $ra, 36($sp)          # 4-byte Folded Reload
jr $ra
addiu $sp, $sp, 40
```

(continues on next page)

(continued from previous page)

```
.set at
.set macro
.set reorder
.end main
$tmp7:
.size main, ($tmp7)-main
.cfi_endproc
```

From the MIPS assembly code generated above, we can see that the first four arguments are saved in registers $$a0$ to $$a3$, and the last two arguments are saved at memory locations $16($sp)$ and $20($sp)$.

Fig. 9.2 shows the location of the arguments in the example code *ch9_1.cpp*.

In the *sum_i()* function, argument 5 is loaded from $48($sp)$ because it was stored at $16($sp)$ in the *main()* function. Since the stack size of *sum_i()* is 32, the address of the incoming argument 5 is calculated as $16 + 32 = 48($sp)$.

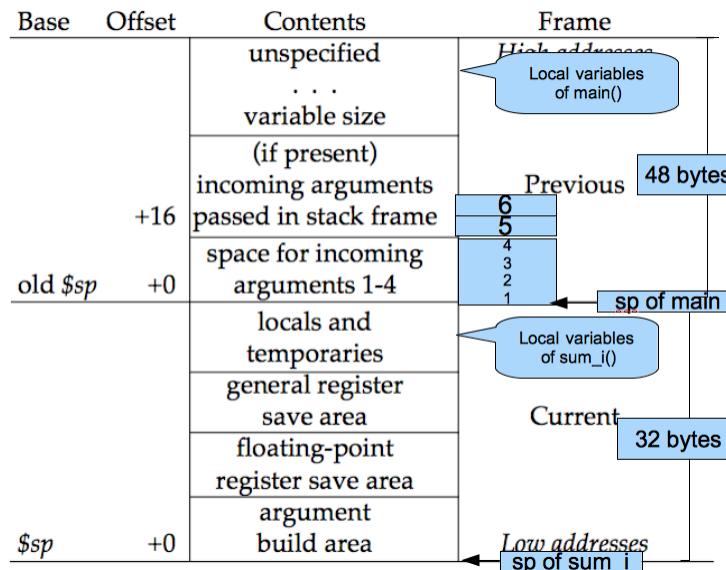


Fig. 9.2: Mips arguments location in stack frame

The document *007-2418-003.pdf* referenced in² is the MIPS assembly language manual. The MIPS Application Binary Interface, referenced in^{Page 398, 3}, includes the diagram shown in Fig. 9.1.

9.2 Load Incoming Arguments from Stack Frame

As discussed in the previous section, supporting function calls requires implementing an argument-passing mechanism using the stack frame.

Before proceeding with the implementation, let's run the old version of the code in *Chapter8_2/* with *ch9_1.cpp* and observe what happens.

```
118-165-79-31:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch9_1.bc -o ch9_1.cpu0.s
```

(continues on next page)

² <http://math-atlas.sourceforge.net-devel/assembly/007-2418-003.pdf>

(continued from previous page)

```
Assertion failed: (InVals.size() == Ins.size() && "LowerFormalArguments didn't
emit the correct number of values!"), function LowerArguments, file /Users/
Jonathan/llvm/test/llvm/lib/CodeGen/SelectionDAG/
SelectionDAGBuilder.cpp, ...
...
0. Program arguments: /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
1. Running pass 'Function Pass Manager' on module 'ch9_1.bc'.
2. Running pass 'CPU0 DAG->DAG Pattern Instruction Selection' on function
'@_Z5sum_iiiiii'.
Illegal instruction: 4
```

Since *Chapter8_2*/defines *LowerFormalArguments()* with an empty body, we receive the error messages shown above.

Before implementing *LowerFormalArguments()*, we must first decide how to pass arguments in a function call.

For demonstration purposes, Cpu0 passes the first two arguments in registers by default, which corresponds to the setting `llc -cpu0-s32-calls=false`.

When using `llc -cpu0-s32-calls=true`, Cpu0 passes all its arguments on the stack.

The function *LowerFormalArguments()* is responsible for creating the incoming arguments. We define it as follows:

Ibdex/chapters/Chapter9_1/Cpu0ISelLowering.h

```
class Cpu0TargetLowering : public TargetLowering {
    /// Cpu0CC - This class provides methods used to analyze formal and call
    /// arguments and inquire about calling convention information.
    class Cpu0CC {
        void analyzeFormalArguments(const SmallVectorImpl<ISD::InputArg> &Ins,
                                    bool IsSoftFloat,
                                    Function::const_arg_iterator FuncArg);

        /// regSize - Size (in number of bits) of integer registers.
        unsigned regSize() const { return IsO32 ? 4 : 4; }
        /// numIntArgRegs - Number of integer registers available for calls.
        unsigned numIntArgRegs() const;

        /// Return pointer to array of integer argument registers.
        const ArrayRef<MCPhysReg> intArgRegs() const;

        void handleByValArg(unsigned ValNo, MVT ValVT, MVT LocVT,
                            CCValAssign::LocInfo LocInfo,
                            ISD::ArgFlagsTy ArgFlags);

        /// useRegsForByval - Returns true if the calling convention allows the
        /// use of registers to pass byval arguments.
        bool useRegsForByval() const { return CallConv != CallingConv::Fast; }
    };
};
```

(continues on next page)

(continued from previous page)

```
/// Return the function that analyzes fixed argument list functions.
llvm::CCAssignFn *fixedArgFn() const;
```

```
void allocateRegs(ByValArgInfo &ByVal, unsigned ByValSize,
                  unsigned Align);
```

```
};
```

```
...
```

```
/// isEligibleForTailCallOptimization - Check whether the call is eligible
/// for tail call optimization.
virtual bool
isEligibleForTailCallOptimization(const Cpu0CC &Cpu0CCInfo,
                                   unsigned NextStackOffset,
                                   const Cpu0FunctionInfo& FI) const = 0;
```

```
/// copyByValArg - Copy argument registers which were used to pass a byval
/// argument to the stack. Create a stack frame object for the byval
/// argument.
void copyByValRegs(SDValue Chain, const SDLoc &DL,
                    std::vector<SDValue> &OutChains, SelectionDAG &DAG,
                    const ISD::ArgFlagsTy &Flags,
                    SmallVectorImpl<SDValue> &InVals,
                    const Argument *FuncArg,
                    const Cpu0CC &CC, const ByValArgInfo &ByVal) const;
```

```
SDValue LowerCall(TargetLowering::CallLoweringInfo &CLI,
                  SmallVectorImpl<SDValue> &InVals) const override;
```

```
...
```

Ibdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```
// addLiveIn - This helper function adds the specified physical register to the
// MachineFunction as a live in value. It also creates a corresponding
// virtual register for it.
static unsigned
addLiveIn(MachineFunction &MF, unsigned PReg, const TargetRegisterClass *RC)
{
    unsigned VReg = MF.getRegInfo().createVirtualRegister(RC);
    MF.getRegInfo().addLiveIn(PReg, VReg);
    return VReg;
}
```

```
=====/
// TODO: Implement a generic logic using tblgen that can support this.
// Cpu0 32 ABI rules:
// ---
```

(continues on next page)

(continued from previous page)

```
//===== /  
  
// Passed in stack only.  
static bool CC_Cpu0S32(unsigned ValNo, MVT ValVT, MVT LocVT,  
                      CCValAssign::LocInfo LocInfo, ISD::ArgFlagsTy ArgFlags,  
                      CCState &State) {  
    // Do not process byval args here.  
    if (ArgFlags.isByVal())  
        return true;  
  
    // Promote i8 and i16  
    if (LocVT == MVT::i8 || LocVT == MVT::i16) {  
        LocVT = MVT::i32;  
        if (ArgFlags.isSExt())  
            LocInfo = CCValAssign::SExt;  
        else if (ArgFlags.isZExt())  
            LocInfo = CCValAssign::ZExt;  
        else  
            LocInfo = CCValAssign::AExt;  
    }  
  
    Align OrigAlign = ArgFlags.getNonZeroOrigAlign();  
    unsigned Offset = State.AllocateStack(ValVT.getSizeInBits() >> 3,  
                                         OrigAlign);  
    State.addLoc(CCValAssign::getMem(ValNo, ValVT, Offset, LocVT, LocInfo));  
    return false;  
}  
  
// Passed first two i32 arguments in registers and others in stack.  
static bool CC_Cpu0O32(unsigned ValNo, MVT ValVT, MVT LocVT,  
                      CCValAssign::LocInfo LocInfo, ISD::ArgFlagsTy ArgFlags,  
                      CCState &State) {  
    static const MCPhysReg IntRegs[] = { Cpu0::A0, Cpu0::A1 };  
  
    // Do not process byval args here.  
    if (ArgFlags.isByVal())  
        return true;  
  
    // Promote i8 and i16  
    if (LocVT == MVT::i8 || LocVT == MVT::i16) {  
        LocVT = MVT::i32;  
        if (ArgFlags.isSExt())  
            LocInfo = CCValAssign::SExt;  
        else if (ArgFlags.isZExt())  
            LocInfo = CCValAssign::ZExt;  
        else  
            LocInfo = CCValAssign::AExt;  
    }  
  
    unsigned Reg;  
  
    // f32 and f64 are allocated in A0, A1 when either of the following
```

(continues on next page)

(continued from previous page)

```

// is true: function is vararg, argument is 3rd or higher, there is previous
// argument which is not f32 or f64.
bool AllocateFloatsInIntReg = true;
Align OrigAlign = ArgFlags.getNonZeroOrigAlign();
bool isI64 = (ValVT == MVT::i32 && OrigAlign == 8);

if (ValVT == MVT::i32 || (ValVT == MVT::f32 && AllocateFloatsInIntReg)) {
    Reg = State.AllocateReg(IntRegs);
    // If this is the first part of an i64 arg,
    // the allocated register must be A0.
    if (isI64 && (Reg == Cpu0::A1))
        Reg = State.AllocateReg(IntRegs);
    LocVT = MVT::i32;
} else if (ValVT == MVT::f64 && AllocateFloatsInIntReg) {
    // Allocate int register. If first
    // available register is Cpu0::A1, shadow it too.
    Reg = State.AllocateReg(IntRegs);
    if (Reg == Cpu0::A1)
        Reg = State.AllocateReg(IntRegs);
    State.AllocateReg(IntRegs);
    LocVT = MVT::i32;
} else
    llvm_unreachable("Cannot handle this ValVT.");

if (!Reg) {
    unsigned Offset = State.AllocateStack(ValVT.getSizeInBits() >> 3,
                                         Align(OrigAlign));
    State.addLoc(CCValAssign::getMem(ValNo, ValVT, Offset, LocVT, LocInfo));
} else
    State.addLoc(CCValAssign::getReg(ValNo, ValVT, Reg, LocVT, LocInfo));

return false;
}

```

```

//=====
//          Call Calling Convention Implementation
//=====

static const MCPhysReg O32IntRegs[] = {
    Cpu0::A0, Cpu0::A1
};

```

```

//@LowerCall {
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                             SmallVectorImpl<SDValue> &InVals) const {

```

```

//@LowerCall {
/// LowerCall - functions arguments are copied from virtual regs to

```

(continues on next page)

(continued from previous page)

```
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {

    return CLI.Chain;

}

//=====

//@LowerFormalArguments {
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {

    MachineFunction &MF = DAG.getMachineFunction();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    Cpu0FI->setVarArgsFrameIndex(0);

    // Assign locations to all of the incoming arguments.
    SmallVector<CCValAssign, 16> ArgLocs;
    CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                   ArgLocs, *DAG.getContext());
    Cpu0CC Cpu0CCInfo(CallConv, ABI::IsO32(),
                      CCInfo);

    const Function &Func = DAG.getMachineFunction().getFunction();
    Function::const_arg_iterator FuncArg = Func.arg_begin();

    bool UseSoftFloat = Subtarget.abiUsesSoftFloat();

    Cpu0CCInfo.analyzeFormalArguments(Ins, UseSoftFloat, FuncArg);
    Cpu0FI->setFormalArgInfo(CCInfo.getNextStackOffset(),
                               Cpu0CCInfo.hasByValArg());

    // Used with vargs to accumulate store chains.
    std::vector<SDValue> OutChains;

    unsigned CurArgIdx = 0;
    Cpu0CC::byval_iterator ByValArg = Cpu0CCInfo.byval_begin();

    //@2 {
```

(continues on next page)

(continued from previous page)

```

for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
//@2 }
    CCValAssign &VA = ArgLocs[i];
    if (Ins[i].isOrigArg()) {
        std::advance(FuncArg, Ins[i].getOrigArgIndex() - CurArgIdx);
        CurArgIdx = Ins[i].getOrigArgIndex();
    }
    EVT ValVT = VA.getValVT();
    ISD::ArgFlagsTy Flags = Ins[i].Flags;
    bool IsRegLoc = VA.isRegLoc();

    //@byval pass {
    if (Flags.isByVal()) {
        assert(Flags.getByValSize() &&
               "ByVal args of size 0 should have been ignored by front-end.");
        assert(ByValArg != Cpu0CCInfo.byval_end());
        copyByValRegs(Chain, DL, OutChains, DAG, Flags, InVals, &FuncArg,
                      Cpu0CCInfo, *ByValArg);
        ++ByValArg;
        continue;
    }
    //@byval pass }
    // Arguments stored on registers
    if (ABI.IsO32() && IsRegLoc) {
        MVT RegVT = VA.getLocVT();
        unsigned ArgReg = VA.getLocReg();
        const TargetRegisterClass *RC = getRegClassFor(RegVT);

        // Transform the arguments stored on
        // physical registers into virtual ones
        unsigned Reg = addLiveIn(DAG.getMachineFunction(), ArgReg, RC);
        SDValue ArgValue = DAG.getCopyFromReg(Chain, DL, Reg, RegVT);

        // If this is an 8 or 16-bit value, it has been passed promoted
        // to 32 bits. Insert an assert[sz]ext to capture this, then
        // truncate to the right size.
        if (VA.getLocInfo() != CCValAssign::Full) {
            unsigned Opcode = 0;
            if (VA.getLocInfo() == CCValAssign::SExt)
                Opcode = ISD::AssertSext;
            else if (VA.getLocInfo() == CCValAssign::ZExt)
                Opcode = ISD::AssertZext;
            if (Opcode)
                ArgValue = DAG.getNode(Opcode, DL, RegVT, ArgValue,
                                      DAG.getValueType(ValVT));
            ArgValue = DAG.getNode(ISD::TRUNCATE, DL, ValVT, ArgValue);
        }

        // Handle floating point arguments passed in integer registers.
        if ((RegVT == MVT::i32 && ValVT == MVT::f32) ||
            (RegVT == MVT::i64 && ValVT == MVT::f64))
            ArgValue = DAG.getNode(ISD::BITCAST, DL, ValVT, ArgValue);
    }
}

```

(continues on next page)

(continued from previous page)

```
InVals.push_back(ArgValue);
} else { // VA.isRegLoc()
    MVT LocVT = VA.getLocVT();

    // sanity check
    assert(VA.isMemLoc());

    // The stack pointer offset is relative to the caller stack frame.
    int FI = MFI.CreateFixedObject(ValVT.getSizeInBits()/8,
                                   VA.getLocMemOffset(), true);

    // Create load nodes to retrieve arguments from the stack
    SDValue FIN = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
    SDValue Load = DAG.getLoad(
        LocVT, DL, Chain, FIN,
        MachinePointerInfo::getFixedStack(DAG.getMachineFunction(), FI));
    InVals.push_back(Load);
    OutChains.push_back(Load.getValue(1));
}
}

//@Ordinary struct type: 1 {
for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    // The cpu0 ABIs for returning structs by value requires that we copy
    // the sret argument into $v0 for the return. Save the argument into
    // a virtual register so that we can access it from the return points.
    if (Ins[i].Flags.isSRet()) {
        unsigned Reg = Cpu0FI->getSRetReturnReg();
        if (!Reg) {
            Reg = MF.getRegInfo().createVirtualRegister(
                getRegClassFor(MVT::i32));
            Cpu0FI->setSRetReturnReg(Reg);
        }
        SDValue Copy = DAG.getCopyToReg(DAG.getEntryNode(), DL, Reg, InVals[i]);
        Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, Copy, Chain);
        break;
    }
}
//@Ordinary struct type: 1 }

// All stores are grouped in one node to allow the matching between
// the size of Ins and InVals. This only happens when on varg functions
if (!OutChains.empty()) {
    OutChains.push_back(Chain);
    Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, OutChains);
}

return Chain;
}
// @LowerFormalArguments

//=====//
```

```

void Cpu0TargetLowering::Cpu0CC::
analyzeFormalArguments(const SmallVectorImpl<ISD::InputArg> &Args,
                      bool IsSoftFloat, Function::const_arg_iterator FuncArg) {
    unsigned NumArgs = Args.size();
    llvm::CCAssignFn *FixedFn = fixedArgFn();
    unsigned CurArgIdx = 0;

    for (unsigned I = 0; I != NumArgs; ++I) {
        MVT ArgVT = Args[I].VT;
        ISD::ArgFlagsTy ArgFlags = Args[I].Flags;
        if (Args[I].isOrigArg()) {
            std::advance(FuncArg, Args[I].getOrigArgIndex() - CurArgIdx);
            CurArgIdx = Args[I].getOrigArgIndex();
        }
        CurArgIdx = Args[I].OrigArgIndex;

        if (ArgFlags.isByVal()) {
            handleByValArg(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags);
            continue;
        }

        MVT RegVT = getRegVT(ArgVT, IsSoftFloat);

        if (!FixedFn(I, ArgVT, RegVT, CCValAssign::Full, ArgFlags, CCInfo))
            continue;
    }

#ifndef NDEBUG
    dbgs() << "Formal Arg #" << I << " has unhandled type "
        << EVT(ArgVT).getEVTString();
#endif
    llvm_unreachable(nullptr);
}
}

```

```

void Cpu0TargetLowering::Cpu0CC::handleByValArg(unsigned ValNo, MVT ValVT,
                                              MVT LocVT,
                                              CCValAssign::LocInfo LocInfo,
                                              ISD::ArgFlagsTy ArgFlags) {
    assert(ArgFlags.getByValSize() && "Byval argument's size shouldn't be 0.");

    struct ByValArgInfo ByVal;
    unsigned RegSize = regSize();
    unsigned ByValSize = alignTo(ArgFlags.getByValSize(), RegSize);
    Align Alignment = std::min(std::max(ArgFlags.getNonZeroByValAlign(),
                                         Align(RegSize)),
                               Align(RegSize * 2));

    if (useRegsForByval())
        allocateRegs(ByVal, ByValSize, Alignment.value());

    // Allocate space on caller's stack.
    ByVal.Address = CCInfo.AllocateStack(ByValSize - RegSize * ByVal.NumRegs,
                                         Alignment);
}

```

(continues on next page)

(continued from previous page)

```

        Alignment);
CCInfo.addLoc(CCValAssign::getMem(ValNo, ValVT, ByVal.Address, LocVT,
                                  LocInfo));
ByValArgs.push_back(ByVal);
}

unsigned Cpu0TargetLowering::Cpu0CC::numIntArgRegs() const {
    return IsO32 ? array_lengthof(O32IntRegs) : 0;
}

```

```

const ArrayRef<MCPhysReg> Cpu0TargetLowering::Cpu0CC::intArgRegs() const {
    return makeArrayRef(O32IntRegs);
}

llvm::CCAssignFn *Cpu0TargetLowering::Cpu0CC::fixedArgFn() const {
    if (IsO32)
        return CC_Cpu0O32;
    else // IsS32
        return CC_Cpu0S32;
}

```

```

void Cpu0TargetLowering::Cpu0CC::allocateRegs(ByValArgInfo &ByVal,
                                              unsigned ByValSize,
                                              unsigned Align) {
    unsigned RegSize = regSize(), NumIntArgRegs = numIntArgRegs();
    const ArrayRef<MCPhysReg> IntArgRegs = intArgRegs();
    assert(!(ByValSize % RegSize) && !(Align % RegSize) &&
           "Byval argument's size and alignment should be a multiple of"
           "RegSize.");
    ByVal.FirstIdx = CCInfo.getFirstUnallocated(IntArgRegs);

    // If Align > RegSize, the first arg register must be even.
    if ((Align > RegSize) && (ByVal.FirstIdx % 2)) {
        CCInfo.AllocateReg(IntArgRegs[ByVal.FirstIdx]);
        ++ByVal.FirstIdx;
    }

    // Mark the registers allocated.
    for (unsigned I = ByVal.FirstIdx; ByValSize && (I < NumIntArgRegs);
         ByValSize -= RegSize, ++I, ++ByVal.NumRegs)
        CCInfo.AllocateReg(IntArgRegs[I]);
}

```

As reviewed in the section “Global variable”⁴, we handled global variable translation by first creating the IR DAG in *LowerGlobalAddress()*, and then completing instruction selection based on the corresponding machine instruction DAGs in *Cpu0InstrInfo.td*.

LowerGlobalAddress() is called when *llc* encounters a global variable access. Similarly, *LowerFormalArguments()* is called when entering a function.

Before entering the “**for loop**”, it gathers incoming argument information using *CCInfo(CallConv, ..., ArgLocs, ...)*.

⁴ <http://jonathan2251.github.io/lbd/globalvar.html#global-variable>

In *ch9_1.cpp*, the function *sum_i(...)* has 6 arguments. Thus, *ArgLocs.size()* is 6, with each argument's information stored in *ArgLocs[i]*.

- If *VA.isRegLoc()* returns true, the argument is passed via register.
- If *VA.isMemLoc()* returns true, the argument is passed via memory stack.

For register-passed arguments, the register is marked as “live-in”, and the value is copied directly from the register.

For stack-passed arguments, a stack offset is created for the frame index object. A load node is then created using this offset and added to the *InVals* vector.

When using `llc -cpu0-s32-calls=false`, the first two arguments are passed in registers, and the remaining arguments are passed in the stack frame.

When using `llc -cpu0-s32-calls=true`, all arguments are passed in the stack frame.

Before handling arguments, *analyzeFormalArguments()* is called. Inside it, *fixedArgFn()* is used to return the function pointer to either *CC_Cpu0O32()* or *CC_Cpu0S32()*.

ArgFlags.isByVal() will be true for “struct pointer byval” arguments, such as `%struct.S* byval` in *tailcall.ll*.

With `llc -cpu0-s32-calls=false`, the stack offset begins at 8 (to allow space in case argument registers are spilled). With `llc -cpu0-s32-calls=true`, the stack offset begins at 0.

For example, when running *ch9_1.cpp* with `llc -cpu0-s32-calls=true` (memory stack only), *LowerFormalArguments()* will be called twice:

- First, for *sum_i()*, it will create six load DAGs for the six incoming arguments.
- Second, for *main()*, no load DAG is created, as there are no incoming arguments.

In addition to *LowerFormalArguments()*, we use *loadRegFromStackSlot()* (defined in an earlier chapter) to generate the machine instruction “**ld \$r, offset(\$sp)**”, which loads arguments from the stack frame.

GetMemOperand(..., FI, ...) returns the memory location of the frame index variable, representing the offset.

For the input *ch9_incoming.cpp* shown below, *LowerFormalArguments()* will generate the red-boxed DAG nodes illustrated in Fig. 9.3 and Fig. 9.4, corresponding to `llc -cpu0-s32-calls=true` and `llc -cpu0-s32-calls=false`, respectively.

The root node at the bottom is created by:

Ibdex/input/ch9_incoming.cpp

```
int sum_i(int x1, int x2, int x3)
{
    int sum = x1 + x2 + x3;

    return sum;
}
```

```
JonathantekiiMac:input Jonathan$ clang -O3 -target mips-unknown-linux-gnu -c
ch9_incoming.cpp -emit-llvm -o ch9_incoming.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch9_incoming.bc -o -
...
define i32 @_Z5sum_iisi(i32 %x1, i32 %x2, i32 %x3) #0 {
    %1 = add nsw i32 %x2, %x1
    %2 = add nsw i32 %1, %x3
```

(continues on next page)

(continued from previous page)

```
ret i32 %2
}
```

In addition to the calling convention and `LowerFormalArguments()`, `Chapter9_1` adds support for instruction selection and printing of the Cpu0 instructions `swi` (software interrupt), `jsub`, and `jalr` (function call).

Ibdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```
def SDT_Cpu0JmpLink : SDTypeProfile<0, 1, [SDTCisVT<0, iPTR]>;
```

```
// Call
def Cpu0JmpLink : SDNode<"Cpu0ISD::JmpLink", SDT_Cpu0JmpLink,
    [SDNPHasChain, SDNPOutGlue, SDNPOptInGlue,
    SDNPVariadic]>;
```

```
class IsTailCall {
    bit isCall = 1;
    bit isTerminator = 1;
    bit isReturn = 1;
    bit isBarrier = 1;
    bit hasExtraSrcRegAllocReq = 1;
    bit isCodeGenOnly = 1;
}
```

```
def calltarget : Operand<iPTR> {
    let EncoderMethod = "getJumpTargetOpValue";
    let OperandType = "OPERAND_PCREL";
}
```

```
let Predicates = [Ch9_1] in {
// Jump and Link (Call)
let isCall=1, hasDelaySlot=1 in {
    // @JumpLink {
    class JumpLink<bits<8> op, string instr_asm>:
        FJ<op, (outs), (ins calltarget:$target, variable_ops),
        !strconcat(instr_asm, "\t$target"), [(Cpu0JmpLink imm:$target)],
        IIBranch> {
    // #if CH >= CH10_1 2
        let DecoderMethod = "DecodeJumpTarget";
    // #endif
        }
    // @JumpLink }

    class JumpLinkReg<bits<8> op, string instr_asm,
                    RegisterClass RC>:
        FA<op, (outs), (ins RC:$rb, variable_ops),
        !strconcat(instr_asm, "\t$rb"), [(Cpu0JmpLink RC:$rb)], IIBranch> {
        let rc = 0;
        let ra = 14;
        let shamt = 0;
    }
```

(continues on next page)

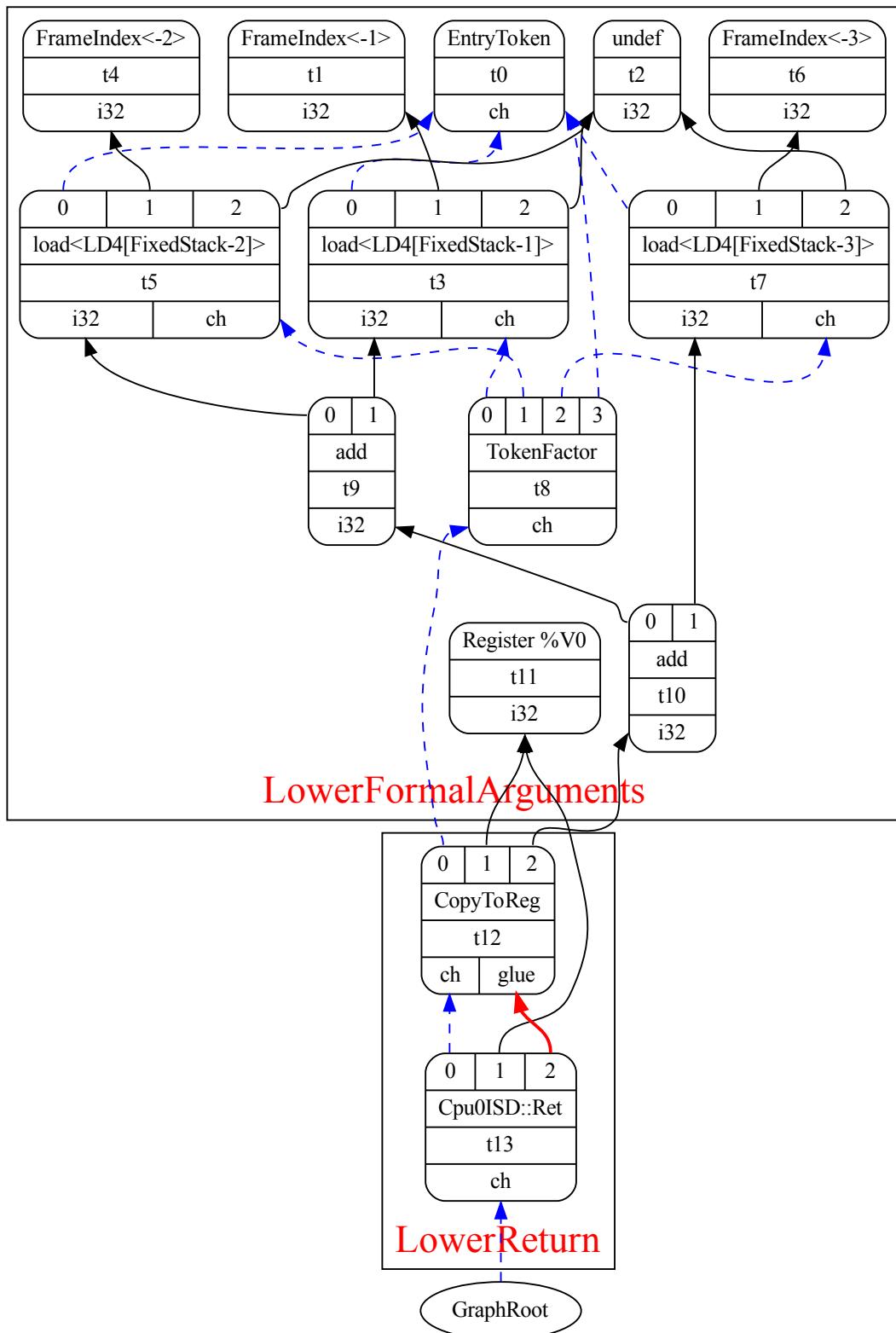
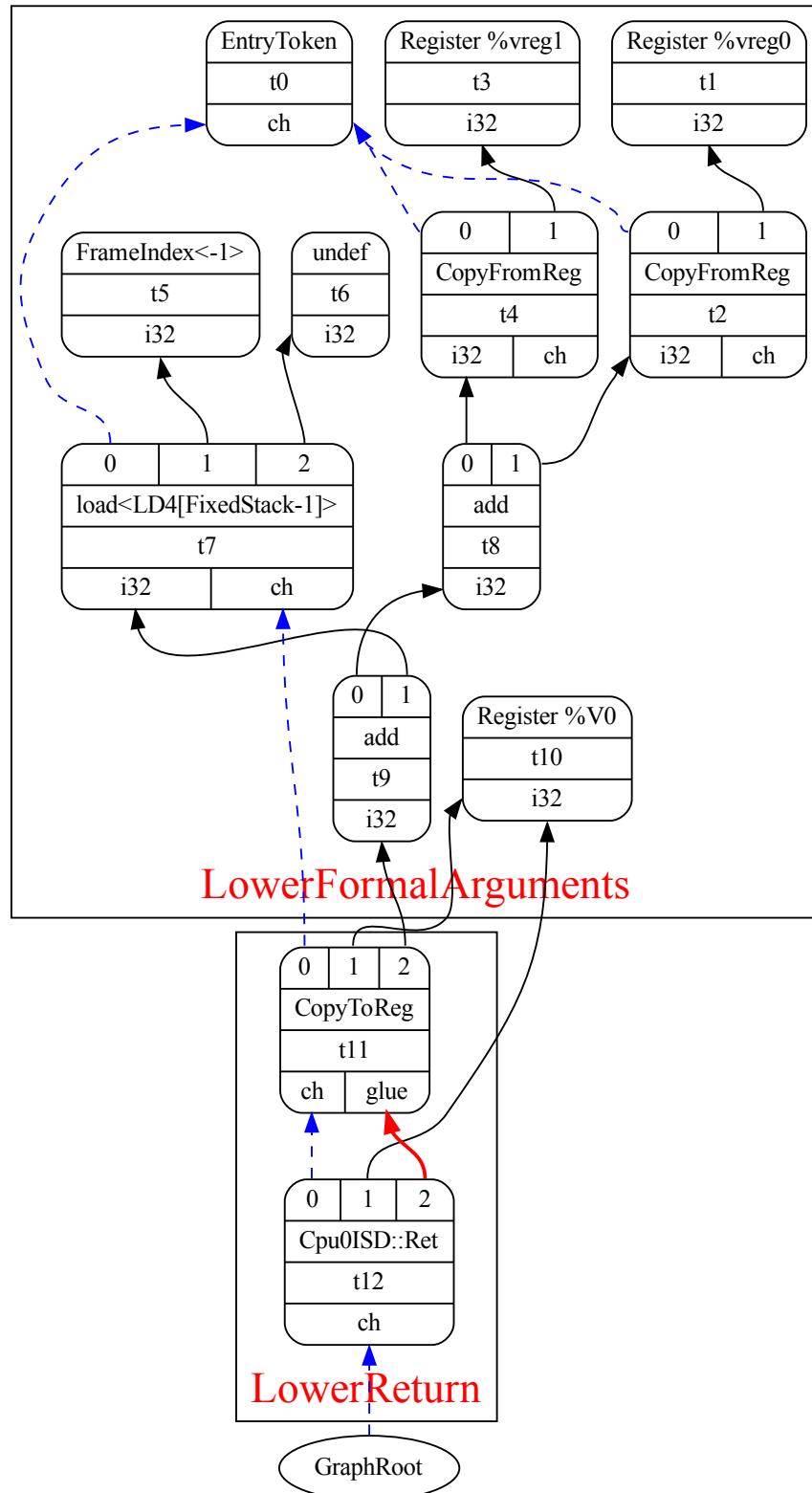


Fig. 9.3: Incoming arguments DAG created for `ch9_incoming.cpp` with `-cpu0-s32-calls=true`

9.2. Load Incoming Arguments from Stack Frame



(continued from previous page)

}

```
/// Jump & link and Return Instructions
let Predicates = [Ch9_1] in {
def JSUB      : JumpLink<0x3b, "jsub">;
}
```

```
let Predicates = [Ch9_1] in {
def JALR     : JumpLinkReg<0x39, "jalr", GPROut>;
}
```

```
let Predicates = [Ch9_1] in {
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
       (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
       (JSUB texternalsym:$dst)>;
```

}

[Index/chapters/Chapter9_1/Cpu0MCInstLower.cpp](#)

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
    MCSymbolRefExpr::VariantKind Kind = MCSymbolRefExpr::VK_None;
    Cpu0MCE Expr::Cpu0ExprKind TargetKind = Cpu0MCE Expr::CEK_None;
    const MCSymbol *Symbol;

    switch(MO.getTargetFlags()) {
```

```
        case Cpu0II::MO_GOT_CALL:
            TargetKind = Cpu0MCE Expr::CEK_GOT_CALL;
            break;
```

```
        ...
    }
    switch (MOTy) {
        ...
    }
```

```
        case MachineOperand::MO_ExternalSymbol:
            Symbol = AsmPrinter.GetExternalSymbolSymbol(MO.getSymbolName());
            Offset += MO.getOffset();
            break;
```

```
        ...
    }
    ...
}
```

```
MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                         unsigned offset) const {
    MachineOperandType MOTy = MO.getType();

    switch (MOTy) {
        // @2
    }

    case MachineOperand::MO_ExternalSymbol:
        return LowerSymbolOperand(MO, MOTy, offset);

    ...
}
```

[Index/chapters/Chapter9_1/MCTargetDesc/Cpu0AsmBackend.cpp](#)

```
// Prepare value for the target space for it
static unsigned adjustFixupValue(const MCFixup &Fixup, uint64_t Value,
                                 MCContext &Ctx) {

    unsigned Kind = Fixup.getKind();

    // Add/subtract and shift
    switch (Kind) {

        case Cpu0::fixup_Cpu0_CALL16:
            ...
    }
}
```

[Index/chapters/Chapter9_1/MCTargetDesc/Cpu0ELFOBJECTWriter.cpp](#)

```
unsigned Cpu0ELFOBJECTWriter::getRelocType(MCContext &Ctx,
                                            const MCValue &Target,
                                            const MCFixup &Fixup,
                                            bool IsPCRel) const {
    // determine the type of the relocation
    unsigned Type = (unsigned)ELF::R_CPU0_NONE;
    unsigned Kind = (unsigned)Fixup.getKind();

    switch (Kind) {

        case Cpu0::fixup_Cpu0_CALL16:
            Type = ELF::R_CPU0_CALL16;
            break;
    }
}
```

```
...
}
...
}
```

Ibdex/chapters/Chapter9_1/MCTargetDesc/Cpu0FixupKinds.h

```
enum Fixups {
    ...
    // resulting in - R_CPU0_CALL16.
    fixup_Cpu0_CALL16,
    ...
    .
}
```

Ibdex/chapters/Chapter9_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
                     SmallVectorImpl<MCFixup> &Fixups,
                     const MCSubtargetInfo &STI) const {

    if (Opcode == Cpu0::JSUB || Opcode == Cpu0::JMP || Opcode == Cpu0::BAL)
    #elif CH >= CH8_2 //1
    if (Opcode == Cpu0::JMP || Opcode == Cpu0::BAL)

        Fixups.push_back(MCFixup::create(0, Expr,
                                         MCFixupKind(Cpu0::fixup_Cpu0_PC24)));
    ...

}

unsigned Cpu0MCCodeEmitter::
getExprOpValue(const MCExpr *Expr, SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {

    // switch(cast<MCSymbolRefExpr>(Expr)->getKind()) {

        case Cpu0MCExpr::CEK_GOT_CALL:
            FixupKind = Cpu0::fixup_Cpu0_CALL16;
            break;

        ...
    }
    ...
}
```

Ibdex/chapters/Chapter9_1/Cpu0MachineFunction.h

```

/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
    Cpu0FunctionInfo(MachineFunction& MF)
        : MF(MF),
        VarArgsFrameIndex(0),
        InArgFIRange(std::make_pair(-1, 0)),
        OutArgFIRange(std::make_pair(-1, 0)), GPFI(0), DynAllocFI(0),
        isInArgFI(int FI) const {
            return FI <= InArgFIRange.first && FI >= InArgFIRange.second;
        }
        void setLastInArgFI(int FI) { InArgFIRange.second = FI; }
        bool isOutArgFI(int FI) const {
            return FI <= OutArgFIRange.first && FI >= OutArgFIRange.second;
        }
        int getGPFI() const { return GPFI; }
        void setGPFI(int FI) { GPFI = FI; }
        bool isGPFI(int FI) const { return GPFI && GPFI == FI; }

        bool isDynAllocFI(int FI) const { return DynAllocFI && DynAllocFI == FI; }

        // Range of frame object indices.
        // InArgFIRange: Range of indices of all frame objects created during call to
        //                 LowerFormalArguments.
        // OutArgFIRange: Range of indices of all frame objects created during call to
        //                 LowerCall except for the frame object for restoring $gp.
        std::pair<int, int> InArgFIRange, OutArgFIRange;

        mutable int DynAllocFI; // Frame index of dynamically allocated stack area.
    ...
};

```

Ibdex/chapters/Chapter9_1/Cpu0SEFrameLowering.h

```

bool spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator MI,
                               ArrayRef<CalleeSavedInfo> CSI,
                               const TargetRegisterInfo *TRI) const override;

```

Ibdex/chapters/Chapter9_1/Cpu0SEFrameLowering.cpp

```

bool Cpu0SEFrameLowering::
spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                           MachineBasicBlock::iterator MI,
                           ArrayRef<CalleeSavedInfo> CSI,
                           const TargetRegisterInfo *TRI) const {
    MachineFunction *MF = MBB.getParent();
    MachineBasicBlock *EntryBlock = &MF->front();
    const TargetInstrInfo &TII = *MF->getSubtarget().getInstrInfo();

    for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
        // Add the callee-saved register as live-in. Do not add if the register is
        // LR and return address is taken, because it has already been added in
        // method Cpu0TargetLowering::LowerRETURNADDR.
        // It's killed at the spill, unless the register is LR and return address
        // is taken.
        unsigned Reg = CSI[i].getReg();
        bool IsRAAndRetAddrIsTaken = (Reg == Cpu0::LR)
            && MF->getFrameInfo().isReturnAddressTaken();
        if (!IsRAAndRetAddrIsTaken)
            EntryBlock->addLiveIn(Reg);

        // Insert the spill to the stack frame.
        bool IsKill = !IsRAAndRetAddrIsTaken;
        const TargetRegisterClass *RC = TRI->getMinimalPhysRegClass(Reg);
        TII.storeRegToStackSlot(*EntryBlock, MI, Reg, IsKill,
                               CSI[i].getFrameIdx(), RC, TRI);
    }

    return true;
}

```

Both *JSUB* and *JALR*, defined in *Cpu0InstrInfo.td* as shown above, use the *Cpu0JmpLink* node. They are distinguishable by their operand types: *JSUB* uses an *imm* (immediate) operand, while *JALR* uses a register operand.

Ibdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```

let Predicates = [Ch9_1] in {
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
           (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
           (JSUB texternalsym:$dst)>;

```

The code instructs TableGen to generate pattern-matching logic that first matches the “*imm*” operand for the “*tglobaladdr*” pattern. If that match fails, it then attempts to match the “*texternalsym*” pattern.

A user-defined function belongs to the “*tglobaladdr*” category. For example, the function *sum_i(...)* defined in *ch9_1.cpp* falls under “*tglobaladdr*”.

On the other hand, functions implicitly used by LLVM, such as *memcpy*, belong to “*texternalsym*”. The *memcpy* function is typically generated when defining a long string. The file *ch9_1_2.cpp* is an example that triggers a call to *memcpy*. This will be shown in the next section with the *Chapter9_2* example code.

The file *Cpu0GenDAGISel.inc* contains the pattern-matching information for *JSUB* and *JALR*, which is generated by TableGen as follows:

```

/*SwitchOpcode*/ 74, TARGET_VAL(Cpu0ISD::JmpLink), // ->734
/*660*/
OPC_RecordNode, // #0 = 'Cpu0JmpLink' chained node
/*661*/
OPC_CaptureGlueInput,
/*662*/
OPC_RecordChild1, // #1 = $target
/*663*/
OPC_Scope, 57, /*->722*/ // 2 children in Scope
/*665*/
OPC_MoveChild, 1,
/*667*/
OPC_SwitchOpcode /*3 cases */, 22, TARGET_VAL(ISD::Constant),
// ->693
/*671*/
OPC_MoveParent,
/*672*/
OPC_EmitMergeInputChains1_0,
/*673*/
OPC_EmitConvertToTarget, 1,
/*675*/
OPC_Scope, 7, /*->684*/ // 2 children in Scope
/*684*/
/*Scope*/ 7, /*->692*/
/*685*/
OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 2,
    // Src: (Cpu0JmpLink (imm:iPTR):$target) - Complexity = 6
    // Dst: (JSUB (imm:iPTR):$target)
/*692*/
0, /*End of Scope*/
/*SwitchOpcode*/ 11, TARGET_VAL(ISD::TargetGlobalAddress), // ->707
/*696*/
OPC_CheckType, MVT::i32,
/*698*/
OPC_MoveParent,
/*699*/
OPC_EmitMergeInputChains1_0,
/*700*/
OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink (tglobaladdr:i32):$dst) - Complexity = 6
    // Dst: (JSUB (tglobaladdr:i32):$dst)
/*710*/
/*SwitchOpcode*/ 11, TARGET_VAL(ISD::TargetExternalSymbol), // ->721
OPC_CheckType, MVT::i32,
/*712*/
OPC_MoveParent,
/*713*/
OPC_EmitMergeInputChains1_0,
/*714*/
OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink (texternalsym:i32):$dst) - Complexity = 6
    // Dst: (JSUB (texternalsym:i32):$dst)
    0, // EndSwitchOpcode
/*722*/
/*Scope*/ 10, /*->733*/
/*723*/
OPC_CheckChild1Type, MVT::i32,
/*725*/
OPC_EmitMergeInputChains1_0,
/*726*/
OPC_MorphNodeTo, TARGET_VAL(Cpu0::JALR), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink CPURegs:i32:$rb) - Complexity = 3
    // Dst: (JALR CPURegs:i32:$rb)
/*733*/
0, /*End of Scope*/

```

After applying the above changes, you can run *Chapter9_1*/ with *ch9_1.cpp* and observe the results as shown below:

```
118-165-79-83:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch9_1.bc -o ch9_1.cpu0.s
Assertion failed: ((CLI.IsTailCall || InVals.size() == CLI.Ins.size()) &&
"LowerCall didn't emit the correct number of values!"), function LowerCallTo,
file /Users/Jonathan/llvm/test/llvm/lib/CodeGen/SelectionDAG/SelectionDAGBuilder.
cpp, ...
...
0. Program arguments: /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
1. Running pass 'Function Pass Manager' on module 'ch9_1.bc'.
2. Running pass 'CPU0 DAG->DAG Pattern Instruction Selection' on function
'@main'
Illegal instruction: 4
```

Now, the `LowerFormalArguments()` has the correct number, but `LowerCall()` has not the correct number of values!

9.3 Store Outgoing Arguments to Stack Frame

Fig. 9.2 illustrates two steps involved in argument passing:

1. Storing outgoing arguments in the caller function.
2. Loading incoming arguments in the callee function.

In the previous section, we implemented `LowerFormalArguments()` to handle “**loading incoming arguments**” in the callee function.

Now, we will implement the part responsible for “**storing outgoing arguments**” in the caller function.

This task is handled by the `LowerCall()` function. Its implementation is shown below:

Ibdex/chapters/Chapter9_2/Cpu0MachineFunction.h

```
/// Create a MachinePointerInfo that has an ExternalSymbolPseudoSourceValue
/// object representing a GOT entry for an external function.
MachinePointerInfo callPtrInfo(const char *ES);

/// Create a MachinePointerInfo that has a GlobalValuePseudoSourceValue object
/// representing a GOT entry for a global function.
MachinePointerInfo callPtrInfo(const GlobalValue *GV);
```

Ibdex/chapters/Chapter9_2/Cpu0MachineFunction.cpp

```
MachinePointerInfo Cpu0FunctionInfo::callPtrInfo(const char *ES) {
    return MachinePointerInfo(MF.getPSVManager().getExternalSymbolCallEntry(ES));
}

MachinePointerInfo Cpu0FunctionInfo::callPtrInfo(const GlobalValue *GV) {
    return MachinePointerInfo(MF.getPSVManager().getGlobalValueCallEntry(GV));
}
```

Ibdex/chapters/Chapter9_2/Cpu0ISelLowering.h

```

/// This function fills Ops, which is the list of operands that will later
/// be used when a function call node is created. It also generates
/// copyToReg nodes to set up argument registers.
virtual void
getOpndList(SmallVectorImpl<SDValue> &Ops,
            std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
            bool IsPICCall, bool GlobalOrExternal, bool InternalLinkage,
            CallLoweringInfo &CLI, SDValue Callee, SDValue Chain) const;

/// Cpu0CC - This class provides methods used to analyze formal and call
/// arguments and inquire about calling convention information.
class Cpu0CC {

    void analyzeCallOperands(const SmallVectorImpl<ISD::OutputArg> &Outs,
                            bool IsVarArg, bool IsSoftFloat,
                            const SDNode *CallNode,
                            std::vector<ArgListEntry> &FuncArgs);

};

Cpu0CC::SpecialCallingConvType getSpecialCallingConv(SDValue Callee) const;

// Lower Operand helpers
SDValue LowerCallResult(SDValue Chain, SDValue InFlag,
                        CallingConv::ID CallConv, bool isVarArg,
                        const SmallVectorImpl<ISD::InputArg> &Ins,
                        const SDLoc &dl, SelectionDAG &DAG,
                        SmallVectorImpl<SDValue> &InVals,
                        const SDNode *CallNode, const Type *RetTy) const;

/// passByValArg - Pass a byval argument in registers or on stack.
void passByValArg(SDValue Chain, const SDLoc &DL,
                   std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
                   SmallVectorImpl<SDValue> &MemOpChains, SDValue StackPtr,
                   MachineFrameInfo &MFI, SelectionDAG &DAG, SDValue Arg,
                   const Cpu0CC &CC, const ByValArgInfo &ByVal,
                   const ISD::ArgFlagsTy &Flags, bool isLittle) const;

SDValue passArgOnStack(SDValue StackPtr, unsigned Offset, SDValue Chain,
                      SDValue Arg, const SDLoc &DL, bool IsTailCall,
                      SelectionDAG &DAG) const;

bool CanLowerReturn(CallingConv::ID CallConv, MachineFunction &MF,
                    bool isVarArg,
                    const SmallVectorImpl<ISD::OutputArg> &Outs,
                    LLVMContext &Context) const override;

```

Ibdex/chapters/Chapter9_2/Cpu0ISelLowering.cpp

```

SDValue
Cpu0TargetLowering::passArgOnStack(SDValue StackPtr, unsigned Offset,
                                    SDValue Chain, SDValue Arg, const SDLoc &DL,
                                    bool IsTailCall, SelectionDAG &DAG) const {
    if (!IsTailCall) {
        SDValue PtrOff =
            DAG.getNode(ISD::ADD, DL, getPointerTy(DAG.getDataLayout()), StackPtr,
                        DAG.getIntPtrConstant(Offset, DL));
        return DAG.getStore(Chain, DL, Arg, PtrOff, MachinePointerInfo());
    }

    MachineFrameInfo &MFI = DAG.getMachineFunction().getFrameInfo();
    int FI = MFI.CreateFixedObject(Arg.getValueSizeInBits() / 8, Offset, false);
    SDValue FIN = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
    return DAG.getStore(Chain, DL, Arg, FIN, MachinePointerInfo(),
                        /* Alignment = */ 0, MachineMemOperand::MOVolatile);
}

void Cpu0TargetLowering::
getOpndList(SmallVectorImpl<SDValue> &Ops,
            std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
            bool IsPICCall, bool GlobalOrExternal, bool InternalLinkage,
            CallLoweringInfo &CLI, SDValue Callee, SDValue Chain) const {
    // T9 should contain the address of the callee function if
    // -relocation-model=pic or it is an indirect call.
    if (IsPICCall || !GlobalOrExternal) {
        unsigned T9Reg = Cpu0::T9;
        RegsToPass.push_front(std::make_pair(T9Reg, Callee));
    } else
        Ops.push_back(Callee);

    // Insert node "GP copy globalreg" before call to function.
    //
    // R_CPU0_CALL* operators (emitted when non-internal functions are called
    // in PIC mode) allow symbols to be resolved via lazy binding.
    // The lazy binding stub requires GP to point to the GOT.
    if (IsPICCall && !InternalLinkage) {
        unsigned GPRReg = Cpu0::GP;
        EVT Ty = MVT::i32;
        RegsToPass.push_back(std::make_pair(GPRReg, getGlobalReg(CLI.DAG, Ty)));
    }

    // Build a sequence of copy-to-reg nodes chained together with token
    // chain and flag operands which copy the outgoing args into registers.
    // The InFlag is necessary since all emitted instructions must be
    // stuck together.
    SDValue InFlag;

    for (unsigned i = 0, e = RegsToPass.size(); i != e; ++i) {
        Chain = CLI.DAG.getCopyToReg(Chain, CLI.DL, RegsToPass[i].first,
                                     RegsToPass[i].second, InFlag);
    }
}

```

(continues on next page)

(continued from previous page)

```

    InFlag = Chain.getValue(1);
}

// Add argument registers to the end of the list so that they are
// known live into the call.
for (unsigned i = 0, e = RegsToPass.size(); i != e; ++i)
    Ops.push_back(CLI.DAG.getRegister(RegsToPass[i].first,
                                      RegsToPass[i].second.getValueType()));

// Add a register mask operand representing the call-preserved registers.
const TargetRegisterInfo *TRI = Subtarget.getRegisterInfo();
const uint32_t *Mask =
    TRI->getCallPreservedMask(CLI.DAG.getMachineFunction(), CLI.CallConv);
assert(Mask && "Missing call preserved mask for calling convention");
Ops.push_back(CLI.DAG.getRegisterMask(Mask));

if (InFlag.getNode())
    Ops.push_back(InFlag);
}

```

```

/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {
    SelectionDAG &DAG = CLI.DAG;
    SDLoc DL = CLI.DL;
    SmallVectorImpl<ISD::OutputArg> &Outs = CLI.Outs;
    SmallVectorImpl<SDValue> &OutVals = CLI.OutVals;
    SmallVectorImpl<ISD::InputArg> &Ins = CLI.Ins;
    SDValue Chain = CLI.Chain;
    SDValue Callee = CLI.Callee;
    bool &IsTailCall = CLI.IsTailCall;
    CallingConv::ID CallConv = CLI.CallConv;
    bool IsVarArg = CLI.IsVarArg;

    MachineFunction &MF = DAG.getMachineFunction();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    const TargetFrameLowering *TFL = MF.getSubtarget().getFrameLowering();
    Cpu0FunctionInfo *FuncInfo = MF.getInfo<Cpu0FunctionInfo>();
    bool IsPIC = isPositionIndependent();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    // Analyze operands of the call, assigning locations to each operand.
    SmallVector<CCValAssign, 16> ArgLocs;
    CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                  ArgLocs, *DAG.getContext());
    Cpu0CC::SpecialCallingConvType SpecialCallingConv =
        getSpecialCallingConv(Callee);
    Cpu0CC Cpu0CCInfo(CallConv, ABI::IsO32(),
                      CCInfo, SpecialCallingConv);

```

(continues on next page)

(continued from previous page)

```
Cpu0CCInfo.analyzeCallOperands(Outs, IsVarArg,
                               Subtarget.abiUsesSoftFloat(),
                               Callee.getNode(), CLI.getArgs());

// Get a count of how many bytes are to be pushed on the stack.
unsigned NextStackOffset = CCInfo.getNextStackOffset();

//@TailCall 1 {
// Check if it's really possible to do a tail call.
if (IsTailCall)
    IsTailCall =
        isEligibleForTailCallOptimization(Cpu0CCInfo, NextStackOffset,
                                           *MF.getInfo<Cpu0FunctionInfo>());

if (!IsTailCall && CLI.CB && CLI.CB->isMustTailCall())
    report_fatal_error("failed to perform tail call elimination on a call "
                       "site marked musttail");

if (IsTailCall)
    ++NumTailCalls;
//@TailCall 1 }

// Chain is the output chain of the last Load/Store or CopyToReg node.
// ByValChain is the output chain of the last Memcpy node created for copying
// byval arguments to the stack.
unsigned StackAlignment = TFL->getStackAlignment();
NextStackOffset = alignTo(NextStackOffset, StackAlignment);
SDValue NextStackOffsetVal = DAG.getIntPtrConstant(NextStackOffset, DL, true);

//@TailCall 2 {
if (!IsTailCall)
    Chain = DAG.getCALLSEQ_START(Chain, NextStackOffset, 0, DL);
//@TailCall 2 }

SDValue StackPtr =
    DAG.getCopyFromReg(Chain, DL, Cpu0::SP,
                       getPointerTy(DAG.getDataLayout()));

// With EABI is it possible to have 16 args on registers.
std::deque< std::pair<unsigned, SDValue> > RegsToPass;
SmallVector<SDValue, 8> MemOpChains;
Cpu0CC::byval_iterator ByValArg = Cpu0CCInfo.byval_begin();

//@1 {
// Walk the register/memloc assignments, inserting copies/loads.
for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
//@1 }
    SDValue Arg = OutVals[i];
    CCValAssign &VA = ArgLocs[i];
    MVT LocVT = VA.getLocVT();
    ISD::ArgFlagsTy Flags = Outs[i].Flags;
```

(continues on next page)

(continued from previous page)

```
//@ByVal Arg {
if (Flags.isByVal()) {
    assert(Flags.getByValSize() &&
           "ByVal args of size 0 should have been ignored by front-end.");
    assert(ByValArg != Cpu0CCInfo.byval_end());
    assert(!IsTailCall &&
           "Do not tail-call optimize if there is a byval argument.");
    passByValArg(Chain, DL, RegsToPass, MemOpChains, StackPtr, MFI, DAG, Arg,
                  Cpu0CCInfo, *ByValArg, Flags, Subtarget.isLittle());
    ++ByValArg;
    continue;
}
//@ByVal Arg }

// Promote the value if needed.
switch (VA.getLocInfo()) {
default: llvm_unreachable("Unknown loc info!");
case CCValAssign::Full:
    break;
case CCValAssign::SExt:
    Arg = DAG.getNode(ISD::SIGN_EXTEND, DL, LocVT, Arg);
    break;
case CCValAssign::ZExt:
    Arg = DAG.getNode(ISD::ZERO_EXTEND, DL, LocVT, Arg);
    break;
case CCValAssign::AExt:
    Arg = DAG.getNode(ISD::ANY_EXTEND, DL, LocVT, Arg);
    break;
}

// Arguments that can be passed on register must be kept at
// RegsToPass vector
if (VA.isRegLoc()) {
    RegsToPass.push_back(std::make_pair(VA.getLocReg(), Arg));
    continue;
}

// Register can't get to this point...
assert(VA.isMemLoc());

// emit ISD::STORE which stores the
// parameter value to a stack Location
MemOpChains.push_back(passArgOnStack(StackPtr, VA.getLocMemOffset(),
                                       Chain, Arg, DL, IsTailCall, DAG));
}

// Transform all store nodes into one single node because all store
// nodes are independent of each other.
if (!MemOpChains.empty())
    Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, MemOpChains);

// If the callee is a GlobalAddress/ExternalSymbol node (quite common, every
```

(continues on next page)

(continued from previous page)

```

// direct call is) turn it into a TargetGlobalAddress/TargetExternalSymbol
// node so that legalize doesn't hack it.
bool IsPICCall = IsPIC; // true if calls are translated to
                        // jalr $t9
bool GlobalOrExternal = false, InternalLinkage = false;
EVT Ty = Callee.getValueType();

if (GlobalAddressSDNode *G = dyn_cast<GlobalAddressSDNode>(Callee)) {
    if (IsPICCall) {
        const GlobalValue *Val = G->getGlobal();
        InternalLinkage = Val->hasInternalLinkage();

        if (InternalLinkage)
            Callee = getAddrLocal(G, Ty, DAG);
        else
            Callee = getAddrGlobal(G, Ty, DAG, Cpu0II::MO_GOT_CALL, Chain,
                                   FuncInfo->callPtrInfo(Val));
    } else
        Callee = DAG.getTargetGlobalAddress(G->getGlobal(), DL,
                                            getPointerTy(DAG.getDataLayout()), 0,
                                            Cpu0II::MO_NO_FLAG);
    GlobalOrExternal = true;
}
else if (ExternalSymbolSDNode *S = dyn_cast<ExternalSymbolSDNode>(Callee)) {
    const char *Sym = S->getSymbol();

    if (!IsPIC) // static
        Callee = DAG.getTargetExternalSymbol(Sym,
                                              getPointerTy(DAG.getDataLayout()),
                                              Cpu0II::MO_NO_FLAG);
    else // PIC
        Callee = getAddrGlobal(S, Ty, DAG, Cpu0II::MO_GOT_CALL, Chain,
                               FuncInfo->callPtrInfo(Sym));

    GlobalOrExternal = true;
}

SmallVector<SDValue, 8> Ops(1, Chain);
SDVTList NodeTys = DAG.getVTList(MVT::Other, MVT::Glue);

getOpndList(Ops, RegsToPass, IsPICCall, GlobalOrExternal, InternalLinkage,
            CLI, Callee, Chain);

//@TailCall 3 {
if (IsTailCall)
    return DAG.getNode(Cpu0ISD::TailCall, DL, MVT::Other, Ops);
//@TailCall 3 }

Chain = DAG.getNode(Cpu0ISD::JmpLink, DL, NodeTys, Ops);
SDValue InFlag = Chain.getValue(1);

// Create the CALLSEQ_END node.

```

(continues on next page)

(continued from previous page)

```

Chain = DAG.getCALLSEQ_END(Chain, NextStackOffsetVal,
                           DAG.getIntPtrConstant(0, DL, true), InFlag, DL);
InFlag = Chain.getValue(1);

// Handle result values, copying them out of physregs into vregs that we
// return.
return LowerCallResult(Chain, InFlag, CallConv, IsVarArg,
                       Ins, DL, DAG, InVals, CLI.Callee.getNode(), CLI.RetTy);
}

```

```

/// LowerCallResult - Lower the result values of a call into the
/// appropriate copies out of appropriate physical registers.
SDValue
Cpu0TargetLowering::LowerCallResult(SDValue Chain, SDValue InFlag,
                                    CallingConv::ID CallConv, bool IsVarArg,
                                    const SmallVectorImpl<ISD::InputArg> &Ins,
                                    const SDLoc &DL, SelectionDAG &DAG,
                                    SmallVectorImpl<SDValue> &InVals,
                                    const SDNode *CallNode,
                                    const Type *RetTy) const {
    // Assign locations to each value returned by this call.
    SmallVector<CCValAssign, 16> RVLocs;
    CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                   RVLocs, *DAG.getContext());

    Cpu0CC Cpu0CCInfo(CallConv, ABI.IsO32(), CCInfo);

    Cpu0CCInfo.analyzeCallResult(Ins, Subtarget.abiUsesSoftFloat(),
                                 CallNode, RetTy);

    // Copy all of the result registers out of their specified physreg.
    for (unsigned i = 0; i != RVLocs.size(); ++i) {
        SDValue Val = DAG.getCopyFromReg(Chain, DL, RVLocs[i].getLocReg(),
                                         RVLocs[i].getLocVT(), InFlag);
        Chain = Val.getValue(1);
        InFlag = Val.getValue(2);

        if (RVLocs[i].getValVT() != RVLocs[i].getLocVT())
            Val = DAG.getNode(ISD::BITCAST, DL, RVLocs[i].getValVT(), Val);

        InVals.push_back(Val);
    }

    return Chain;
}

```

```

bool
Cpu0TargetLowering::CanLowerReturn(CallingConv::ID CallConv,
                                   MachineFunction &MF, bool IsVarArg,
                                   const SmallVectorImpl<ISD::OutputArg> &Outs,
                                   LLVMContext &Context) const {

```

(continues on next page)

(continued from previous page)

```

SmallVector<CCValAssign, 16> RVLocs;
CCState CCInfo(CallConv, IsVarArg, MF,
               RVLocs, Context);
return CCInfo.CheckReturn(Outs, RetCC_Cpu0);
}

```

```

Cpu0TargetLowering::Cpu0CC::SpecialCallingConvType
Cpu0TargetLowering::getSpecialCallingConv(SDValue Callee) const {
    Cpu0CC::SpecialCallingConvType SpecialCallingConv =
        Cpu0CC::NoSpecialCallingConv;
    return SpecialCallingConv;
}

```

```

void Cpu0TargetLowering::Cpu0CC::
analyzeCallOperands(const SmallVectorImpl<ISD::OutputArg> &Args,
                     bool IsVarArg, bool IsSoftFloat, const SDNode *CallNode,
                     std::vector<ArgListEntry> &FuncArgs) {
//@analyzeCallOperands body {
    assert((CallConv != CallingConv::Fast || !IsVarArg) &&
           "CallingConv::Fast shouldn't be used for vararg functions.");

    unsigned NumOpnds = Args.size();
    llvm::CCAssignFn *FixedFn = fixedArgFn();

    //@3 {
    for (unsigned I = 0; I != NumOpnds; ++I) {
        //@3
        MVT ArgVT = Args[I].VT;
        ISD::ArgFlagsTy ArgFlags = Args[I].Flags;
        bool R;

        if (ArgFlags.isByVal()) {
            handleByValArg(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags);
            continue;
        }

        {
            MVT RegVT = getRegVT(ArgVT, IsSoftFloat);
            R = FixedFn(I, ArgVT, RegVT, CCValAssign::Full, ArgFlags, CCInfo);
        }

        if (R) {
#ifndef NDEBUG
            dbgs() << "Call operand #" << I << " has unhandled type "
                << EVT(ArgVT).getEVTString();
#endif
            llvm_unreachable(nullptr);
        }
    }
}

```

Just like loading incoming arguments from the stack frame, we call *CCInfo(CallConv, ..., ArgLocs, ...)* to obtain outgoing

argument information before entering the “**for loop**”.

The loop structure is almost identical to that in *LowerFormalArguments()*, except that *LowerCall()* creates a “store DAG vector” instead of a “load DAG vector”.

After the “**for loop**”, it generates the instruction `**ld \$t9, %call16(_Z5sum_iiiiii)(\$gp)**` followed by *jalr \$t9* to call the subroutine (where **\$6** is **\$t9**) in PIC (Position Independent Code) mode.

As with loading incoming arguments, we need to implement *storeRegToStackSlot()* in an earlier chapter to handle storing outgoing arguments.

9.3.1 Pseudo Hook Instructions ADJCALLSTACKDOWN and ADJCALLSTACKUP

DAG.getCALLSEQ_START() and *DAG.getCALLSEQ_END()* are invoked before and after the “**for loop**”, respectively. These insert *CALLSEQ_START* and *CALLSEQ_END*, which are later translated into the pseudo machine instructions *ADJCALLSTACKDOWN* and *ADJCALLSTACKUP*.

These pseudo instructions are defined in *Cpu0InstrInfo.td* as shown below:

Ibdex/chapters/Chapter9_2/Cpu0InstrInfo.td

```
def SDT_Cpu0CallSeqStart : SDCallSeqStart<[SDTCisVT<0, i32>]>;
def SDT_Cpu0CallSeqEnd   : SDCallSeqEnd<[SDTCisVT<0, i32>, SDTCisVT<1, i32>]>;
```

```
// These are target-independent nodes, but have target-specific formats.
def callseq_start : SDNode<"ISD::CALLSEQ_START", SDT_Cpu0CallSeqStart,
                     [SDNPHasChain, SDNPOutGlue]>;
def callseq_end   : SDNode<"ISD::CALLSEQ_END", SDT_Cpu0CallSeqEnd,
                     [SDNPHasChain, SDNPOptInGlue, SDNPOutGlue]>;
```

```
//-----
// Pseudo instructions
//-----

let Predicates = [Ch9_2] in {
// As stack alignment is always done with addiu, we need a 16-bit immediate
let Defs = [SP], Uses = [SP] in {
def ADJCALLSTACKDOWN : Cpu0Pseudo<(outs), (ins uimm16:$amt1, uimm16:$amt2),
                      "!ADJCALLSTACKDOWN $amt1",
                      [(callseq_start timm:$amt1, timm:$amt2)]>;
def ADJCALLSTACKUP   : Cpu0Pseudo<(outs), (ins uimm16:$amt1, uimm16:$amt2),
                      "!ADJCALLSTACKUP $amt1",
                      [(callseq_end timm:$amt1, timm:$amt2)]>;
}

//@def CPRESTORE {
// When handling PIC code the assembler needs .cupload and .cprestore
// directives. If the real instructions corresponding these directives
// are used, we have the same behavior, but get also a bunch of warnings
// from the assembler.
let hasSideEffects = 0 in
def CPRESTORE : Cpu0Pseudo<(outs), (ins i32imm:$loc, CPUREgs:$gp),
                           ".cprestore\t$loc", []>;
} // let Predicates = [Ch9_2]
```

With the definition below, `eliminateCallFramePseudoInstr()` will be called when LLVM encounters the pseudo instructions `ADJCALLSTACKDOWN` and `ADJCALLSTACKUP`.

This function simply discards these two pseudo instructions. LLVM will then automatically adjust the stack offset as needed.

Ibdex/chapters/Chapter9_2/Cpu0InstrInfo.cpp

```
Cpu0InstrInfo::Cpu0InstrInfo(const Cpu0Subtarget &STI)
:
Cpu0GenInstrInfo(Cpu0::ADJCALLSTACKDOWN, Cpu0::ADJCALLSTACKUP),
```

Ibdex/chapters/Chapter9_2/Cpu0FrameLowering.h

```
MachineBasicBlock::iterator
eliminateCallFramePseudoInstr(MachineFunction &MF,
                               MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator I) const override;
```

Ibdex/chapters/Chapter9_2/Cpu0FrameLowering.cpp

```
// Eliminate ADJCALLSTACKDOWN, ADJCALLSTACKUP pseudo instructions
MachineBasicBlock::iterator Cpu0FrameLowering::
eliminateCallFramePseudoInstr(MachineFunction &MF, MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator I) const {

    return MBB.erase(I);
}
```

9.3.2 Read LowerCall() with Graphviz's Help

The complete DAGs created for outgoing arguments are shown in Fig. 9.5 for `ch9_outgoing.cpp` with `cpu032I`.

The `LowerCall()` function (excluding the call to `LowerCallResult()`) will generate the DAG nodes shown in Fig. 9.6 for `ch9_outgoing.cpp` with `cpu032I`.

The corresponding code for the DAG nodes `Store` and `TargetGlobalAddress` is listed in the figures. Users can match other DAG nodes to the `LowerCall()` function code accordingly.

By using the Graphviz tool with the `llc` option `-view-dag-combine1-dags`, you can design a small input in C or LLVM IR, then inspect the DAGs to better understand the behavior of `LowerCall()` and `LowerFormalArguments()`.

In the later sub-sections, “Variable Arguments” and “Dynamic Stack Allocation Support”, you can create input examples that demonstrate these features. You can then use the DAGs to confirm your understanding of the logic in these two functions.

For more information about Graphviz, refer to the section “Display LLVM IR Nodes with Graphviz” in Chapter 4, *Arithmetic and Logic Instructions*.

The DAG diagrams can be generated using the `llc` option as shown below:

Ibdex/input/ch9_outgoing.cpp

```
extern int sum_i(int x1);

int call_sum_i() {
    return sum_i(1);
}
```

```
JonathantekiiMac:input Jonathan$ clang -O3 -target mips-unknown-linux-gnu -c ch9_outgoing.cpp -emit-llvm -o ch9_outgoing.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llvm-dis ch9_outgoing.bc -o -
...
define i32 @_Z10call_sum_iv() #0 {
    %1 = tail call i32 @_Z5sum_ii(i32 1)
    ret i32 %1
}
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc -march=cpu0 -mcpu=cpu032I -view-dag-combine1-dags -relocation-model=static -filetype=asm ch9_outgoing.bc -o -
    .text
    .section .mdebug.abiS32
    .previous
    .file "ch9_outgoing.bc"
Writing '/var/folders/rf/8bgdgt9d6vgf5sn8h8_zycd0000gn/T/dag._Z10call_sum_iv-0dfa1.dot'... done.
Running 'Graphviz' program...
```

As mentioned in the previous section, the option `llc -cpu0-s32-calls=true` uses the S32 calling convention, which passes all arguments in registers.

In contrast, the option `llc -cpu0-s32-calls=false` uses the O32 convention, which passes the first two arguments in registers and the remaining arguments on the stack.

The resulting behavior is shown as follows:

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=true -relocation-model=pic -filetype=asm ch9_1.bc -o -
    .text
    .section .mdebug.abiS32
    .previous
    .file "ch9_1.bc"
    .globl      _Z5sum_iiiiii
    .align      2
    .type     _Z5sum_iiiiii,@function
    .ent     _Z5sum_iiiiii      # @_Z5sum_iiiiii
_Z5sum_iiiiii:
    .frame      $fp,32,$lr
    .mask       0x00000000,0
    .set      noreorder
    .cupload     $t9
    .set      nomacro
# BB#0:
```

(continues on next page)

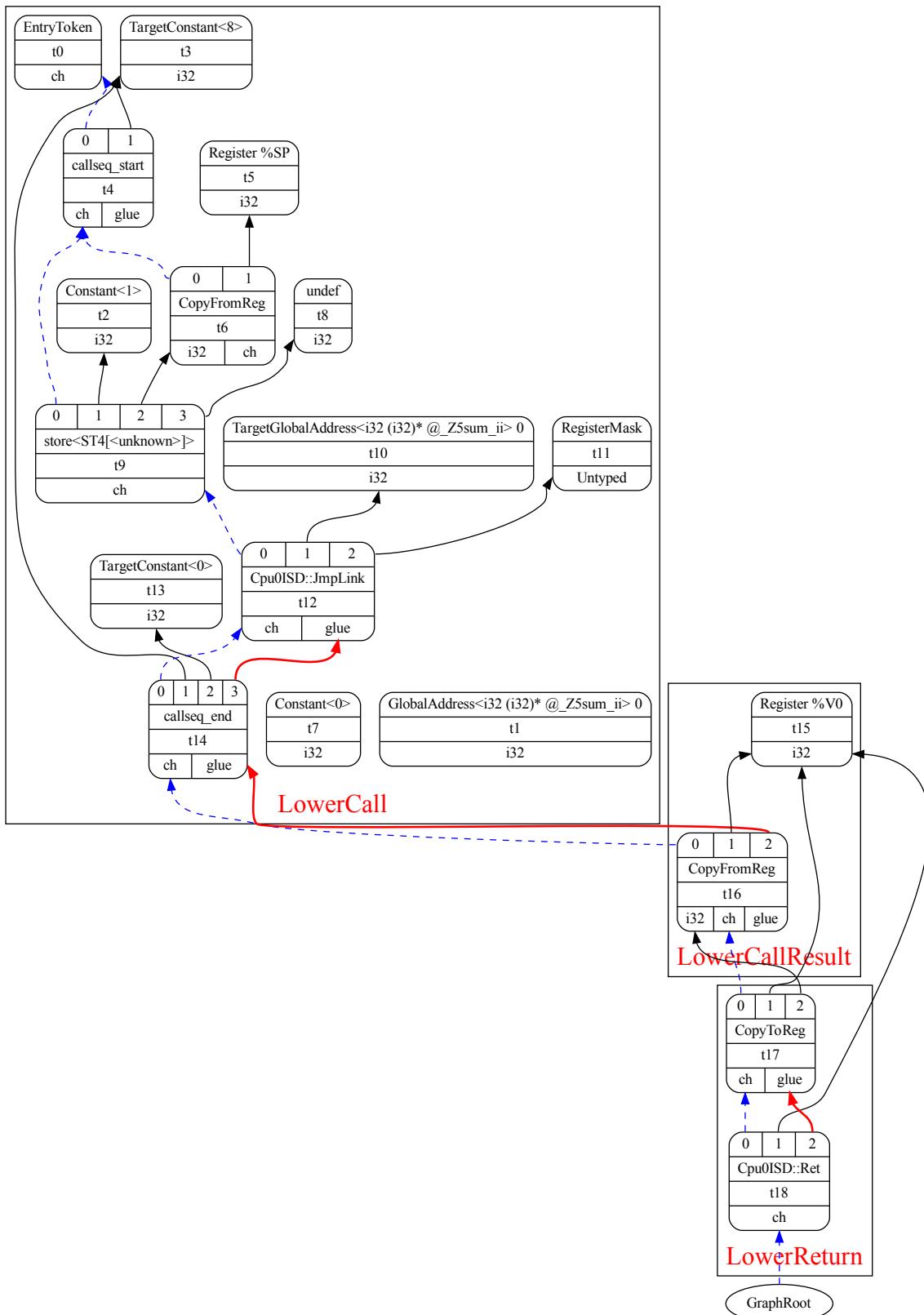


Fig. 9.5: Outgoing arguments DAG (A) created for `ch9_outgoing.cpp` with `-cpu0-s32-calls=true`

9.3. Store Outgoing Arguments to Stack Frame

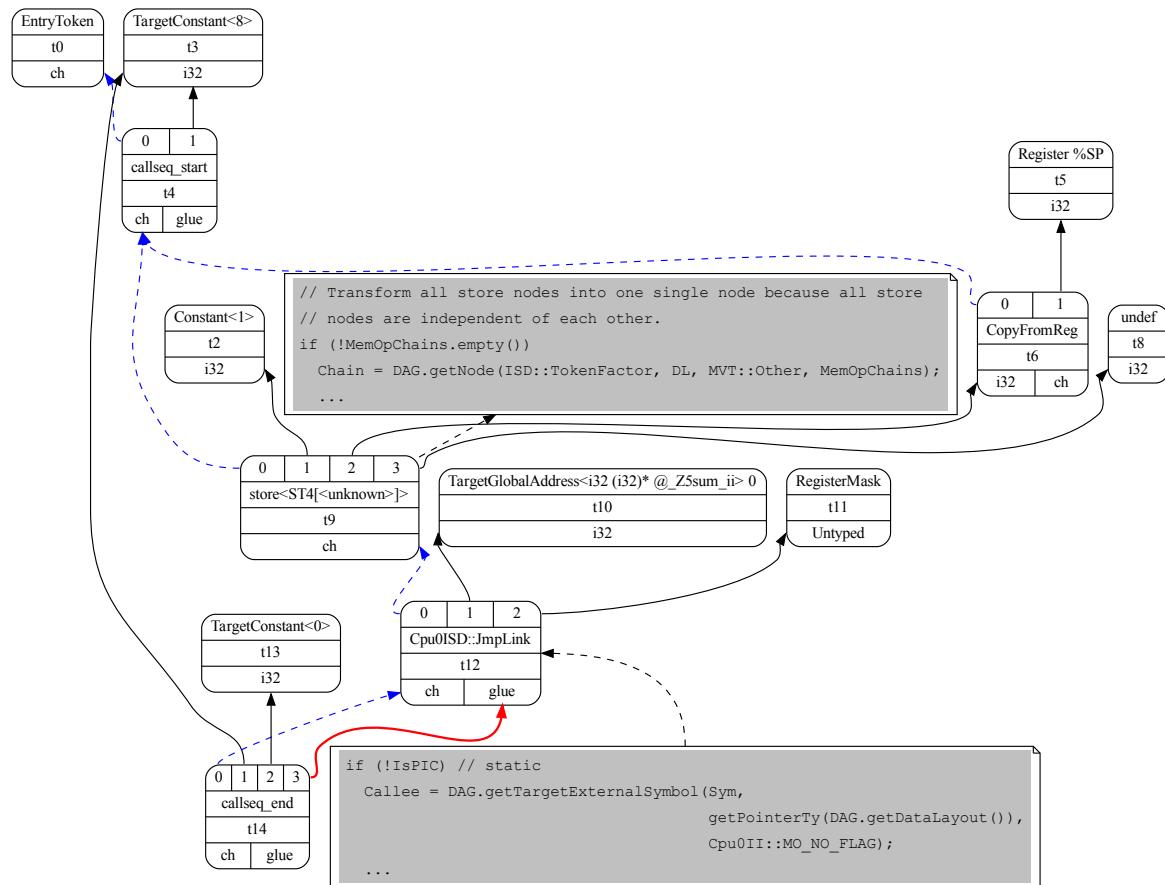


Fig. 9.6: Outgoing arguments DAG (B) created by LowerCall() for ch9_outgoing.cpp with -cpu0-s32-calls=true

(continued from previous page)

```

addiu $sp, $sp, -32
ld    $2, 52($sp)
ld    $3, 48($sp)
ld    $4, 44($sp)
ld    $5, 40($sp)
ld    $t9, 36($sp)
ld    $7, 32($sp)
st    $7, 28($sp)
st    $t9, 24($sp)
st    $5, 20($sp)
st    $4, 16($sp)
st    $3, 12($sp)
lui   $3, %got_hi(gI)
addu $3, $3, $gp
st    $2, 8($sp)
ld    $3, %got_lo(gI) ($3)
ld    $3, 0($3)
ld    $4, 28($sp)
addu $3, $3, $4
ld    $4, 24($sp)
addu $3, $3, $4
ld    $4, 20($sp)
addu $3, $3, $4
ld    $4, 16($sp)
addu $3, $3, $4
ld    $4, 12($sp)
addu $3, $3, $4
addu $2, $3, $2
st    $2, 4($sp)
addiu $sp, $sp, 32
ret   $lr
nop
.set macro
.set reorder
.end _Z5sum_iiiiii

$tmp0:
.size _Z5sum_iiiiii, ($tmp0)-_Z5sum_iiiiii

.globl main
.align 2
.type main,@function
.ent main # @main

main:
.frame $fp,40,$lr
.mask 0x00004000,-4
.set noreorder
.cupload $t9
.set nomacro

# BB#0:
addiu $sp, $sp, -40
st    $lr, 36($sp)      # 4-byte Folded Spill
addiu $2, $zero, 0

```

(continues on next page)

(continued from previous page)

```

st    $2, 32($sp)
addiu $2, $zero, 6
st    $2, 20($sp)
addiu $2, $zero, 5
st    $2, 16($sp)
addiu $2, $zero, 4
st    $2, 12($sp)
addiu $2, $zero, 3
st    $2, 8($sp)
addiu $2, $zero, 2
st    $2, 4($sp)
addiu $2, $zero, 1
st    $2, 0($sp)
ld    $t9, %call16(_Z5sum_iiiiiii)($gp)
jalr $t9
nop
st    $2, 28($sp)
ld    $lr, 36($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 40
ret   $lr
nop
.set macro
.set reorder
.end main

$tmp1:
.size main, ($tmp1)-main

.type gI,@object          # @gI
.data
.globl      gI
.align     2
gI:
.4byte     100             # 0x64
.size gI, 4

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc -march=cpu0 -mcpu=cpu032II -cpu0-s32-calls=false -relocation-model=pic -filetype=asm ch9_1.bc -o -
...
.globl      main
.align     2
.type main,@function
.ent   main                 # @main
main:
.frame      $fp,40,$lr
.mask       0x00004000,-4
.set  noreorder
.cupload    $t9
.set  nomacro
# BB#0:
addiu $sp, $sp, -40
st    $lr, 36($sp)           # 4-byte Folded Spill

```

(continues on next page)

(continued from previous page)

```
addiu $2, $zero, 0
st    $2, 32($sp)
addiu $2, $zero, 6
st    $2, 20($sp)
addiu $2, $zero, 5
st    $2, 16($sp)
addiu $2, $zero, 4
st    $2, 12($sp)
addiu $2, $zero, 3
st    $2, 8($sp)
ld    $t9, %call16(_Z5sum_iiiiii) ($gp)
addiu $4, $zero, 1
addiu $5, $zero, 2
jalr $t9
nop
st    $2, 28($sp)
ld    $lr, 36($sp)          # 4-byte Folded Reload
addiu $sp, $sp, 40
ret    $lr
nop
.set macro
.set reorder
.end main
```

9.3.3 Long and Short String Initialization

In the previous section, we mentioned the *JSUB texternalsym* pattern.

Run *Chapter9_2* with *ch9_1_2.cpp* to observe the following results:

For a long string, LLVM generates a call to `memcpy()` to initialize the string—for example, `char str[81] = "Hello world"`

For a short string, the *call memcpy* is optimized and translated into a direct *store* with a constant value during the optimization stages.

Ibdex/input/ch9_1_2.cpp

```
int main()
{
    char str[81] = "Hello world";
    char s[6] = "Hello";

    return 0;
}
```

(continues on next page)

(continued from previous page)

```
@_ZZ4mainE1s = private unnamed_addr constant [6 x i8] c"Hello\00", align 1

; Function Attrs: nounwind
define i32 @_main() #0 {
entry:
    %retval = alloca i32, align 4
    %str = alloca [81 x i8], align 1
    store i32 0, i32* %retval
    %0 = bitcast [81 x i8]* %str to i8*
    call void @llvm.memcpy.p0i8.p0i8.i32(i8* %0, i8* getelementptr inbounds
        ([81 x i8]* @_ZZ4mainE3str, i32 0, i32 0), i32 81, i32 1, i1 false)
    %1 = bitcast [6 x i8]* %s to i8*
    call void @llvm.memcpy.p0i8.p0i8.i32(i8* %1, i8* getelementptr inbounds
        ([6 x i8]* @_ZZ4mainE1s, i32 0, i32 0), i32 6, i32 1, i1 false)

    ret i32 0
}
```

```
Jonathan@tekiiMac:~/input$ clang -target mips-unknown-linux-gnu -c ch9_1_2.cpp -emit-llvm -o ch9_1_2.bc
```

```
Jonathan@tekiiMac:~/input$ /Users/Jonathan/llvm/test/build
```

```
/bin/llc -march=cpu0 -mcpu=cpu032II -cpu0-s32-calls=true
```

```
-relocation-model=static -filetype=asm ch9_1_2.bc -o -
```

```
.section .mdebug.abi32
```

```
...
```

```
    lui    $2, %hi($_ZZ4mainE3str)
    ori    $2, $2, %lo($_ZZ4mainE3str)
    st    $2, 4($sp)
    addiu $2, $sp, 24
    st    $2, 0($sp)
    jsub  memcpy
    nop
    lui    $2, %hi($_ZZ4mainE1s)
    ori    $2, $2, %lo($_ZZ4mainE1s)
    lbu   $3, 4($2)
    shl   $3, $3, 8
    lbu   $4, 5($2)
    or    $3, $3, $4
    sh    $3, 20($sp)
    lbu   $3, 2($2)
    shl   $3, $3, 8
    lbu   $4, 3($2)
    or    $3, $3, $4
    lbu   $4, 1($2)
    lbu   $2, 0($2)
    shl   $2, $2, 8
    or    $2, $2, $4
    shl   $2, $2, 16
    or    $2, $2, $3
    st    $2, 16($sp)
```

```
...
```

```
.type   $_ZZ4mainE3str,@object  # @_ZZ4mainE3str
```

(continues on next page)

(continued from previous page)

```

.section      .rodata, "a", @progbits
$$_ZZ4mainE3str:
    .asciz      "Hello world\000\000\000\000\000\000\000\000\000\000\000\000\
    \000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\
    \000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\
    \000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000"
    .size $$_ZZ4mainE3str, 81

    .type $_ZZ4mainE1s,@object      # @_ZZ4mainE1s
    .section      .rodata.str1.1, "aMS", @progbits, 1
$_ZZ4mainE1s:
    .asciz      "Hello"
    .size $_ZZ4mainE1s, 6

```

The *call memcpy* for a short string is optimized by LLVM before the “DAG-to-DAG Pattern Instruction Selection” stage.

It is translated into a *store* with a constant value, as shown below:

```

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build
/bin/llc -march=cpu0 -mcpu=cpu032II -cpu0-s32-calls=true
-relocation-model=static -filetype=asm ch9_1_2.bc -debug -o -

Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 35 nodes:
...
0x7fd909030810: <multiple use>
0x7fd909030c10: i32 = Constant<1214606444> // 1214606444=0x48656c6c="Hell"

0x7fd909030910: <multiple use>
0x7fd90902d810: <multiple use>
0x7fd909030d10: ch = store 0x7fd909030810, 0x7fd909030c10, 0x7fd909030910,
0x7fd90902d810<ST4[%1]>

0x7fd909030810: <multiple use>
0x7fd909030e10: i16 = Constant<28416> // 28416=0x6f00="o\0"

...
0x7fd90902d810: <multiple use>
0x7fd909031210: ch = store 0x7fd909030810, 0x7fd909030e10, 0x7fd909031010,
0x7fd90902d810<ST2[%1+4] (align=4)>
...

```

The incoming arguments refer to the *formal arguments* as defined in compiler and programming language literature. The outgoing arguments refer to the *actual arguments* passed during a function call.

Summary as Table: Callee incoming arguments and caller outgoing arguments.

Table 9.1: Callee incoming arguments and caller outgoing arguments

Description	Callee	Caller
Charged Function	LowerFormalArguments()	LowerCall()
Charged Function Created	Create load vectors for incoming arguments	Create store vectors for outgoing arguments

9.4 Structure Type Support

9.4.1 Ordinary Struct Type

The following code in *Chapter9_1*/ and *Chapter3_4*/ supports ordinary structure types in function calls.

Ibdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {
```

```
for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    // The cpu0 ABIs for returning structs by value requires that we copy
    // the sret argument into $v0 for the return. Save the argument into
    // a virtual register so that we can access it from the return points.
    if (Ins[i].Flags.isSRet()) {
        unsigned Reg = Cpu0FI->getSRetReturnReg();
        if (!Reg) {
            Reg = MF.getRegInfo().createVirtualRegister(
                getRegClassFor(MVT::i32));
            Cpu0FI->setSRetReturnReg(Reg);
        }
        SDValue Copy = DAG.getCopyToReg(DAG.getEntryNode(), DL, Reg, InVals[i]);
        Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, Copy, Chain);
        break;
    }
}
```

```
}
```

```
SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                               CallingConv::ID CallConv, bool IsVarArg,
                               const SmallVectorImpl<ISD::OutputArg> &Outs,
                               const SmallVectorImpl<SDValue> &OutVals,
                               const SDLoc &DL, SelectionDAG &DAG) const {
```

```
// The cpu0 ABIs for returning structs by value requires that we copy
// the sret argument into $v0 for the return. We saved the argument into
// a virtual register in the entry block, so now we copy the value out
// and into $v0.
if (MF.getFunction().hasStructRetAttr()) {
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    unsigned Reg = Cpu0FI->getSRetReturnReg();

    if (!Reg)
        llvm_unreachable("sret virtual register not created in the entry block");
    SDValue Val =
        DAG.getCopyFromReg(Chain, DL, Reg, getPointerTy(DAG.getDataLayout()));
    unsigned V0 = Cpu0::V0;

    Chain = DAG.getCopyToReg(Chain, DL, V0, Val, Flag);
    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(V0, getPointerTy(DAG.getDataLayout())));
}
}
```

}

In addition to the code above, we defined the calling convention in an earlier chapter as follows:

Index/chapters/Chapter3_4/Cpu0CallingConv.td

```
def RetCC_Cpu0EABI : CallingConv<[
    // i32 are returned in registers V0, V1, A0, A1
    CCIfType<[i32], CCAssignToReg<[V0, V1, A0, A1]>>
]>;
```

This means that for the return value, we store it in registers *V0*, *V1*, *A0*, and *A1* if the size of the return value does not exceed four registers.

If it exceeds four registers, Cpu0 will store the value in memory and return a pointer to that memory in a register.

For demonstration, let's run *Chapter9_2/* with *ch9_1_struct.cpp* and explain using this example.

Index/input/ch9_1_struct.cpp

```
extern "C" int printf(const char *format, ...);

struct Date
{
    int year;
    int month;
    int day;
    int hour;
    int minute;
    int second;
};

static Date gDate = {2012, 10, 12, 1, 2, 3};

struct Time
```

(continues on next page)

(continued from previous page)

```
{  
    int hour;  
    int minute;  
    int second;  
};  
static Time gTime = {2, 20, 30};  
  
static Date getDate()  
{  
    return gDate;  
}  
  
static Date copyDate(Date date)  
{  
    return date;  
}  
  
static Date copyDate(Date* date)  
{  
    return *date;  
}  
  
static Time copyTime(Time time)  
{  
    return time;  
}  
  
static Time copyTime(Time* time)  
{  
    return *time;  
}  
  
int test_func_arg_struct()  
{  
    Time time1 = {1, 10, 12};  
    Date date1 = getDate();  
    Date date2 = copyDate(date1);  
    Date date3 = copyDate(&date1);  
    Time time2 = copyTime(time1);  
    Time time3 = copyTime(&time1);  
    if (!(date1.year == 2012 && date1.month == 10 && date1.day == 12 && date1.hour  
        == 1 && date1.minute == 10 && date1.second == 12))  
        return 1;  
    if (!(date2.year == 2012 && date2.month == 10 && date2.day == 12 && date2.hour  
        == 1 && date2.minute == 10 && date2.second == 12))  
        return 1;  
    if (!(time2.hour == 1 && time2.minute == 10 && time2.second == 12))  
        return 1;  
    if (!(time3.hour == 1 && time3.minute == 10 && time3.second == 12))  
        return 1;  
  
#ifdef PRINT_TEST
```

(continues on next page)

(continued from previous page)

```

printf("date1 = %d %d %d %d %d", date1.year, date1.month, date1.day,
      date1.hour, date1.minute, date1.second); // date1 = 2012 10 12 1 2 3
if (date1.year == 2012 && date1.month == 10 && date1.day == 12 && date1.hour
    == 1 && date1.minute == 2 && date1.second == 3)
    printf(", PASS\n");
else
    printf(", FAIL\n");
printf("date2 = %d %d %d %d %d", date2.year, date2.month, date2.day,
      date2.hour, date2.minute, date2.second); // date2 = 2012 10 12 1 2 3
if (date2.year == 2012 && date2.month == 10 && date2.day == 12 && date2.hour
    == 1 && date2.minute == 2 && date2.second == 3)
    printf(", PASS\n");
else
    printf(", FAIL\n");
// time2 = 1 10 12
printf("time2 = %d %d %d", time2.hour, time2.minute, time2.second);
if (time2.hour == 1 && time2.minute == 10 && time2.second == 12)
    printf(", PASS\n");
else
    printf(", FAIL\n");
// time3 = 1 10 12
printf("time3 = %d %d %d", time3.hour, time3.minute, time3.second);
if (time3.hour == 1 && time3.minute == 10 && time3.second == 12)
    printf(", PASS\n");
else
    printf(", FAIL\n");
#endif

    return 0;
}

```

```

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch9_1_struct.bc -o -
.section .mdebug.abi32
.previous
.file "ch9_1_struct.bc"
.text
.globl _Z7getDatev
.align 2
.type _Z7getDatev,@function
.ent _Z7getDatev          # @_Z7getDatev
_Z7getDatev:
.cfi_startproc
.frame $sp,0,$lr
.mask 0x00000000,0
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
    lui    $2, %got_hi(gDate)
    addu $2, $2, $gp

```

(continues on next page)

(continued from previous page)

```

        ld      $3, %got_lo(gDate) ($2)
        ld      $2, 0($sp)
ld  $4, 20($3)           // save gDate contents to 212..192($sp)
st  $4, 20($2)
ld  $4, 16($3)
st  $4, 16($2)
ld  $4, 12($3)
st  $4, 12($2)
ld  $4, 8($3)
st  $4, 8($2)
ld  $4, 4($3)
st  $4, 4($2)
ld  $3, 0($3)
st  $3, 0($2)
ret $lr
nop
.set macro
.set reorder
.end _Z7getDatev
$tmp0:
.size _Z7getDatev, ($tmp0)-_Z7getDatev
.cfi_endproc
...
.globl _Z20test_func_arg_structv
.align 2
.type _Z20test_func_arg_structv,@function
.ent _Z20test_func_arg_structv          # @main
_Z20test_func_arg_structv:
.cfi_startproc
.frame $sp,248,$lr
.mask 0x00004180,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
    addiu $sp, $sp, -200
    st   $lr, 196($sp)           # 4-byte Folded Spill
    st   $8, 192($sp)           # 4-byte Folded Spill
    ld   $2, %got($_ZZ20test_func_arg_structvE5time1) ($gp)
    ori  $2, $2, %lo($_ZZ20test_func_arg_structvE5time1)
    ld   $3, 8($2)
    st   $3, 184($sp)
    ld   $3, 4($2)
    st   $3, 180($sp)
    ld   $2, 0($2)
    st   $2, 176($sp)
    addiu $8, $sp, 152
    st   $8, 0($sp)
    ld   $t9, %call16(_Z7getDatev) ($gp) // copy gDate contents to date1, 176..
→152($sp)
    jalr $t9
    nop

```

(continues on next page)

(continued from previous page)

```

ld    $gp, 176($sp)
ld    $2, 172($sp)
st    $2, 124($sp)
ld    $2, 168($sp)
st    $2, 120($sp)
ld    $2, 164($sp)
st    $2, 116($sp)
ld    $2, 160($sp)
st    $2, 112($sp)
ld    $2, 156($sp)
st    $2, 108($sp)
ld    $2, 152($sp)
st    $2, 104($sp)
...

```

The *ch9_1_constructor.cpp* includes an implementation of the C++ class *Date*.

This can also be translated by the Cpu0 backend, since the frontend (Clang, in this case) translates C++ classes into equivalent C language constructs.

If you comment out the *if hasStructRetAttr()* part in both of the functions mentioned above, the output Cpu0 code for *ch9_1_struct.cpp* will use register *\$3* instead of *\$2* as the return register, as shown below:

```

.text
.section .mdebug.abiS32
.previous
.file "ch9_1_struct.bc"
.globl _Z7getDatev
.align 2
.type _Z7getDatev,@function
.ent _Z7getDatev      # @_Z7getDatev

_Z7getDatev:
.frame $fp,0,$lr
.mask 0x00000000,0
.set noreorder
.cupload $t9
.set nomacro

# BB#0:
    lui    $2, %got_hi(gDate)
    addu $2, $2, $gp
    ld    $2, %got_lo(gDate) ($2)
    ld    $3, 0($sp)
    ld    $4, 20($2)
    st    $4, 20($3)
    ld    $4, 16($2)
    st    $4, 16($3)
    ld    $4, 12($2)
    st    $4, 12($3)
    ld    $4, 8($2)
    st    $4, 8($3)
    ld    $4, 4($2)
    st    $4, 4($3)
    ld    $2, 0($2)

```

(continues on next page)

(continued from previous page)

```

st    $2, 0($3)
ret   $lr
nop
...

```

According to the MIPS ABI, the address for returning a struct variable must be placed in register \$2.

9.4.2 Byval Struct Type

The following code in *Chapter9_1*/ and *Chapter9_2*/ supports the *byval* structure type in function calls.

Ibdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```

void Cpu0TargetLowering::
copyByValRegs(SDValue Chain, const SDLoc &DL, std::vector<SDValue> &OutChains,
              SelectionDAG &DAG, const ISD::ArgFlagsTy &Flags,
              SmallVectorImpl<SDValue> &InVals, const Argument *FuncArg,
              const Cpu0CC &CC, const ByValArgInfo &ByVal) const {
    MachineFunction &MF = DAG.getMachineFunction();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    unsigned RegAreaSize = ByVal.NumRegs * CC.regSize();
    unsigned FrameObjSize = std::max(Flags.getByValSize(), RegAreaSize);
    int FrameObjOffset;

    const ArrayRef<MCPhysReg> ByValArgRegs = CC.intArgRegs();

    if (RegAreaSize)
        FrameObjOffset = (int)CC.reservedArgArea() -
            (int)((CC.numIntArgRegs() - ByVal.FirstIdx) * CC.regSize());
    else
        FrameObjOffset = ByVal.Address;

    // Create frame object.
    EVT PtrTy = getPointerTy(DAG.getDataLayout());
    int FI = MFI.CreateFixedObject(FrameObjSize, FrameObjOffset, true);
    SDValue FIN = DAG.getFrameIndex(FI, PtrTy);
    InVals.push_back(FIN);

    if (!ByVal.NumRegs)
        return;

    // Copy arg registers.
    MVT RegTy = MVT::getIntegerVT(CC.regSize() * 8);
    const TargetRegisterClass *RC = getRegClassFor(RegTy);

    for (unsigned I = 0; I < ByVal.NumRegs; ++I) {
        unsigned ArgReg = ByValArgRegs[ByVal.FirstIdx + I];
        unsigned VReg = addLiveIn(MF, ArgReg, RC);
        unsigned Offset = I * CC.regSize();
        SDValue StorePtr = DAG.getNode(ISD::ADD, DL, PtrTy, FIN,
                                      DAG.getConstant(Offset, DL, PtrTy));
        SDValue Store = DAG.getStore(Chain, DL, DAG.getRegister(VReg, RegTy),
...

```

(continues on next page)

(continued from previous page)

```

        StorePtr, MachinePointerInfo(FuncArg, Offset));
    OutChains.push_back(Store);
}
}

```

```

/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {

```

```

for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {

```

```

if (Flags.isByVal()) {
    assert(Flags.getByValSize() &&
            "ByVal args of size 0 should have been ignored by front-end.");
    assert(ByValArg != Cpu0CCInfo.byval_end());
    copyByValRegs(Chain, DL, OutChains, DAG, Flags, InVals, &*FuncArg,
                  Cpu0CCInfo, *ByValArg);
    ++ByValArg;
    continue;
}

```

```

...
.
```

```

for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    // The cpu0 ABIs for returning structs by value requires that we copy
    // the sret argument into $v0 for the return. Save the argument into
    // a virtual register so that we can access it from the return points.
    if (Ins[i].Flags.isSRet()) {
        unsigned Reg = Cpu0FI->getSRetReturnReg();
        if (!Reg) {
            Reg = MF.getRegInfo().createVirtualRegister(
                getRegClassFor(MVT::i32));
            Cpu0FI->setSRetReturnReg(Reg);
        }
        SDValue Copy = DAG.getCopyToReg(DAG.getEntryNode(), DL, Reg, InVals[i]);
        Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, Copy, Chain);
        break;
    }
}

```

```

...
}
```

[Index](#)/[chapters](#)/[Chapter9_2](#)/[Cpu0ISelLowering.cpp](#)

```
// Copy byVal arg to registers and stack.
void Cpu0TargetLowering::
passByValArg(SDValue Chain, const SDLoc &DL,
             std::deque<std::pair<unsigned, SDValue>> &RegsToPass,
             SmallVectorImpl<SDValue> &MemOpChains, SDValue StackPtr,
             MachineFrameInfo &MFI, SelectionDAG &DAG, SDValue Arg,
             const Cpu0CC &CC, const ByValArgInfo &ByVal,
             const ISD::ArgFlagsTy &Flags, bool isLittle) const {
    unsigned ByValSizeInBytes = Flags.getByValSize();
    unsigned OffsetInBytes = 0; // From beginning of struct
    unsigned RegSizeInBytes = CC.regSize();
    unsigned Alignment = std::min((unsigned)Flags.getNonZeroByValAlign().value(),  

        RegSizeInBytes);
    EVT PtrTy = getPointerTy(DAG.getDataLayout()),
    RegTy = MVT::getIntegerVT(RegSizeInBytes * 8);

    if (ByVal.NumRegs) {
        const ArrayRef<MCPhysReg> ArgRegs = CC.intArgRegs();
        bool LeftoverBytes = (ByVal.NumRegs * RegSizeInBytes > ByValSizeInBytes);
        unsigned I = 0;

        // Copy words to registers.
        for (; I < ByVal.NumRegs - LeftoverBytes;
              ++I, OffsetInBytes += RegSizeInBytes) {
            SDValue LoadPtr = DAG.getNode(ISD::ADD, DL, PtrTy, Arg,
                                         DAG.getConstant(OffsetInBytes, DL, PtrTy));
            SDValue LoadVal = DAG.getLoad(RegTy, DL, Chain, LoadPtr,
                                         MachinePointerInfo());
            MemOpChains.push_back(LoadVal.getValue(1));
            unsigned ArgReg = ArgRegs[ByVal.FirstIdx + I];
            RegsToPass.push_back(std::make_pair(ArgReg, LoadVal));
        }

        // Return if the struct has been fully copied.
        if (ByValSizeInBytes == OffsetInBytes)
            return;

        // Copy the remainder of the byval argument with sub-word loads and shifts.
        if (LeftoverBytes) {
            assert((ByValSizeInBytes > OffsetInBytes) &&
                     (ByValSizeInBytes < OffsetInBytes + RegSizeInBytes) &&
                     "Size of the remainder should be smaller than RegSizeInBytes.");
            SDValue Val;

            for (unsigned LoadSizeInBytes = RegSizeInBytes / 2, TotalBytesLoaded = 0;
                  OffsetInBytes < ByValSizeInBytes; LoadSizeInBytes /= 2) {
                unsigned RemainingSizeInBytes = ByValSizeInBytes - OffsetInBytes;

                if (RemainingSizeInBytes < LoadSizeInBytes)
                    continue;
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

// Load subword.
SDValue LoadPtr = DAG.getNode(ISD::ADD, DL, PtrTy, Arg,
                               DAG.getConstant(OffsetInBytes, DL, PtrTy));
SDValue LoadVal = DAG.getExtLoad(
    ISD::ZEXTLOAD, DL, RegTy, Chain, LoadPtr, MachinePointerInfo(),
    MVT::getIntegerVT(LoadSizeInBytes * 8), Alignment);
MemOpChains.push_back(LoadVal.getValue(1));

// Shift the loaded value.
unsigned Shamt;

if (isLittle)
    Shamt = TotalBytesLoaded * 8;
else
    Shamt = (RegSizeInBytes - (TotalBytesLoaded + LoadSizeInBytes)) * 8;

SDValue Shift = DAG.getNode(ISD::SHL, DL, RegTy, LoadVal,
                            DAG.getConstant(Shamt, DL, MVT::i32));

if (Val.getNode())
    Val = DAG.getNode(ISD::OR, DL, RegTy, Val, Shift);
else
    Val = Shift;

OffsetInBytes += LoadSizeInBytes;
TotalBytesLoaded += LoadSizeInBytes;
Alignment = std::min(Alignment, LoadSizeInBytes);
}

unsigned ArgReg = ArgRegs[ByVal.FirstIdx + I];
RegsToPass.push_back(std::make_pair(ArgReg, Val));
return;
}
}

// Copy remainder of byval arg to it with memcpy.
unsigned MemCpySize = ByValSizeInBytes - OffsetInBytes;
SDValue Src = DAG.getNode(ISD::ADD, DL, PtrTy, Arg,
                           DAG.getConstant(OffsetInBytes, DL, PtrTy));
SDValue Dst = DAG.getNode(ISD::ADD, DL, PtrTy, StackPtr,
                           DAG.getIntPtrConstant(ByVal.Address, DL));
Chain = DAG.getMemcpy(Chain, DL, Dst, Src,
                      DAG.getConstant(MemCpySize, DL, PtrTy),
                      Align(Alignment), /*isVolatile=*/false, /*AlwaysInline=*/
→false,
                      /*isTailCall=*/false,
                      MachinePointerInfo(), MachinePointerInfo());
MemOpChains.push_back(Chain);
}

```

/// LowerCall - functions arguments are copied **from virtual** regs to
 /// (physical regs)/(stack frame), CALLSEQ_START **and** CALLSEQ_END are emitted.

(continues on next page)

(continued from previous page)

```
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {

    // Walk the register/memloc assignments, inserting copies/loads.
    for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {

        if (Flags.isByVal()) {
            assert(Flags.getByValSize() &&
                   "ByVal args of size 0 should have been ignored by front-end.");
            assert(ByValArg != Cpu0CCInfo.byval_end());
            assert(!IsTailCall &&
                   "Do not tail-call optimize if there is a byval argument.");
            passByValArg(Chain, DL, RegsToPass, MemOpChains, StackPtr, MFI, DAG, Arg,
                         Cpu0CCInfo, *ByValArg, Flags, Subtarget.isLittle());
            ++ByValArg;
            continue;
        }

        ...
    }
}
```

In `LowerCall()`, `Flags.isByVal()` will be *true* if the function call in the caller contains a **byval** struct type, as shown below:

Ibdex/input/tailcall.ll

```
define internal fastcc i32 @caller9_1() nounwind noinline {
entry:
...
%call = tail call i32 @callee9(%struct.S* byval @gs1) nounwind
ret i32 %call
}
```

In `LowerFormalArguments()`, `Flags.isByVal()` will be *true* when it encounters a **byval** parameter in the callee function, as shown below:

Ibdex/input/tailcall.ll

```
define i32 @caller12(%struct.S* nocapture byval %a0) nounwind {
entry:
...
}
```

At this point, I don't know how to make Clang generate *byval* IR using the C language.

9.5 Function Call Optimization

9.5.1 Tail Call Optimization

Tail call optimization is applied in certain function call situations. In some cases, the caller and callee can share the same memory stack.

When applied to recursive function calls, this optimization often reduces the stack space requirement from linear, or $O(n)$, to constant, or $O(1)$ ⁵.

LLVM IR supports *tailcall* as described here⁶.

The *tailcall* instructions appearing in *Cpu0ISelLowering.cpp* and *Cpu0InstrInfo.td* are used to implement tail call optimization.

Ibdex/input/ch9_2_tailcall.cpp

```
int factorial(int x)
{
    if (x > 0)
        return x*factorial(x-1);
    else
        return 1;
}

int test_tailcall(int a)
{
    return factorial(a);
}
```

Run *Chapter9_2*/with *ch9_2_tailcall.cpp* to get the following result.

```
JonathantekiiMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu -c
ch9_2_tailcall.cpp -emit-llvm -o ch9_2_tailcall.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
 llvm-dis ch9_2_tailcall.bc -o -
...
; Function Attrs: nounwind readnone
define i32 @_Z9factoriali(i32 %x) #0 {
    %1 = icmp sgt i32 %x, 0
    br i1 %1, label %tailrecurse, label %tailrecurse._crit_edge

tailrecurse:                                ; preds = %tailrecurse, %0
    %x.tr2 = phi i32 [ %2, %tailrecurse ], [ %x, %0 ]
    %accumulator.tr1 = phi i32 [ %3, %tailrecurse ], [ 1, %0 ]
    %2 = add nsw i32 %x.tr2, -1
    %3 = mul nsw i32 %x.tr2, %accumulator.tr1
    %4 = icmp sgt i32 %2, 0
    br i1 %4, label %tailrecurse, label %tailrecurse._crit_edge
```

(continues on next page)

⁵ http://en.wikipedia.org/wiki/Tail_call

⁶ <http://llvm.org/docs/CodeGenerator.html#tail-call-optimization>

(continued from previous page)

```

tailrecurse._crit_edge:                                ; preds = %tailrecurse, %0
    %accumulator.tr.lcssa = phi i32 [ 1, %0 ], [ %3, %tailrecurse ]
    ret i32 %accumulator.tr.lcssa
}

; Function Attrs: nounwind readnone
define i32 @_Z13test_tailcalli(i32 %a) #0 {
    %1 = tail call i32 @_Z9factoriali(i32 %a)
    ret i32 %1
}
...
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llc -march=cpu0 -mcpu=cpu032II -relocation-model=static -filetype=asm
-enable-cpu0-tail-calls ch9_2_tailcall.bc -stats -o -
    .text
    .section .mdebug.abi32
    .previous
    .file "ch9_2_tailcall.bc"
    .globl      _Z9factoriali
    .align      2
    .type      _Z9factoriali,@function
    .ent      _Z9factoriali          # @_Z9factoriali
_Z9factoriali:
    .frame      $sp,0,$lr
    .mask       0x00000000,0
    .set      noreorder
    .set      nomacro
# BB#0:
    addiu $2, $zero, 1
    slt   $3, $4, $2
    bne   $3, $zero, $BB0_2
    nop
$BB0_1:                                     # %tailrecurse
                                                # =>This Inner Loop Header: Depth=1
    mul   $2, $4, $2
    addiu $4, $4, -1
    addiu $3, $zero, 0
    slt   $3, $3, $4
    bne   $3, $zero, $BB0_1
    nop
$BB0_2:                                     # %tailrecurse._crit_edge
    ret   $lr
    nop
    .set  macro
    .set  reorder
    .end  _Z9factoriali
$tmp0:
    .size _Z9factoriali, ($tmp0)-_Z9factoriali

    .globl      _Z13test_tailcalli
    .align      2
    .type      _Z13test_tailcalli,@function

```

(continues on next page)

(continued from previous page)

```

.ent _Z13test_tailcalli      # @_Z13test_tailcalli
_Z13test_tailcalli:
    .frame      $sp,0,$lr
    .mask       0x00000000,0
    .set  noreorder
    .set  nomacro
# BB#0:
    jmp  _Z9factoriali
    nop
    .set  macro
    .set  reorder
    .end  _Z13test_tailcalli
$tmp1:
    .size _Z13test_tailcalli, ($tmp1)-_Z13test_tailcalli

=====
... Statistics Collected ...
=====

...
1 cpu0-lower      - Number of tail calls
...

```

The tail call optimization shares the caller's and callee's stack, and it is applied in *cpu032II* only for this example (it uses *jmp _Z9factoriali* instead of *jsub _Z9factoriali*).

However, *cpu032I* (which passes all arguments on the stack) does not satisfy the condition *NextStackOffset <= FI.getIncomingArgSize()* in *isEligibleForTailCallOptimization()*, and thus returns *false* for the function, as shown below:

Ibdex/chapters/Chapter9_2/Cpu0SEISelLowering.cpp

```

bool Cpu0SETargetLowering::
isEligibleForTailCallOptimization(const Cpu0CC &Cpu0CCInfo,
                                  unsigned NextStackOffset,
                                  const Cpu0FunctionInfo& FI) const {
    if (!EnableCpu0TailCalls)
        return false;

    // Return false if either the callee or caller has a byval argument.
    if (Cpu0CCInfo.hasByValArg() || FI.hasByvalArg())
        return false;

    // Return true if the callee's argument area is no larger than the
    // caller's.
    return NextStackOffset <= FI.getIncomingArgSize();
}

```

Ibdex/chapters/Chapter9_2/Cpu0ISelLowering.cpp

```

/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {

    // Check if it's really possible to do a tail call.
    if (IsTailCall)
        IsTailCall =
            isEligibleForTailCallOptimization(Cpu0CCInfo, NextStackOffset,
                                              *MF.getInfo<Cpu0FunctionInfo>());

    if (!IsTailCall && CLI.CB && CLI.CB->isMustTailCall())
        report_fatal_error("failed to perform tail call elimination on a call "
                           "site marked musttail");

    if (IsTailCall)
        ++NumTailCalls;

    if (!IsTailCall)
        Chain = DAG.getCALLSEQ_START(Chain, NextStackOffset, 0, DL);

    if (IsTailCall)
        return DAG.getNode(Cpu0ISD::TailCall, DL, MVT::Other, Ops);

    ...
}

```

Since tail call optimization translates the call into a *jmp* instruction directly instead of *jsub*, the *callseq_start*, *callseq_end*, and the DAG nodes created in *LowerCallResult()* and *LowerReturn()* are unnecessary. It creates DAGs for *ch9_2_tailcall.cpp* as shown in Fig. 9.7.

Finally, the DAGs translation of the tail call is listed in the following table.

Table 9.2: the DAGs translation of tail call

Stage	DAG	Function
Backend lowering	Cpu0ISD::TailCall	LowerCall()
Instruction selection	TAILCALL	note 1
Instruction Print	JMP	note 2

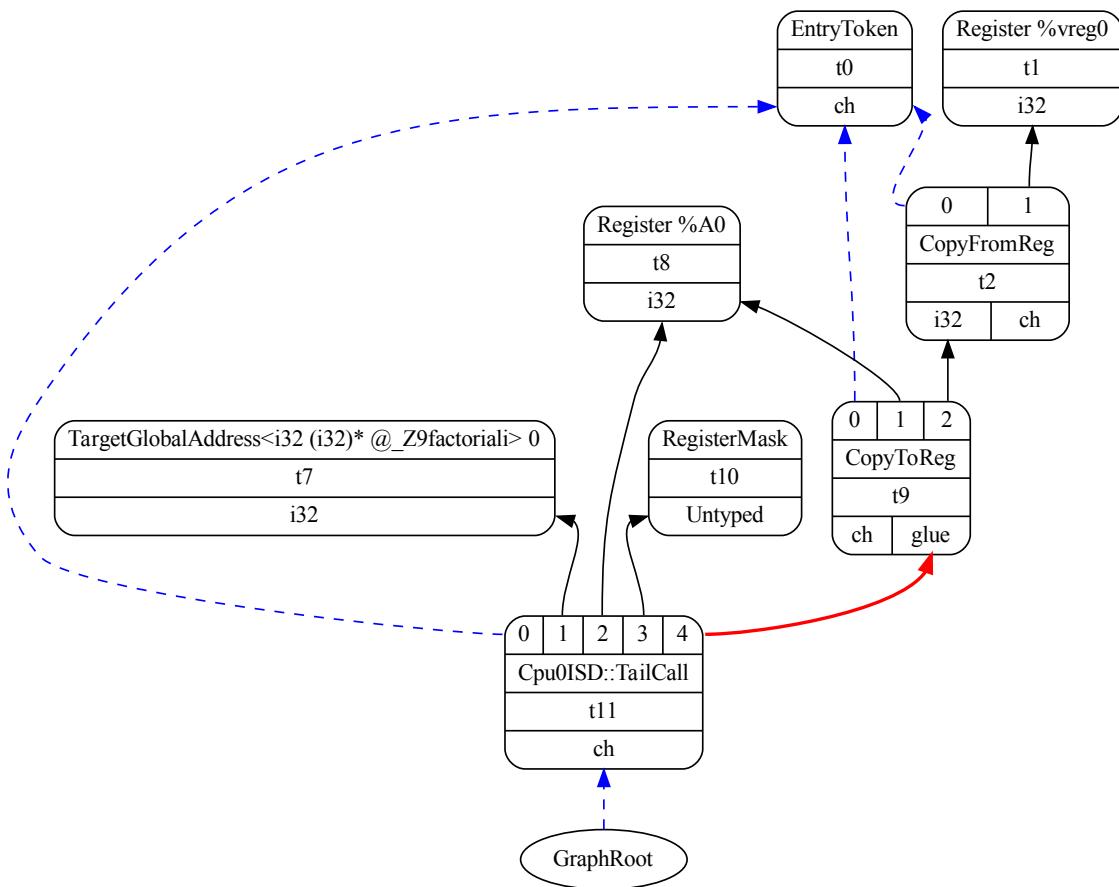
note 1: by Cpu0InstrInfo.td as follows,

Ibdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```

// Tail call
def Cpu0TailCall : SDNode<"Cpu0ISD::TailCall", SDT_Cpu0JmpLink,
                    [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

```


 Fig. 9.7: Outgoing arguments DAGs created for `ch9_2_tailcall.cpp`

```
def : Pat<(Cpu0TailCall (iPTR tglobaladdr:$dst)),  
        (TAILCALL tglobaladdr:$dst)>;  
def : Pat<(Cpu0TailCall (iPTR texternalsym:$dst)),  
        (TAILCALL texternalsym:$dst)>;
```

note 2: by Cpu0InstrInfo.td and emitPseudoExpansionLowering() of Cpu0AsmPrinter.cpp as follows,

Ibdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```
let isCall = 1, isTerminator = 1, isReturn = 1, isBarrier = 1, hasDelaySlot = 1,  
    hasExtraSrcRegAllocReq = 1, Defs = [AT] in {  
    class TailCall<Instruction JumpInst> :  
        PseudoSE<(outs), (ins calltarget:$target), [], IIBranch>,  
        PseudoInstExpansion<(JumpInst jmptarget:$target)>;  
  
    class TailCallReg<RegisterClass RO, Instruction JRInst,  
                    RegisterClass ResRO = RO> :  
        PseudoSE<(outs), (ins RO:$rs), [(Cpu0TailCall RO:$rs)], IIBranch>,  
        PseudoInstExpansion<(JRInst ResRO:$rs)>;  
}
```

```
let Predicates = [Ch9_1] in {  
def TAILCALL : TailCall<JMP>;  
def TAILCALL_R : TailCallReg<GPROut, JR>;  
}
```

Ibdex/chapters/Chapter9_1/Cpu0AsmPrinter.h

```
// tblgen'reated function.  
bool emitPseudoExpansionLowering(MCStreamer &OutStreamer,  
                                  const MachineInstr *MI);
```

Ibdex/chapters/Chapter9_1/Cpu0AsmPrinter.cpp

```
//- emitInstruction() must exists or will have run time error.  
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {  
//@EmitInstruction body {  
    if (MI->isDebugValue()) {  
        SmallString<128> Str;  
        raw_svector_ostream OS(Str);  
  
        PrintDebugValueComment(MI, OS);  
        return;  
    }  
  
    //@print out instruction:  
    // Print out both ordinary instruction and boudle instruction  
    MachineBasicBlock::const_instr_iterator I = MI->getIterator();  
    MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();  
  
    do {
```

(continues on next page)

(continued from previous page)

```

// Do any auto-generated pseudo lowerings.
if (emitPseudoExpansionLowering(*OutStreamer, &*I))
    continue;

if (I->isPseudo() && !isLongBranchPseudo(I->getOpcode()))
    llvm_unreachable("Pseudo opcode found in emitInstruction()");

MCInst TmpInst0;
// Call Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) to
// extracts MCInst from MachineInstr.
MCInstLowering.Lower(&*I, TmpInst0);
OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
} while ((++I != E) && I->isInsideBundle()); // Delay slot check
}

```

The function `emitPseudoExpansionLowering()` is generated by TableGen and is located in `Cpu0GenMCPseudoLowering.inc`.

9.5.2 Recursion optimization

As mentioned in the last section, cpu032I cannot perform tail call optimization in `ch9_2_tailcall.cpp` due to the limitation that the argument size condition is not satisfied.

However, when running with the `clang -O3` optimization option, it can achieve the same or even better performance than tail call optimization, as shown below:

```

JonathantekiiMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu -c
ch9_2_tailcall.cpp -emit-llvm -o ch9_2_tailcall.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch9_2_tailcall.bc -o -
...
; Function Attrs: nounwind readnone
define i32 @_Z9factoriali(i32 %x) #0 {
    %1 = icmp sgt i32 %x, 0
    br i1 %1, label %tailrecurse.preheader, label %tailrecurse._crit_edge

tailrecurse.preheader:                                ; preds = %0
    br label %tailrecurse

tailrecurse:                                         ; preds = %tailrecurse,
%tailrecurse.preheader
    %x.tr2 = phi i32 [ %2, %tailrecurse ], [ %x, %tailrecurse.preheader ]
    %accumulator.tr1 = phi i32 [ %3, %tailrecurse ], [ 1, %tailrecurse.preheader ]
    %2 = add nsw i32 %x.tr2, -1
    %3 = mul nsw i32 %x.tr2, %accumulator.tr1
    %4 = icmp sgt i32 %2, 0
    br i1 %4, label %tailrecurse, label %tailrecurse._crit_edge.loopexit

tailrecurse._crit_edge.loopexit:                      ; preds = %tailrecurse
    %.lcssa = phi i32 [ %3, %tailrecurse ]
    br label %tailrecurse._crit_edge

tailrecurse._crit_edge:                            ; preds = %tailrecurse._crit

```

(continues on next page)

(continued from previous page)

```

_edge.loopexit, %0
%accumulator.tr.lcssa = phi i32 [ 1, %0 ], [ %.lcssa, %tailrecurse._crit_edge
.loopexit ]
ret i32 %accumulator.tr.lcssa
}

; Function Attrs: nounwind readnone
define i32 @_Z13test_tailcalli(i32 %a) #0 {
    %1 = icmp sgt i32 %a, 0
    br i1 %1, label %tailrecurse.i.preheader, label %_Z9factoriali.exit

tailrecurse.i.preheader:                                ; preds = %0
    br label %tailrecurse.i

tailrecurse.i:                                         ; preds = %tailrecurse.i,
    %tailrecurse.i.preheader
    %x.tr2.i = phi i32 [ %2, %tailrecurse.i ], [ %a, %tailrecurse.i.preheader ]
    %accumulator.tr1.i = phi i32 [ %3, %tailrecurse.i ], [ 1, %tailrecurse.i.
    preheader ]
    %2 = add nsw i32 %x.tr2.i, -1
    %3 = mul nsw i32 %accumulator.tr1.i, %x.tr2.i
    %4 = icmp sgt i32 %2, 0
    br i1 %4, label %tailrecurse.i, label %_Z9factoriali.exit.loopexit

_Z9factoriali.exit.loopexit:                            ; preds = %tailrecurse.i
    %.lcssa = phi i32 [ %3, %tailrecurse.i ]
    br label %_Z9factoriali.exit

_Z9factoriali.exit:                                    ; preds = %_Z9factoriali.
exit.loopexit, %0
    %accumulator.tr.lcssa.i = phi i32 [ 1, %0 ], [ %.lcssa, %_Z9factoriali.
exit.loopexit ]
    ret i32 %accumulator.tr.lcssa.i
}

...
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_2_tailcall.bc -o -
    .text
    .section .mdebug.abiS32
    .previous
    .file "ch9_2_tailcall.bc"
    .globl _Z9factoriali
    .align 2
    .type _Z9factoriali,@function
    .ent _Z9factoriali          # @_Z9factoriali
_Z9factoriali:
    .frame $sp,0,$lr
    .mask 0x00000000,0
    .set noreorder
    .set nomacro
# BB#0:

```

(continues on next page)

(continued from previous page)

```

addiu $2, $zero, 1
ld    $3, 0($sp)
cmp   $sw, $3, $2
jlt   $sw, $BB0_2
nop
$BB0_1:                                # %tailrecuse
# =>This Inner Loop Header: Depth=1
mul   $2, $3, $2
addiu $3, $3, -1
addiu $4, $zero, 0
cmp   $sw, $3, $4
jgt   $sw, $BB0_1
nop
$BB0_2:                                # %tailrecuse._crit_edge
ret   $lr
nop
.set  macro
.set  reorder
.end  _Z9factoriali
$tmp0:
.size _Z9factoriali, ($tmp0)-_Z9factoriali

.globl      _Z13test_tailcalli
.align      2
.type _Z13test_tailcalli,@function
.ent  _Z13test_tailcalli      # @_Z13test_tailcalli
_Z13test_tailcalli:
.frame      $sp,0,$lr
.mask       0x00000000,0
.set  noreorder
.set  nomacro
# BB#0:
addiu $2, $zero, 1
ld    $3, 0($sp)
cmp   $sw, $3, $2
jlt   $sw, $BB1_2
nop
$BB1_1:                                # %tailrecuse.i
# =>This Inner Loop Header: Depth=1
mul   $2, $2, $3
addiu $3, $3, -1
addiu $4, $zero, 0
cmp   $sw, $3, $4
jgt   $sw, $BB1_1
nop
$BB1_2:                                # %_Z9factoriali.exit
ret   $lr
nop
.set  macro
.set  reorder
.end  _Z13test_tailcalli
$tmp1:

```

(continues on next page)

(continued from previous page)

```
.size _Z13test_tailcalli, ($tmp1)-_Z13test_tailcalli
```

According to the above LLVM IR, the `clang -O3` option replaces recursion with a loop by inlining the callee recursion function. This is a frontend optimization achieved through cross-function analysis.

Cpu0 doesn't support *fastcc*⁷, but it can pass the *fastcc* keyword in the IR. MIPS supports *fastcc* by using as many registers as possible without strictly following the ABI specification.

9.6 Other Features Supported

This section supports features for the “\$gp register caller saved register in PIC addressing mode,”“variable number of arguments,” and “dynamic stack allocation.”

Run *Chapter9_2*/with *ch9_3_vararg.cpp* to get the following error:

Ibdex/input/ch9_3_vararg.cpp

```
#include <stdarg.h>

int sum_i(int amount, ...)
{
    int i = 0;
    int val = 0;
    int sum = 0;

    va_list vl;
    va_start(vl, amount);
    for (i = 0; i < amount; i++)
    {
        val = va_arg(vl, int);
        sum += val;
    }
    va_end(vl);

    return sum;
}

long long sum_ll(long long amount, ...)
{
    long long i = 0;
    long long val = 0;
    long long sum = 0;

    va_list vl;
    va_start(vl, amount);
    for (i = 0; i < amount; i++)
    {
        val = va_arg(vl, long long);
        sum += val;
    }
}
```

(continues on next page)

⁷ <http://llvm.org/docs/LangRef.html#calling-conventions>

(continued from previous page)

```

va_end(vl);

return sum;
}

int test_va_arg()
{
    int a = sum_i(6, 0, 1, 2, 3, 4, 5);
    long long b = sum_ll(6LL, 0LL, 1LL, 2LL, 3LL, -4LL, -5LL);

    return a+(int)b; // 12
}

```

```

118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_vararg.cpp -emit-llvm -o ch9_3_vararg.bc
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_3_vararg.bc -o -
...
LLVM ERROR: Cannot select: 0x7f8b6902fd10: ch = vastart 0x7f8b6902fa10,
0x7f8b6902fb10, 0x7f8b6902fc10 [ORD=9] [ID=22]
0x7f8b6902fb10: i32 = FrameIndex<5> [ORD=7] [ID=9]
In function: _Z5sum_iiz

```

Ibdex/input/ch9_3_alloc.cpp

```

// This file needed compile without option, -target mips-unknown-linux-gnu, so
// it is verified by build-run_backend2.sh or verified in lld linker support
// (build-slinker.sh).

##include <alloca.h>
##include <stdlib.h>

int sum(int x1, int x2, int x3, int x4, int x5, int x6)
{
    int sum = x1 + x2 + x3 + x4 + x5 + x6;

    return sum;
}

int weight_sum(int x1, int x2, int x3, int x4, int x5, int x6)
{
    int *b = (int*)alloca(sizeof(int) * 1 * x1);
    int* b = (int*)__builtin_alloca(sizeof(int) * 1 * x1);
    int *a = b;
    *b = x3;

    int weight = sum(3*x1, x2, x3, x4, 2*x5, x6);

    return (weight + (*a));
}

```

(continues on next page)

(continued from previous page)

```
int test_alloc()
{
    int a = weight_sum(1, 2, 3, 4, 5, 6); // 31

    return a;
}
```

Run *Chapter9_2* with *ch9_3_alloc.cpp* to get the following error.

```
118-165-72-242:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_alloc.cpp -emit-llvm -o ch9_3_alloc.bc
118-165-72-242:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_3_alloc.bc -o -
...
LLVM ERROR: Cannot select: 0x7ffd8b02ff10: i32, ch = dynamic_stackalloc
0x7ffd8b02f910:1, 0x7ffd8b02fe10, 0x7ffd8b02c010 [ORD=12] [ID=48]
0x7ffd8b02fe10: i32 = and 0x7ffd8b02fc10, 0x7ffd8b02fd10 [ORD=12] [ID=47]
0x7ffd8b02fc10: i32 = add 0x7ffd8b02fa10, 0x7ffd8b02fb10 [ORD=12] [ID=46]
0x7ffd8b02fa10: i32 = shl 0x7ffd8b02f910, 0x7ffd8b02f510 [ORD=12] [ID=45]
0x7ffd8b02f910: i32, ch = load 0x7ffd8b02ee10, 0x7ffd8b02e310,
0x7ffd8b02b310<LD4[%1]> [ID=44]
0x7ffd8b02e310: i32 = FrameIndex<1> [ORD=3] [ID=10]
0x7ffd8b02b310: i32 = undef [ORD=1] [ID=2]
0x7ffd8b02f510: i32 = Constant<2> [ID=25]
0x7ffd8b02fb10: i32 = Constant<7> [ORD=12] [ID=16]
0x7ffd8b02fd10: i32 = Constant<-8> [ORD=12] [ID=17]
0x7ffd8b02c010: i32 = Constant<0> [ORD=12] [ID=8]
In function: _Z5sum_iiiiiii
```

9.6.1 Global Variables Accessing In PIC Addressing Mode

In order to support Global Variables accessing in PIC mode, The \$gp Register Caller Saved Register in PIC Addressing Mode have to be solved.

According to the original Cpu0 website information, it only supports “**jsub**” for 24-bit address range access. We added “**jalr**” to Cpu0 and expanded it to 32-bit addressing. We made this change for two reasons:

1. Cpu0 can be expanded to 32-bit address space by simply adding this instruction.
2. Cpu0 and this book are designed as a tutorial for better understanding.

We reserve “**jalr**” for PIC mode, which is used for dynamic linking functions, to demonstrate:

1. How the caller handles the caller-saved register **\$gp** when calling a function.
2. How code in the shared library function uses **\$gp** to access the global variable address.
3. Why using **jalr** for dynamic linking functions is easier to implement and faster. As we discussed in the [section PIC Mode in Chapter Global Variables](#) this solution is popular in real applications and deserves to be incorporated into the official Cpu0 design in compiler books.

In the chapter on “Global Variables,” we mentioned two link types: static link and dynamic link. The option **-relocation-model=static** is for static link functions, while **-relocation-model=pic** is for dynamic link functions. An example of a dynamic link function is calling functions from a shared library.

Shared libraries consist of many dynamic link functions that are typically loaded at runtime. Since shared libraries can be loaded at different memory addresses, the address of a global variable cannot be determined at link time. However, the distance between the global variable address and the start address of the shared library function can be calculated once it has been loaded.

Let's run *Chapter9_3/* with *ch9_gprestore.cpp* to get the following result. We will add comments in the result for explanation.

Ibdex/input/ch9_gprestore.cpp

```
extern int sum_i(int x1);

int call_sum_i() {
    int a = sum_i(1);
    a += sum_i(2);
    return a;
}
```

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II-cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_gprestore.bc -o -
...
.cupload $t9
.set nomacro
# BB#0:                                # %entry
addiu $sp, $sp, -24
$tmp0:
.cfi_def_cfa_offset 24
st $lr, 12($sp)           # 4-byte Folded Spill
st $fp, 16($sp)           # 4-byte Folded Spill
$tmp1:
.cfi_offset 14, -4
$tmp2:
.cfi_offset 12, -8
.cprestore 8 // save $gp to 8($sp)
ld $t9, %call16(_Z5sum_ii)($gp)
addiu $4, $zero, 1
jalr $t9
nop
ld $gp, 8($sp) // restore $gp from 8($sp)
add $8, $zero, $2
ld $t9, %call16(_Z5sum_ii)($gp)
addiu $4, $zero, 2
jalr $t9
nop
ld $gp, 8($sp) // restore $gp from 8($sp)
addu $2, $2, $8
ld $8, 8($sp)           # 4-byte Folded Reload
ld $lr, 12($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
```

As mentioned in the code comment, “**.cprestore 8**” is a pseudo instruction for saving **\$gp** to **8(\$sp)**, while the instruc-

tion “`ld $gp, 8($sp)`” restores the `$gp`. Refer to Table 8-1 of the “MIPSpro TM Assembly Language Programmer’s Guide”^{Page 401, 2} for more details.

In other words, `$gp` is a caller-saved register, so the `main()` function needs to save and restore `$gp` before and after calling the shared library `_Z5sum_ii()` function.

In LLVM MIPS 3.5, the `.cprestore` instruction was removed in PIC mode, meaning `$gp` is no longer treated as a caller-saved register in PIC. However, it is still present in Cpu0, and this feature can be removed by not defining it in `Cpu0Config.h`.

The `#ifdef ENABLE_GPRESTORE` part of the code in Cpu0 can be removed, but it comes with the cost of reserving the `$gp` register as a specific register that cannot be allocated for program variables in PIC mode. As explained in earlier chapters on “Global Variables,” PIC is not a critical function, and its performance advantage can be considered negligible in dynamic linking. Therefore, we keep this feature in Cpu0.

Reserving `$gp` as a specific register in PIC mode will save a lot of code during programming. When reserving `$gp`, the `.cprestore` can be disabled using the option “`-cpu0-reserve-gp`”.

The `.cupload` instruction is still needed even when reserving `$gp` (since programmers may implement boot code functions with a mix of C and assembly). In this case, the programmer can set the `$gp` value through `.cupload`.

If enabling `-cpu0-no-cupload`, and undefining `ENABLE_GPRESTORE` or enabling `-cpu0-reserve-gp`, the `.cupload` and `$gp` save/restore instructions will not be issued, as shown in the following.

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II-cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_gprestore.bc -cpu0-no-cupload
-cpu0-reserve-gp -o -
...
# BB#0:
addiu $sp, $sp, -24
$tmp0:
.cfi_def_cfa_offset 24
st $lr, 20($sp)          # 4-byte Folded Spill
st $fp, 16($sp)          # 4-byte Folded Spill
$tmp1:
.cfi_offset 14, -4
$tmp2:
.cfi_offset 12, -8
move $fp, $sp
$tmp3:
.cfi_def_cfa_register 12
ld $t9, %call16(_Z5sum_ii)($gp)
addiu $4, $zero, 1
jalr $t9
nop
st $2, 12($fp)
addiu $4, $zero, 2
ld $t9, %call16(_Z5sum_ii)($gp)
jalr $t9
nop
ld $3, 12($fp)
addu $2, $3, $2
st $2, 12($fp)
move $sp, $fp
ld $fp, 16($sp)          # 4-byte Folded Reload
ld $lr, 20($sp)          # 4-byte Folded Reload
```

(continues on next page)

(continued from previous page)

```

addiu $sp, $sp, 24
ret $lr
nop

```

LLVM Mips 3.1 emits the directives `.cupload` and `.cprestore`, and Cpu0 inherits this behavior from that version. However, newer versions of LLVM Mips replace `.cupload` with actual instructions and remove `.cprestore` entirely. In these versions, the `$gp` register is treated as a reserved register in PIC (position-independent code) mode.

According to the MIPS assembly documentation I referenced, `$gp` is considered a “caller-saved register.” Cpu0 follows this convention and provides an option to reserve the `$gp` register accordingly.

```

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=mips -relocation-model=pic -filetype=asm ch9_gprestore.bc
-o -
...
# BB#0:                                # %entry
    lui $2, %hi(_gp_disp)
    ori $2, $2, %lo(_gp_disp)
    addiu $sp, $sp, -32
$tmp0:
    .cfi_def_cfa_offset 32
    sw $ra, 28($sp)          # 4-byte Folded Spill
    sw $fp, 24($sp)          # 4-byte Folded Spill
    sw $16, 20($sp)          # 4-byte Folded Spill
$tmp1:
    .cfi_offset 31, -4
$tmp2:
    .cfi_offset 30, -8
$tmp3:
    .cfi_offset 16, -12
    move $fp, $sp
$tmp4:
    .cfi_def_cfa_register 30
    addu $16, $2, $25
    lw $25, %call16(_Z5sum_ii)($16)
    addiu $4, $zero, 1
    jalr $25
    move $gp, $16
    sw $2, 16($fp)
    lw $25, %call16(_Z5sum_ii)($16)
    jalr $25
    addiu $4, $zero, 2
    lw $1, 16($fp)
    addu $2, $1, $2
    sw $2, 16($fp)
    move $sp, $fp
    lw $16, 20($sp)          # 4-byte Folded Reload
    lw $fp, 24($sp)          # 4-byte Folded Reload
    lw $ra, 28($sp)          # 4-byte Folded Reload
    jr $ra
    addiu $sp, $sp, 32

```

The following code, added in Chapter9_3/, emits `.cprestore` or the corresponding machine instructions before the first

PIC function call.

Ibdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {
```

```
#ifdef ENABLE_GPRESTORE
if (!Cpu0ReserveGP) {
    // If this is the first call, create a stack frame object that points to
    // a location to which .cprestore saves $gp.
    if (IsPIC && Cpu0FI->globalBaseRegFixed() && !Cpu0FI->getGPF() )
        Cpu0FI->setGPF(MFI.CreateFixedObject(4, 0, true));
    if (Cpu0FI->needGPSaveRestore())
        MFI.setObjectOffset(Cpu0FI->getGPF(), NextStackOffset);
}
#endif
```

```
...
```

Ibdex/chapters/Chapter9_3/Cpu0MachineFunction.h

```
#ifdef ENABLE_GPRESTORE
bool needGPSaveRestore() const { return getGPF(); }
#endif
```

Ibdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {

#ifdef ENABLE_GPRESTORE
    // Restore GP from the saved stack location
    if (Cpu0FI->needGPSaveRestore()) {
        unsigned Offset = MFI.getObjectOffset(Cpu0FI->getGPF());
        BuildMI(MBB, MBBI, dl, TII.get(Cpu0::CPRESTORE)).addImm(Offset)
            .addReg(Cpu0::GP);
    }
#endif

}
```

Ibdex/chapters/Chapter9_3/Cpu0RegisterInfo.cpp

```
//- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
```

(continues on next page)

(continued from previous page)

```
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                     unsigned FIOperandNum, RegScavenger *RS) const {
```

```
#ifdef ENABLE_GPRESTORE //2
    if (CpuOFI->isOutArgFI(FrameIndex) || CpuOFI->isGPFI(FrameIndex) ||
        CpuOFI->isDynAllocFI(FrameIndex))
        Offset = spOffset;
    else
#endif
...
}
```

Ibdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
// When handling PIC code the assembler needs .cupload and .cprestore
// directives. If the real instructions corresponding these directives
// are used, we have the same behavior, but get also a bunch of warnings
// from the assembler.
let hasSideEffects = 0 in
def CPRESTORE : Cpu0Pseudo<(outs), (ins i32imm:$loc, CPUREgs:$gp),
    ".cprestore\t$loc", []>;
```

Ibdex/chapters/Chapter9_3/Cpu0AsmPrinter.cpp

```
#ifdef ENABLE_GPRESTORE
void Cpu0AsmPrinter::EmitInstrWithMacroNoAT(const MachineInstr *MI) {
    MCInst TmpInst;

    MCInstLowering.Lower(MI, TmpInst);
    OutStreamer->emitRawText(StringRef("\t.set\tmacro"));
    if (CpuOFI->get.EmitNOAT())
        OutStreamer->emitRawText(StringRef("\t.set\tat"));
    OutStreamer->emitInstruction(TmpInst, getSubtargetInfo());
    if (CpuOFI->get.EmitNOAT())
        OutStreamer->emitRawText(StringRef("\t.set\tnoat"));
    OutStreamer->emitRawText(StringRef("\t.set\tnomacro"));
}
#endif
```

```
#ifdef ENABLE_GPRESTORE
void Cpu0AsmPrinter::emitPseudoCPRestore(MCStreamer &OutStreamer,
                                         const MachineInstr *MI) {
    SmallVector<MCInst, 4> MCInsts;
    const MachineOperand &MO = MI->getOperand(0);
    assert(MO.isImm() && "CPRESTORE's operand must be an immediate.");
```

(continues on next page)

(continued from previous page)

```

int64_t Offset = MO.getImm();

if (OutStreamer.hasRawTextSupport()) {
    // output assembly
    if (!isInt<16>(Offset)) {
        EmitInstrWithMacroNoAT(MI);
        return;
    }
    MCInst TmpInst0;
    MCInstLowering.Lower(MI, TmpInst0);
    OutStreamer.emitInstruction(TmpInst0, getSubtargetInfo());
} else {
    // output elf
    MCInstLowering.LowerCPRESTORE(Offset, MCInsts);

    for (SmallVector<MCInst, 4>::iterator I = MCInsts.begin();
         I != MCInsts.end(); ++I)
        OutStreamer.emitInstruction(*I, getSubtargetInfo());

    return;
}
#endif

```

```

// - emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {

```

```

#ifdef ENABLE_GPRESTORE
    if (I->getOpcode() == Cpu0::CPRESTORE) {
        emitPseudoCPRestore(*OutStreamer, &*I);
        continue;
    }
#endif

```

```

    ...
}
```

Ibdex/chapters/Chapter9_3/Cpu0MCInstLower.h

```

#ifdef ENABLE_GPRESTORE
    void LowerCPRESTORE(int64_t Offset, SmallVector<MCInst, 4>& MCInsts);
#endif

```

Ibdex/chapters/Chapter9_3/Cpu0MCInstLower.cpp

```

#ifdef ENABLE_GPRESTORE
// Lower ".cprestore offset" to "st $gp, offset($sp)".
void Cpu0MCInstLower::LowerCPRESTORE(int64_t Offset,
                                      SmallVector<MCInst, 4>& MCInsts) {
    assert(isInt<32>(Offset) && (Offset >= 0) &&

```

(continues on next page)

(continued from previous page)

```

"Imm operand of .cprestore must be a non-negative 32-bit value.");

MCOperand SPReg = MCOperand::createReg(Cpu0::SP), BaseReg = SPReg;
MCOperand GPReg = MCOperand::createReg(Cpu0::GP);
MCOperand ZEROReg = MCOperand::createReg(Cpu0::ZERO);

if (!isInt<16>(Offset)) {
    unsigned Hi = ((Offset + 0x8000) >> 16) & 0xffff;
    Offset &= 0xffff;
    MCOperand ATReg = MCOperand::createReg(Cpu0::AT);
    BaseReg = ATReg;

    // lui    at,hi
    // add    at,at,sp
    MCInsts.resize(2);
    CreateMCInst(MCInsts[0], Cpu0::LUI, ATReg, ZEROReg, MCOperand::createImm(Hi));
    CreateMCInst(MCInsts[1], Cpu0::ADD, ATReg, ATReg, SPReg);
}

MCInst St;
CreateMCInst(St, Cpu0::ST, GPReg, BaseReg, MCOperand::createImm(Offset));
MCInsts.push_back(St);
}
#endif

```

The added code in `Cpu0AsmPrinter.cpp`, as shown above, will call `LowerCPRESTORE()` when the user runs the program with `llc -filetype=obj`.

The added code in `Cpu0MCInstLower.cpp`, as shown above, handles the machine instructions for `.cprestore`.

```

118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch9_1.bc -o ch9_1.cpu0.o
118-165-76-131:input Jonathan$ hexdump ch9_1.cpu0.o
...
// .cprestore machine instruction " 01 ad 00 18"
00000d0 01 ad 00 18 09 20 00 00 01 2d 00 40 09 20 00 06
...

118-165-67-25:input Jonathan$ cat ch9_1.cpu0.s
...
.ent _Z5sum_iiiiiii      # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
...
.cupload $t9 // assign $gp = $t9 by loader when loader load re-entry function
               // (shared library) of _Z5sum_iiiiiii
.set nomacro
# BB#0:
...
.ent main             # @main
...
.cprestore 24 // save $gp to 24($sp)
...

```

Running `llc -static` will emit the `jsub` instruction instead of `jalr`, as shown below:

```
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=static -filetype=
asm ch9_1.bc -o ch9_1.cpu0.s
118-165-76-131:input Jonathan$ cat ch9_1.cpu0.s
...
jsub _Z5sum_iiiiii
...
```

Run `ch9_1.bc` with `llc -filetype=obj`, and you will find the `Cx` of `jsub Cx` is 0, since `Cx` is calculated by the linker, as shown below. Mips has the same 0 in its `jal` instruction.

```
// jsub _Z5sum_iiiiii translate into 2B 00 00 00
00F0: 2B 00 00 00 01 2D 00 34 00 ED 00 3C 09 DD 00 40
```

The following code will emit `ld $gp, ($gp save slot on stack)` after `jalr` by creating the file `Cpu0EmitGPRestore.cpp`, which runs as a function pass.

Ibdex/chapters/Chapter9_3/CMakeLists.txt

```
Cpu0EmitGPRestore.cpp
```

Ibdex/chapters/Chapter9_3/Cpu0TargetMachine.cpp

```
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {

#ifndef ENABLE_GPRESTORE
    void addPreRegAlloc() override;
#endif

#ifndef ENABLE_GPRESTORE
void Cpu0PassConfig::addPreRegAlloc() {
    if (!Cpu0ReserveGP) {
        // $gp is a caller-saved register.
        addPass(createCpu0EmitGPRestorePass(getCpu0TargetMachine()));
    }
    return;
}
#endif
```

Ibdex/chapters/Chapter9_3/Cpu0.h

```
#ifndef ENABLE_GPRESTORE
FunctionPass *createCpu0EmitGPRestorePass(Cpu0TargetMachine &TM);
#endif
```

Ibdex/chapters/Chapter9_3/Cpu0EmitGPRestore.cpp

```

//===== Cpu0EmitGPRestore.cpp - Emit GP Restore Instruction =====/
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This pass emits instructions that restore $gp right
// after jalr instructions.
//
//=====

#include "Cpu0.h"
#if CH >= CH9_3
#define ENABLE_GPRESTORE

#include "Cpu0TargetMachine.h"
#include "Cpu0MachineFunction.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/TargetInstrInfo.h"

using namespace llvm;

#define DEBUG_TYPE "emit-gp-restore"

namespace {
    struct Inserter : public MachineFunctionPass {

        TargetMachine &TM;

        static char ID;
        Inserter(TargetMachine &tm)
            : MachineFunctionPass(ID), TM(tm) { }

        StringRef getPassName() const override {
            return "Cpu0 Emit GP Restore";
        }

        bool runOnMachineFunction(MachineFunction &F) override;
    };
    char Inserter::ID = 0;
} // end of anonymous namespace

bool Inserter::runOnMachineFunction(MachineFunction &F) {
    Cpu0FunctionInfo *Cpu0FI = F.getInfo<Cpu0FunctionInfo>();
    const TargetSubtargetInfo *STI = TM.getSubtargetImpl(F.getFunction());
    const TargetInstrInfo *TII = STI->getInstrInfo();
}

```

(continues on next page)

(continued from previous page)

```
if ((TM.getRelocationModel() != Reloc::PIC_) ||
    (!Cpu0FI->globalBaseRegFixed()))
    return false;

bool Changed = false;
int FI = Cpu0FI->getGPFIndex();

for (MachineFunction::iterator MFI = F.begin(), MFE = F.end();
     MFI != MFE; ++MFI) {
    MachineBasicBlock& MBB = *MFI;
    MachineBasicBlock::iterator I = MFI->begin();

    /// isEHPad - Indicate that this basic block is entered via an
    /// exception handler.
    // If MBB is a landing pad, insert instruction that restores $gp after
    // EH_LABEL.
    if (MBB.isEHPad()) {
        // Find EH_LABEL first.
        for (; I->getOpcode() != TargetOpcode::EH_LABEL; ++I) ;

        // Insert ld.
        ++I;
        DebugLoc dl = I != MBB.end() ? I->getDebugLoc() : DebugLoc();
        BuildMI(MBB, I, dl, TII->get(Cpu0::LD), Cpu0::GP).addFrameIndex(FI)
            .addImm(0);
        Changed = true;
    }

    while (I != MFI->end()) {
        if (I->getOpcode() != Cpu0::JALR) {
            ++I;
            continue;
        }

        DebugLoc dl = I->getDebugLoc();
        // emit ld $gp, ($gp save slot on stack) after jalr
        BuildMI(MBB, ++I, dl, TII->get(Cpu0::LD), Cpu0::GP).addFrameIndex(FI)
            .addImm(0);
        Changed = true;
    }
}

return Changed;
}

/// createCpu0EmitGPRestorePass - Returns a pass that emits instructions that
/// restores $gp clobbered by jalr instructions.
FunctionPass *llvm::createCpu0EmitGPRestorePass(Cpu0TargetMachine &tm) {
    return new Inserter(tm);
}
```

(continues on next page)

(continued from previous page)

```
#endif
#endif
```

9.6.2 Variable number of arguments

Until now, we supported a fixed number of arguments in formal function definitions (Incoming Arguments). This subsection adds support for a variable number of arguments, as the C language allows this feature.

Run Chapter9_3/ with ch9_3_vararg.cpp and use the clang option clang -target mips-unknown-linux-gnu to get the following result:

```
118-165-76-131:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_vararg.cpp -emit-llvm -o ch9_3_vararg.bc
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_3_vararg.bc -o ch9_3_vararg.cpu0.s
118-165-76-131:input Jonathan$ cat ch9_3_vararg.cpu0.s
.section .mdebug.abi32
.previous
.file "ch9_3_vararg.bc"
.text
.globl _Z5sum_iiz
.align 2
.type _Z5sum_iiz,@function
.ent _Z5sum_iiz           # @_Z5sum_iiz
_Z5sum_iiz:
.frame $fp,24,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -24
st $fp, 20($sp)          # 4-byte Folded Spill
move $fp, $sp
ld $2, 24($fp)           // amount
st $2, 16($fp)           // amount
addiu $2, $zero, 0
st $2, 12($fp)           // i
st $2, 8($fp)            // val
st $2, 4($fp)             // sum
addiu $3, $fp, 28
st $3, 0($fp)             // arg_ptr = 2nd argument = &arg[1],
                           // since &arg[0] = 24($sp)
st $2, 12($fp)
$BB0_1:                  # =>This Inner Loop Header: Depth=1
ld $2, 16($fp)
ld $3, 12($fp)
cmp $sw, $3, $2           // compare(i, amount)
jge $BB0_4
nop
jmp $BB0_2
```

(continues on next page)

(continued from previous page)

```

nop
$BB0_2:                                #    in Loop: Header=BB0_1 Depth=1
    // i < amount
    ld $2, 0($fp)
    addiu $3, $2, 4    // arg_ptr + 4
    st $3, 0($fp)
    ld $2, 0($2)      // *arg_ptr
    st $2, 8($fp)
    ld $3, 4($fp)      // sum
    add $2, $3, $2      // sum += *arg_ptr
    st $2, 4($fp)
# BB#3:                                #    in Loop: Header=BB0_1 Depth=1
    // i >= amount
    ld $2, 12($fp)
    addiu $2, $2, 1    // i++
    st $2, 12($fp)
    jmp $BB0_1
    nop
$BB0_4:
    ld $2, 4($fp)
    move $sp, $fp
    ld $fp, 20($sp)          # 4-byte Folded Reload
    addiu $sp, $sp, 24
    ret $lr
    .set macro
    .set reorder
    .end _Z5sum_iiz
$tmp1:
.size _Z5sum_iiz, ($tmp1)-_Z5sum_iiz

.globl _Z11test_varargv
.align 2
.type _Z11test_varargv,@function
.ent _Z11test_varargv                  # @_Z11test_varargv
_Z11test_varargv:
.frame $sp,88,$lr
.mask 0x00004000,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
    addiu $sp, $sp, -48
    st $lr, 44($sp)          # 4-byte Folded Spill
    st $fp, 40($sp)          # 4-byte Folded Spill
    move $fp, $sp
    .cprestore 32
    addiu $2, $zero, 5
    st $2, 24($sp)
    addiu $2, $zero, 4
    st $2, 20($sp)
    addiu $2, $zero, 3
    st $2, 16($sp)

```

(continues on next page)

(continued from previous page)

```

addiu $2, $zero, 2
st $2, 12($sp)
addiu $2, $zero, 1
st $2, 8($sp)
addiu $2, $zero, 0
st $2, 4($sp)
addiu $2, $zero, 6
st $2, 0($sp)
ld $t9, %call16(_Z5sum_iiz)($gp)
jalr $t9
nop
ld $gp, 28($fp)
st $2, 36($fp)
move $sp, $fp
ld $fp, 40($sp)           # 4-byte Folded Reload
ld $lr, 44($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 48
ret $lr
nop
.set macro
.set reorder
.end _Z11test_varargv
$tmp1:
.size _Z11test_varargv, ($tmp1)-_Z11test_varargv

```

The analysis of output `ch9_3_vararg.cpu0.s` is shown in the comments above.

As described in the code in `# BB#0`, we get the first argument `amount` from `ld $2, 24($fp)`, since the stack size of the callee function `_Z5sum_iiz()` is 24. Then we set the argument pointer, `arg_ptr`, to `0($fp)`, which is `&arg[1]`.

Next, we check `i < amount` in block `$BB0_1`. If `i < amount`, we enter `$BB0_2`. In `$BB0_2`, the code performs `sum += *arg_ptr` and `arg_ptr += 4`. In `# BB#3`, the code increments `i` with `i += 1`.

To support variable numbers of arguments, the following code needs to be added in `Chapter9_3/`.

The file `ch9_3_template.cpp` contains a C++ template example. It can also be translated into Cpu0 backend code.

Ibdex/chapters/Chapter9_3/Cpu0ISelLowering.h

```

class Cpu0TargetLowering : public TargetLowering {
public:
    /// Cpu0CC - This class provides methods used to analyze formal and call
    /// arguments and inquire about calling convention information.
    class Cpu0CC {
public:
    /// Return the function that analyzes variable argument list functions.
    llvm::CCToAssignFn *varArgFn() const;
    ...
};

SDValue lowerVASTART(SDValue Op, SelectionDAG &DAG) const;
SDValue lowerFRAMEADDR(SDValue Op, SelectionDAG &DAG) const;

```

(continues on next page)

(continued from previous page)

```
SDValue lowerRETURNADDR(SDValue Op, SelectionDAG &DAG) const;
SDValue lowerEH_RETURN(SDValue Op, SelectionDAG &DAG) const;
SDValue lowerADD(SDValue Op, SelectionDAG &DAG) const;
```

```
/// writeVarArgRegs - Write variable function arguments passed in registers
/// to the stack. Also create a stack frame object for the first variable
/// argument.
void writeVarArgRegs(std::vector<SDValue> &OutChains, const Cpu0CC &CC,
                     SDValue Chain, const SDLoc &DL, SelectionDAG &DAG) const;
```

```
...
. };
```

[Index/chapters/Chapter9_3/Cpu0ISelLowering.cpp](#)

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
: TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
setOperationAction(ISD::VASTART, MVT::Other, Custom);
```

```
// Support va_arg(): variable numbers (not fixed numbers) of arguments
// (parameters) for function all
setOperationAction(ISD::VAARG, MVT::Other, Expand);
setOperationAction(ISD::VACOPY, MVT::Other, Expand);
setOperationAction(ISD::VAEND, MVT::Other, Expand);

//@llvm.stacksave
// Use the default for now
setOperationAction(ISD::STACKSAVE, MVT::Other, Expand);
setOperationAction(ISD::STACKRESTORE, MVT::Other, Expand);
```

```
...
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    switch (Op.getOpcode())
    {
```

```
        case ISD::VASTART: return lowerVASTART(Op, DAG);
```

```
    }
    return SDValue();
}
```

```

SDValue Cpu0TargetLowering::lowerVASTART(SDValue Op, SelectionDAG &DAG) const {
    MachineFunction &MF = DAG.getMachineFunction();
    Cpu0FunctionInfo *FuncInfo = MF.getInfo<Cpu0FunctionInfo>();

    SDLoc DL = SDLoc(Op);
    SDValue FI = DAG.getFrameIndex(FuncInfo->getVarArgsFrameIndex(),
                                    getPointerTy(MF.getDataLayout()));

    // vastart just stores the address of the VarArgsFrameIndex slot into the
    // memory location argument.
    const Value *SV = cast<SrcValueSDNode>(Op.getOperand(2))->getValue();
    return DAG.getStore(Op.getOperand(0), DL, FI, Op.getOperand(1),
                         MachinePointerInfo(SV));
}

```

```

/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {

if (IsVarArg)
    writeVarArgRegs(OutChains, Cpu0CCInfo, Chain, DL, DAG);

...
}

void Cpu0TargetLowering::Cpu0CC::
analyzeCallOperands(const SmallVectorImpl<ISD::OutputArg> &Args,
                     bool IsVarArg, bool IsSoftFloat, const SDNode *CallNode,
                     std::vector<ArgListEntry> &FuncArgs) {

```

```
    llvm::CCAssignFn *VarFn = varArgFn();
```

```
for (unsigned I = 0; I != NumOpnds; ++I) {
```

```
    if (IsVarArg && !Args[I].IsFixed)
        R = VarFn(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags, CCInfo);
    else
```

```
    ...
}
```

```
llvm::CCAssignFn *Cpu0TargetLowering::Cpu0CC::varArgFn() const {
    if (IsO32)
        return CC_Cpu0O32;
    else // IsS32
        return CC_Cpu0S32;
}
```

```
void Cpu0TargetLowering::writeVarArgRegs(std::vector<SDValue> &OutChains,
                                         const Cpu0CC &CC, SDValue Chain,
                                         const SLDL &DL, SelectionDAG &DAG) const {
    unsigned NumRegs = CC.numIntArgRegs();
    const ArrayRef<MCPhysReg> ArgRegs = CC.intArgRegs();
    const CCState &CCInfo = CC.getCCInfo();
    unsigned Idx = CCInfo.getFirstUnallocated(ArgRegs);
    unsigned RegSize = CC.regSize();
    MVT RegTy = MVT::getIntegerVT(RegSize * 8);
    const TargetRegisterClass *RC = getRegClassFor(RegTy);
    MachineFunction &MF = DAG.getMachineFunction();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MFI.getInfo<Cpu0FunctionInfo>();

    // Offset of the first variable argument from stack pointer.
    int VaArgOffset;

    if (NumRegs == Idx)
        VaArgOffset = alignTo(CCInfo.getNextStackOffset(), RegSize);
    else
        VaArgOffset = (int)CC.reservedArgArea() - (int)(RegSize * (NumRegs - Idx));

    // Record the frame index of the first variable argument
    // which is a value necessary to VASTART.
    int FI = MFI.CreateFixedObject(RegSize, VaArgOffset, true);
    Cpu0FI->setVarArgsFrameIndex(FI);

    // Copy the integer registers that have not been used for argument passing
    // to the argument register save area. For O32, the save area is allocated
    // in the caller's stack frame, while for N32/64, it is allocated in the
    // callee's stack frame.
    for (unsigned I = Idx; I < NumRegs; ++I, VaArgOffset += RegSize) {
        unsigned Reg = addLiveIn(MF, ArgRegs[I], RC);
        SDValue ArgValue = DAG.getCopyFromReg(Chain, DL, Reg, RegTy);
        FI = MFI.CreateFixedObject(RegSize, VaArgOffset, true);
        SDValue PtrOff = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
        SDValue Store = DAG.getStore(Chain, DL, ArgValue, PtrOff,
                                     MachinePointerInfo());
        cast<StoreSDNode>(Store.getNode())->getMemOperand()->setValue(
            (Value *)nullptr);
        OutChains.push_back(Store);
    }
}
```

Ibdex/input/ch9_3_template.cpp

```
#include <stdarg.h>

template<class T>
T sum(T amount, ...)
{
    T i = 0;
    T val = 0;
    T sum = 0;

    va_list vl;
    va_start(vl, amount);
    for (i = 0; i < amount; i++)
    {
        val = va_arg(vl, T);
        sum += val;
    }
    va_end(vl);

    return sum;
}

int test_template()
{
    int a = (int)(sum<int>(6, 0, 1, 2, 3, 4, 5));

    return a; // 15
}

long long test_template_ll()
{
    long long a = (long long)(sum<long long>(6LL, 0LL, 1LL, 2LL, -3LL, 4LL, -5LL));

    return a; // -1
}
```

MIPS QEMU reference⁸ can be downloaded and run with GCC to verify the result using the `printf()` function at this point.

We will verify the correctness of the code in the chapter “Verify backend on Verilog simulator” through the Cpu0 Verilog-language machine.

9.6.3 Dynamic stack allocation support

Even though the C language rarely uses dynamic stack allocation, some other languages rely on it frequently. The following C example demonstrates its use.

Chapter9_3 supports dynamic stack allocation with the following code added.

⁸ <http://developer.mips.com/clang-llvm/>

Ibdex/chapters/Chapter9_2/Cpu0FrameLowering.cpp

```
// Eliminate ADJCALLSTACKDOWN, ADJCALLSTACKUP pseudo instructions
MachineBasicBlock::iterator Cpu0FrameLowering::
eliminateCallFramePseudoInstr(MachineFunction &MF, MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator I) const {
#if CH >= CH9_3 // dynamic alloc
    unsigned SP = Cpu0::SP;

    if (!hasReservedCallFrame(MF)) {
        int64_t Amount = I->getOperand(0).getImm();
        if (I->getOpcode() == Cpu0::ADJCALLSTACKDOWN)
            Amount = -Amount;

        STI.getInstrInfo()->adjustStackPtr(SP, Amount, MBB, I);
    }
#endif // dynamic alloc

    return MBB.erase(I);
}
```

Ibdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {

    unsigned FP = Cpu0::FP;
    unsigned ZERO = Cpu0::ZERO;
    unsigned ADDu = Cpu0::ADDu;

    // if framepointer enabled, set it to point to the stack pointer.
    if (hasFP(MF)) {
        if (Cpu0FI->callsEhDwarf()) {
            BuildMI(MBB, MBBI, dl, TII.get(ADDu), Cpu0::V0).addReg(FP).addReg(ZERO)
                .setMIFlag(MachineInstr::FrameSetup);
        }
        // @ Insert instruction "move $fp, $sp" at this location.
        BuildMI(MBB, MBBI, dl, TII.get(ADDu), FP).addReg(SP).addReg(ZERO)
            .setMIFlag(MachineInstr::FrameSetup);

        // emit ".cfi_def_cfa_register $fp"
        unsigned CFIIndex = MF.addFrameInst(MCCFIInstruction::createDefCfaRegister(
            nullptr, MRI->getDwarfRegNum(FP, true)));
        BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
            .addCFIIndex(CFIIndex);
    }
}

void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
```

```

unsigned FP = Cpu0::FP;
unsigned ZERO = Cpu0::ZERO;
unsigned ADDu = Cpu0::ADDu;

// If framepointer enabled, restore the stack pointer.
if (hasFP(MF)) {
    // Find the first instruction that restores a callee-saved register.
    MachineBasicBlock::iterator I = MBBI;

    for (unsigned i = 0; i < MFI.getCalleeSavedInfo().size(); ++i)
        --I;

    // Insert instruction "move $sp, $fp" at this location.
    BuildMI(MBB, I, DL, TII.get(ADDu), SP).addReg(FP).addReg(ZERO);
}

```

```
}
```

```

unsigned FP = Cpu0::FP;

// Mark $fp as used if function has dedicated frame pointer.
if (hasFP(MF))
    setAliasRegs(MF, SavedRegs, FP);

```

Ibdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
: TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
    setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32, Expand);
```

```
    setStackPointerRegisterToSaveRestore(Cpu0::SP);
```

```
}
```

Ibdex/chapters/Chapter9_3/Cpu0RegisterInfo.cpp

```
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {

    // Reserve FP if this function should have a dedicated frame pointer register.
    if (MF.getSubtarget().getFrameLowering()>hasFP(MF)) {
        Reserved.set(Cpu0::FP);
    }
}
```

```
}
```

```
//- If no eliminateFrameIndex(), it will hang on run.  
// pure virtual method  
// FrameIndex represent objects inside a abstract stack.  
// We must replace FrameIndex with an stack/frame pointer  
// direct reference.  
void Cpu0RegisterInfo::  
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,  
                    unsigned FIOperandNum, RegScavenger *RS) const {
```

```
if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isGPKI(FrameIndex) ||  
    Cpu0FI->isDynAllocFI(FrameIndex))  
    Offset = spOffset;
```

```
}
```

Run Chapter9_3 with ch9_3_alloc.cpp to get the following correct result.

```
118-165-72-242:input Jonathan$ clang -target mips-unknown-linux-gnu -c  
ch9_3_alloc.cpp -emit-llvm -o ch9_3_alloc.bc  
118-165-72-242:input Jonathan$ llvmdis ch9_3_alloc.bc -o ch9_3_alloc.ll  
118-165-72-242:input Jonathan$ cat ch9_3_alloc.ll  
; ModuleID = 'ch9_3_alloc.bc'  
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-  
f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:f80:128:128-n8:16:  
32:64-S128"  
target triple = "x86_64-apple-macosx10.8.0"  
  
define i32 @_Z5sum_iiiiii(i32 %x1, i32 %x2, i32 %x3, i32 %x4, i32 %x5, i32 %x6)  
nounwind uwtable ssp {  
    ...  
    %9 = alloca i8, i32 %8          // int* b = (int*)__builtin_alloca(sizeof(int) * 1 *  
    ↵x1);  
    %10 = bitcast i8* %9 to i32*  
    store i32* %10, i32** %b, align 4  
    ...  
}  
...  
  
118-165-72-242:input Jonathan$ /Users/Jonathan/llvm/test/build/  
bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false  
-relocation-model=pic -filetype=asm ch9_3_alloc.bc -o ch9_3_alloc.cpu0.s  
118-165-72-242:input Jonathan$ cat ch9_3_alloc.cpu0.s  
...  
    .globl _Z10weight_sumiiiii  
.align 2  
.type _Z10weight_sumiiiii,@function  
.ent _Z10weight_sumiiiii    # @_Z10weight_sumiiiii  
_Z10weight_sumiiiii:  
    .frame $fp,48,$lr  
    .mask 0x00005000,-4
```

(continues on next page)

(continued from previous page)

```

.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -48
st $lr, 44($sp)          # 4-byte Folded Spill
st $fp, 40($sp)          # 4-byte Folded Spill
move $fp, $sp
.cprestore 24
ld $2, 68($fp)
ld $3, 64($fp)
ld $t9, 60($fp)
ld $7, 56($fp)
st $4, 36($fp)
st $5, 32($fp)
st $7, 28($fp)
st $t9, 24($fp)
st $3, 20($fp)
st $2, 16($fp)
shl $2, $2, 2    // $2 = sizeof(int) * 1 * x2;
addiu $2, $2, 7
addiu $3, $zero, -8
and $2, $2, $3
addiu $sp, $sp, 0
subu $2, $sp, $2
addu $sp, $zero, $2 // set sp to the bottom of alloca area
addiu $sp, $sp, 0
st $2, 12($fp)
st $2, 8($fp)
ld $2, 12($fp)
ld $3, 28($fp)
st $3, 0($2)    // *b = x3
ld $5, 32($fp)
ld $2, 36($fp)
ld $3, 20($fp)
ld $4, 28($fp)
ld $t9, 24($fp)
ld $7, 16($fp)
addiu $sp, $sp, -24
st $7, 20($sp)
st $t9, 12($sp)
st $4, 8($sp)
shl $3, $3, 1
st $3, 16($sp)
addiu $3, $zero, 3
mul $4, $2, $3
ld $t9, %call16(_Z3sumiiiiii)($gp)
jalr $t9
nop
ld $gp, 24($fp)
addiu $sp, $sp, 24
st $2, 4($fp)

```

(continues on next page)

(continued from previous page)

```

ld  $3, 8($fp)
ld  $3, 0($3)
addu $2, $2, $3
move $sp, $fp
ld  $fp, 40($sp)          # 4-byte Folded Reload
ld  $lr, 44($sp)          # 4-byte Folded Reload
addiu $sp, $sp, -48
ret $lr
nop
.set macro
.set reorder
.end _Z10weight_sumiiiii
$func_end1:
.size _Z10weight_sumiiiii, ($func_end1)-_Z10weight_sumiiiii
...

```

As you can see, dynamic stack allocation requires frame pointer register `fp` support. As shown in the assembly above, the `sp` is adjusted to `sp - 48` when entering the function by the instruction `addiu $sp, $sp, -48`.

Next, `fp` is set to `sp`, which is positioned just above the area allocated by `alloca()`, as illustrated in Fig. 9.8, when the instruction `move $fp, $sp` is encountered.

After that, `sp` is moved to the space just below the `alloca()` allocation. Note that the space pointed to by `b, *b = (int*)__builtin_alloca(sizeof(int) * 2 * x6)`, is allocated at run time, because the size depends on the `x1` variable and cannot be determined at link time.

Fig. 9.9 illustrates how the stack pointer is restored to the caller's stack bottom. As described above, `fp` is set to the address just above the `alloca()` area.

The first step restores `sp` from `fp` using the instruction `move $sp, $fp`. Next, `sp` is adjusted back to the caller's stack bottom using `addiu $sp, $sp, 40`.

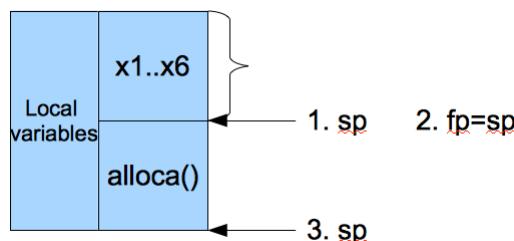


Fig. 9.8: Frame pointer changes when enter function

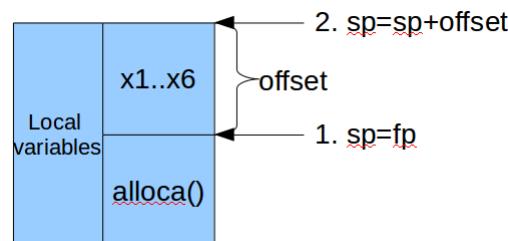


Fig. 9.9: Stack pointer changes when exit function

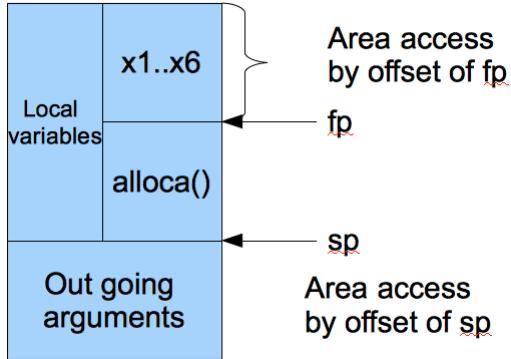


Fig. 9.10: fp and sp access areas

Using `fp` to keep the old stack pointer value is not the only solution. In fact, we can store the size of the `alloca()` spaces at a specific memory address and restore `sp` to its previous value by adding back the size of the `alloca()` area.

Most ABIs, such as MIPS and ARM, access the area above `alloca()` using `fp` and the area below `alloca()` using `sp`, as depicted in Fig. 9.10.

The reason for this design is performance in accessing local variables. Since RISC CPUs commonly use immediate offsets for load and store instructions, using both `fp` and `sp` to access the two separate areas of local variables provides better performance compared to using only `sp`.

```
ld      $2, 64($fp)
st      $3, 4($sp)
```

Cpu0 uses `fp` and `sp` to access the areas above and below `alloca()`, respectively. As shown in `ch9_3_alloc.cpu0.s`, it accesses local variables (above the `alloca()` area) using `fp` offset, and accesses outgoing arguments (below the `alloca()` area) using `sp` offset.

Additionally, the instruction `move $sp, $fp` is an alias for the actual machine instruction `addu $fp, $sp, $zero`. The machine code emitted is the latter, while the former is used for easier readability by users.

This alias is defined by the code added in Chapter3_2 and Chapter3_5, as shown below:

Ibdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
void Cpu0InstPrinter::printInst(const MCInst *MI, uint64_t Address,
                               StringRef Annot, const MCSubtargetInfo &STI,
                               raw_ostream &O) {
    // Try to print any aliases first.
    if (!printAliasInstr(MI, Address, O))
```

Ibdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
class Cpu0InstAlias<string Asm, dag Result, bit Emit = 0b1> :
    InstAlias<Asm, Result, Emit>;
```

```
let Predicates = [Ch3_5] in {
//=====
// Instruction aliases
```

(continues on next page)

(continued from previous page)

```
//=====
def : Cpu0InstAlias<"move $dst, $src",
    (ADDu GPROut:$dst, GPROut:$src, ZERO), 1>;
}
```

Finally, the `MFI->hasVarSizedObjects()` defined in `hasReservedCallFrame()` of `Cpu0SEFrameLowering.cpp` is set to true when the IR contains `%9 = alloca i8, i32 %8,` which corresponds to `(int*)__builtin_alloca(sizeof(int) * 1 * x1);` in C code.

This triggers generation of the assembly instruction `addiu $sp, $sp, -24` for `ch9_3_alloc.cpp` by invoking `adjustStackPtr()` inside `eliminateCallFramePseudoInstr()` of `Cpu0FrameLowering.cpp`.

The file `ch9_3_longlongshift.cpp` demonstrates support for the type **long long shift operations**, which can be tested now as shown below.

Ibdex/input/ch9_3_longlongshift.cpp

```
#include "debug.h"

long long test_longlong_shift1()
{
    long long a = 4;
    long long b = 0x12;
    long long c;
    long long d;

    c = (b >> a); // cc = 0x1
    d = (b << a); // cc = 0x120

    long long e = 0x7FFFFFFFFFFFFFFFLL >> 63;
    return (c+d+e); // 0x121 = 289
}

long long test_longlong_shift2()
{
    long long a = 48;
    long long b = 0x001666660000000a;
    long long c;

    c = (b >> a);

    return c; // 22
}
```

```
114-37-150-209:input Jonathan$ clang -O0 -target mips-unknown-linux-gnu
-c ch9_3_longlongshift.cpp -emit-llvm -o ch9_3_longlongshift.bc

114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/
 llvm-dis ch9_3_longlongshift.bc -o -
...
; Function Attrs: nounwind
define i64 @_Z19test_longlong_shiftv() #0 {
```

(continues on next page)

(continued from previous page)

```
%a = alloca i64, align 8
%b = alloca i64, align 8
%c = alloca i64, align 8
%d = alloca i64, align 8
store i64 4, i64* %a, align 8
store i64 18, i64* %b, align 8
%1 = load i64* %b, align 8
%2 = load i64* %a, align 8
%3 = ash r i64 %1, %2
store i64 %3, i64* %c, align 8
%4 = load i64* %b, align 8
%5 = load i64* %a, align 8
%6 = shl i64 %4, %5
store i64 %6, i64* %d, align 8
%7 = load i64* %c, align 8
%8 = load i64* %d, align 8
%9 = add nsw i64 %7, %8
ret i64 %9
}
...
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_3_longlongshift.bc -o -
.text
.section .mdebug.abi32
.previous
.file "ch9_3_longlongshift.bc"
.globl _Z20test_longlong_shift1v
.align 2
.type _Z20test_longlong_shift1v,@function
.ent _Z20test_longlong_shift1v # @_Z20test_longlong_shift1v
_Z20test_longlong_shift1v:
.frame $fp,56,$lr
.mask 0x00005000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -56
st $lr, 52($sp)          # 4-byte Folded Spill
st $fp, 48($sp)          # 4-byte Folded Spill
move $fp, $sp
addiu $2, $zero, 4
st $2, 44($fp)
addiu $4, $zero, 0
st $4, 40($fp)
addiu $5, $zero, 18
st $5, 36($fp)
st $4, 32($fp)
ld $2, 44($fp)
st $2, 8($sp)
jsub __lshrdi3
nop
```

(continues on next page)

(continued from previous page)

```
st  $3, 28($fp)
st  $2, 24($fp)
ld  $2, 44($fp)
st  $2, 8($sp)
ld  $4, 32($fp)
ld  $5, 36($fp)
jsub __ashldi3
nop
st  $3, 20($fp)
st  $2, 16($fp)
ld  $4, 28($fp)
addu $4, $4, $3
cmp $sw, $4, $3
andi $3, $sw, 1
addu $2, $3, $2
ld  $3, 24($fp)
addu $2, $3, $2
addu $3, $zero, $4
move $sp, $fp
ld  $fp, 48($sp)           # 4-byte Folded Reload
ld  $lr, 52($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 56
ret $lr
nop
.set macro
.set reorder
.end _Z20test_longlong_shift1v
$tmp0:
.size _Z20test_longlong_shift1v, ($tmp0)-_Z20test_longlong_shift1v

.globl _Z20test_longlong_shift2v
.align 2
.type _Z20test_longlong_shift2v,@function
.ent _Z20test_longlong_shift2v # @_Z20test_longlong_shift2v
_Z20test_longlong_shift2v:
.frame $fp,48,$lr
.mask 0x00005000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -48
st  $lr, 44($sp)           # 4-byte Folded Spill
st  $fp, 40($sp)           # 4-byte Folded Spill
move $fp, $sp
addiu $2, $zero, 48
st  $2, 36($fp)
addiu $2, $zero, 0
st  $2, 32($fp)
addiu $5, $zero, 10
st  $5, 28($fp)
lui $2, 22
ori $4, $2, 26214
```

(continues on next page)

(continued from previous page)

```

st  $4, 24($fp)
ld  $2, 36($fp)
st  $2, 8($sp)
jsub __lshrdi3
nop
st  $3, 20($fp)
st  $2, 16($fp)
move $sp, $fp
ld  $fp, 40($sp)          # 4-byte Folded Reload
ld  $lr, 44($sp)          # 4-byte Folded Reload
addiu $sp, $sp, 48
ret $lr
nop
.set macro
.set reorder
.end _Z20test_longlong_shift2v
$tmp1:
.size _Z20test_longlong_shift2v, ($tmp1)-_Z20test_longlong_shift2v

```

9.6.4 Variable sized array support

LLVM supports variable sized arrays (VLA) as introduced in C99⁹¹⁰. The following code is added to support this feature. These intrinsics are set to expand, meaning LLVM replaces them with other DAG nodes during code generation.

Ibdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```

SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    switch (Op.getOpcode())
    {

        // Use the default for now
        setOperationAction(ISD::STACKSAVE,           MVT::Other, Expand);
        setOperationAction(ISD::STACKRESTORE,         MVT::Other, Expand);

        ...
    }
    ...
}

```

Ibdex/input/ch9_3_stacksave.cpp

```

int test_stacksaverestore(unsigned x) {
    // CHECK: call i8* @llvm.stacksave()
    char s1[x];
    s1[x] = 5;
}

```

(continues on next page)

⁹ <http://www.llvm.org/docs/LangRef.html#llvm-stacksave-intrinsic>

¹⁰ https://en.wikipedia.org/wiki/Variable-length_array

(continued from previous page)

```

return s1[x];
// CHECK: call void @_llvm.stackrestore(i8*
}

```

```

JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_stacksave.cpp -emit-llvm -o ch9_3_stacksave.bc
JonathantekiiMac:input Jonathan$ llvm-dis ch9_3_stacksave.bc -o -
define i32 @_Z21test_stacksaverestorej(i32 zeroext %x) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i8*
    %3 = alloca i32
    store i32 %x, i32* %1, align 4
    %4 = load i32, i32* %1, align 4
    %5 = call i8* @_llvm.stacksave()
    store i8* %5, i8** %2
    %6 = alloca i8, i32 %4, align 1
    %7 = load i32, i32* %1, align 4
    %8 = getelementptr inbounds i8, i8* %6, i32 %7
    store i8 %5, i8* %8, align 1
    %9 = load i32, i32* %1, align 4
    %10 = getelementptr inbounds i8, i8* %6, i32 %9
    %11 = load i8, i8* %10, align 1
    %12 = sext i8 %11 to i32
    store i32 %1, i32* %3
    %13 = load i8*, i8** %2
    call void @_llvm.stackrestore(i8* %13)
    ret i32 %12
}

JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_3_stacksave.bc -o -
...

```

9.6.5 Function related Intrinsics support

I believe these LLVM intrinsic IRs are used for implementing exception handling¹¹¹². With these IRs, a programmer can record the frame address and return address, which can be used in C++ programs to implement exception handlers, as shown in the example below.

To support these LLVM intrinsic IRs, the following code is added to the Cpu0 backend.

Ibdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```

Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {

```

¹¹ <http://llvm.org/docs/ExceptionHandling.html#overview>

¹² <http://llvm.org/docs/LangRef.html#llvm-returnaddress-intrinsic>

```

setOperationAction(ISD::EH_RETURN, MVT::Other, Custom);

setOperationAction(ISD::ADD, MVT::i32, Custom);

}

SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    switch (Op.getOpcode())
    {

case ISD::FRAMEADDR:      return lowerFRAMEADDR(Op, DAG);
case ISD::RETURNADDR:     return lowerRETURNADDR(Op, DAG);
case ISD::EH_RETURN:      return lowerEH_RETURN(Op, DAG);
case ISD::ADD:            return lowerADD(Op, DAG);

    ...
}
    ...
}

```

```

SDValue Cpu0TargetLowering::
lowerFRAMEADDR(SDValue Op, SelectionDAG &DAG) const {
    // check the depth
    assert((cast<ConstantSDNode>(Op.getOperand(0))->getZExtValue() == 0) &&
           "Frame address can only be determined for current frame.");

    MachineFrameInfo &MFI = DAG.getMachineFunction().getFrameInfo();
    MFI.setFrameAddressIsTaken(true);
    EVT VT = Op.getValueType();
    SDLoc DL(Op);
    SDValue FrameAddr = DAG.getCopyFromReg(
        DAG.getEntryNode(), DL, Cpu0::FP, VT);
    return FrameAddr;
}

SDValue Cpu0TargetLowering::lowerRETURNADDR(SDValue Op,
                                             SelectionDAG &DAG) const {
    if (verifyReturnAddressArgumentIsConstant(Op, DAG))
        return SDValue();

    // check the depth
    assert((cast<ConstantSDNode>(Op.getOperand(0))->getZExtValue() == 0) &&
           "Return address can be determined only for current frame.");

    MachineFunction &MF = DAG.getMachineFunction();
    MachineFrameInfo &MFI = MF.getFrameInfo();
    MVT VT = Op.getSimpleValueType();
    unsigned LR = Cpu0::LR;

```

(continues on next page)

(continued from previous page)

```

MFI.setReturnAddressIsTaken(true);

// Return LR, which contains the return address. Mark it an implicit live-in.
unsigned Reg = MF.addLiveIn(LR, getRegClassFor(VT));
return DAG.getCopyFromReg(DAG.getEntryNode(), SDLoc(Op), Reg, VT);
}

// An EH_RETURN is the result of lowering llvm.eh.return which in turn is
// generated from _builtin_eh_return (offset, handler)
// The effect of this is to adjust the stack pointer by "offset"
// and then branch to "handler".
SDValue Cpu0TargetLowering::lowerEH_RETURN(SDValue Op, SelectionDAG &DAG)
    const {
    MachineFunction &MF = DAG.getMachineFunction();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    Cpu0FI->setCallsEhReturn();
    SDValue Chain      = Op.getOperand(0);
    SDValue Offset     = Op.getOperand(1);
    SDValue Handler   = Op.getOperand(2);
    SDLoc DL(Op);
    EVT Ty = MVT::i32;

    // Store stack offset in V1, store jump target in V0. Glue CopyToReg and
    // EH_RETURN nodes, so that instructions are emitted back-to-back.
    unsigned OffsetReg = Cpu0::V1;
    unsigned AddrReg = Cpu0::V0;
    Chain = DAG.getCopyToReg(Chain, DL, OffsetReg, Offset, SDValue());
    Chain = DAG.getCopyToReg(Chain, DL, AddrReg, Handler, Chain.getValue(1));
    return DAG.getNode(Cpu0ISD::EH_RETURN, DL, MVT::Other, Chain,
        DAG.getRegister(OffsetReg, Ty),
        DAG.getRegister(AddrReg, getPointerTy(MF.getDataLayout())),
        Chain.getValue(1));
}

SDValue Cpu0TargetLowering::lowerADD(SDValue Op, SelectionDAG &DAG) const {
    if (Op->getOperand(0).getOpcode() != ISD::FRAMEADDR
        || cast<ConstantSDNode>
            (Op->getOperand(0).getOperand(0))->getZExtValue() != 0
        || Op->getOperand(1).getOpcode() != ISD::FRAME_TO_ARGS_OFFSET)
        return SDValue();

    MachineFunction &MF = DAG.getMachineFunction();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    Cpu0FI->setCallsEhDwarf();
    return Op;
}

```

frameaddress and returnaddress intrinsics

Run the following input to get the corresponding result.

lbdex/input/ch9_3_frame_return_addr.cpp

```
int display_frameaddress() {
    return (int)__builtin_frame_address(0);
}

extern int fn();

int display_returnaddress() {
    int a = (int)__builtin_return_address(0);
    fn();
    return a;
}
```

```
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
 llvm-dis ch9_3_frame_return_addr.bc -o -
 ...
; Function Attrs: nounwind
define i32 @_Z20display_frameaddressv() #0 {
    %1 = call i8* @llvm.frameaddress(i32 0)
    %2 = ptrtoint i8* %1 to i32
    ret i32 %2
}

; Function Attrs: nounwind readnone
declare i8* @llvm.frameaddress(i32) #1

define i32 @_Z22display_returnaddressv() #2 {
    %a = alloca i32, align 4
    %1 = call i8* @llvm.returnaddress(i32 0)
    %2 = ptrtoint i8* %1 to i32
    store i32 %2, i32* %a, align 4
    %3 = call i32 @_Z2fnv()
    %4 = load i32, i32* %a, align 4
    ret i32 %4
}

JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -filetype=asm ch9_3_frame_return_addr.bc
-o -
.text
.section .mdebug.abi032
.previous
.file "ch9_3_frame_return_addr.bc"
.globl _Z20display_frameaddressv
.align 2
.type _Z20display_frameaddressv,@function
.ent _Z20display_frameaddressv # @_Z20display_frameaddressv
_Z20display_frameaddressv:
```

(continues on next page)

(continued from previous page)

```

.frame $fp,8,$lr
.mask      0x00001000,-4
.set      noreorder
.set      nomacro

# BB#0:
    addiu $sp, $sp, -8
    st      $fp, 4($sp)                                # 4-
    ↪byte Folded Spill
    move   $fp, $sp
    addu   $2, $zero, $fp
    move   $sp, $fp
    ld     $fp, 4($sp)                                # 4-
    ↪byte Folded Reload
    addiu $sp, $sp, 8
    ret $lr
    nop
    .set      macro
    .set      reorder
    .end      _Z20display_frameaddressv

$func_end0:
    .size _Z20display_frameaddressv, ($func_end0)-_Z20display_frameaddressv

    .globl _Z22display_returnaddress1v
    .align 2
    .type _Z22display_returnaddress1v,@function
    .ent   _Z22display_returnaddress1v # @_Z22display_returnaddress1v

_Z22display_returnaddress1v:
    .cfi_startproc
    .frame $fp,24,$lr
    .mask      0x00005000,-4
    .set      noreorder
    .set      nomacro

# BB#0:
    addiu $sp, $sp, -24
$tmp0:
    .cfi_def_cfa_offset 24
    st      $lr, 20($sp)                                # 4-byte Folded
    ↪Spill
    st      $fp, 16($sp)                                # 4-byte Folded
    ↪Spill
$tmp1:
    .cfi_offset 14, -4
$tmp2:
    .cfi_offset 12, -8
    move   $fp, $sp
$tmp3:
    .cfi_def_cfa_register 12
    st      $lr, 12($fp)
    jsub   _Z2fnv
    nop
    ld     $2, 12($fp)
    move   $sp, $fp

```

(continues on next page)

(continued from previous page)

```

        ld      $fp, 16($sp)                                # 4-byte Folded_
↳Reload
        ld      $lr, 20($sp)                                # 4-byte Folded_
↳Reload
        addiu $sp, $sp, 24
        ret $lr
        nop
        .set   macro
        .set   reorder
        .end   _Z22display_returnaddress1v
$func_end1:
        .size  _Z22display_returnaddress1v, ($func_end1)-_Z22display_returnaddress1v
        .cfi_endproc
    
```

The `asm ld $2, 12($fp)` in function `_Z22display_returnaddress1v` reloads `$lr` to `$2` after `jsub _Z3fnv`. The reason that Cpu0 doesn't produce `addiu $2, $zero, $lr` is that, if a buggy program in `_Z3fnv` modifies the `$lr` value without following the ABI, then it will load an incorrect `$lr` into `$2`.

The following code kills the `$lr` register and makes the reference to `$lr` by loading from a stack slot rather than using the register directly.

Ibdex/chapters/Chapter9_1/Cpu0SEFrameLowering.cpp

```

bool Cpu0SEFrameLowering::
spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                           MachineBasicBlock::iterator MI,
                           const std::vector<CalleeSavedInfo> &CSI,
                           const TargetRegisterInfo *TRI) const {
    ...
    for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
        // Add the callee-saved register as live-in. Do not add if the register is
        // LR and return address is taken, because it has already been added in
        // method Cpu0TargetLowering::LowerRETURNADDR.
        // It's killed at the spill, unless the register is LR and return address
        // is taken.
        unsigned Reg = CSI[i].getReg();
        bool IsRAAndRetAddrIsTaken = (Reg == Cpu0::LR)
            && MF->getFrameInfo()->isReturnAddressTaken();
        if (!IsRAAndRetAddrIsTaken)
            EntryBlock->addLiveIn(Reg);

        // Insert the spill to the stack frame.
        bool IsKill = !IsRAAndRetAddrIsTaken;
        const TargetRegisterClass *RC = TRI->getMinimalPhysRegClass(Reg);
        TII.storeRegToStackSlot(*EntryBlock, MI, Reg, IsKill,
                               CSI[i].getFrameIdx(), RC, TRI);
    }
    ...
}
    
```

eh.return intrinsic

Considering the following code,

unwind example

```
int func() {
    if (...) {
        throw std::bad_alloc();
    }
}

int A() {
    try {
        func();
    }
    catch(...) {
        ...
    }
}

int B() {
    try {
        func();
        A();
    }
    catch(...) {
        ...
    }
}
```

When B() -> calls func() -> exception occurs, the frame is unwound to B and handled by B's exception handler. When B() -> calls A() -> calls func() -> exception occurs, the frame is unwound to A and handled by A's exception handler.

`__builtin_eh_return(offset, handler)` adjusts the stack by the given offset and then jumps to the handler. `__builtin_eh_return` is used in the GCC unwinder (libgcc), but not in the LLVM unwinder (libunwind)¹³.

Besides `lowerRETURNADDR()` in `Cpu0ISelLowering`, the following code is only for `eh.return` support. It can run with the input `ch9_3_detect_exception.cpp` as shown below.

Ibdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {

    if (Cpu0FI->callsEhReturn()) {
        // Insert instructions that spill eh data registers.
        for (int I = 0; I < ABI.EhDataRegSize(); ++I) {
            if (!MBB.isLiveIn(ABI.GetEhDataReg(I)))
                MBB.addLiveIn(ABI.GetEhDataReg(I));
            TII.storeRegToStackSlot(MBB, MBBI, ABI.GetEhDataReg(I), false,
                                   Cpu0FI->getEhDataRegFI(I), RC, &RegInfo);
        }
    }
}
```

(continues on next page)

¹³ <https://llvm.org/docs/ExceptionHandling.html#exception-handling-support-on-the-target>

(continued from previous page)

```
// Emit .cfi_offset directives for eh data registers.
for (int I = 0; I < ABI.EhDataRegSize(); ++I) {
    int64_t Offset = MFI.getOffset(Cpu0FI->getEhDataRegFI(I));
    unsigned Reg = MRI->getDwarfRegNum(ABI.GetEhDataReg(I), true);
    unsigned CFIIndex = MF.addFrameInst(
        MCCFIInstruction::createOffset(nullptr, Reg, Offset));
    BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
        .addCFIIndex(CFIIndex);
}
}
```

```
...
```

```
void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
```

```
if (Cpu0FI->callsEhReturn()) {
    const TargetRegisterClass *RC = &Cpu0::GPROutRegClass;

    // Find first instruction that restores a callee-saved register.
    MachineBasicBlock::iterator I = MBBI;
    for (unsigned i = 0; i < MFI.getCalleeSavedInfo().size(); ++i)
        --I;

    // Insert instructions that restore eh data registers.
    for (int J = 0; J < ABI.EhDataRegSize(); ++J) {
        TII.loadRegFromStackSlot(MBB, I, ABI.GetEhDataReg(J),
                               Cpu0FI->getEhDataRegFI(J), RC, &RegInfo);
    }
}
```

```
...
```

```
// This method is called immediately before PrologEpilogInserter scans the
// physical registers used to determine what callee saved registers should be
// spilled. This method is optional.
void Cpu0SEFrameLowering::determineCalleeSaves(MachineFunction &MF,
                                                BitVector &SavedRegs,
                                                RegScavenger *RS) const {
```

```
// Create spill slots for eh data registers if function calls eh_return.
if (Cpu0FI->callsEhReturn())
    Cpu0FI->createEhDataRegsFI();
```

```
...
```

Ibdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
// Exception handling related node and instructions.
// The conversion sequence is:
// ISD::EH_RETURN -> Cpu0ISD::EH_RETURN ->
// CPU0eh_return -> (stack change + indirect branch)
//
// CPU0eh_return takes the place of regular return instruction
// but takes two arguments (V1, V0) which are used for storing
// the offset and return address respectively.
def SDT_Cpu0EHRET : SDTypeProfile<0, 2, [SDTCisInt<0>, SDTCisPtrTy<1>]>;

def CPU0ehret : SDNode<"Cpu0ISD::EH_RETURN", SDT_Cpu0EHRET,
                     [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

let Uses = [V0, V1], isTerminator = 1, isReturn = 1, isBarrier = 1 in {
    def CPU0eh_return32 : Cpu0Pseudo<(outs), (ins GPROut:$spoff, GPROut:$dst), "", 
                           [(CPU0ehret GPROut:$spoff, GPROut:$dst)]>;
}
}
```

Ibdex/chapters/Chapter9_3/Cpu0SEInstrInfo.h

```
void expandEhReturn(MachineBasicBlock &MBB,
                     MachineBasicBlock::iterator I) const;
```

Ibdex/chapters/Chapter9_3/Cpu0SEInstrInfo.cpp

```
/// Expand Pseudo instructions into real backend instructions
bool Cpu0SEInstrInfo::expandPostRAPseudo(MachineInstr &MI) const {

    case Cpu0::CPU0eh_return32:
        expandEhReturn(MBB, MI);
        break;

    ...
}

void Cpu0SEInstrInfo::expandEhReturn(MachineBasicBlock &MBB,
                                     MachineBasicBlock::iterator I) const {
    // This pseudo instruction is generated as part of the lowering of
    // ISD::EH_RETURN. We convert it to a stack increment by OffsetReg, and
    // indirect jump to TargetReg
    unsigned ADDU = Cpu0::ADDU;
    unsigned SP = Cpu0::SP;
    unsigned LR = Cpu0::LR;
    unsigned T9 = Cpu0::T9;
    unsigned ZERO = Cpu0::ZERO;
    unsigned OffsetReg = I->getOperand(0).getReg();
    unsigned TargetReg = I->getOperand(1).getReg();

    // addu $lr, $v0, $zero
```

(continues on next page)

(continued from previous page)

```
// addu $sp, $sp, $v1
// jr    $lr (via RetLR)
const TargetMachine &TM = MBB.getParent()->getTarget();
if (TM.isPositionIndependent())
    BuildMI(MBB, I, I->getDebugLoc(), get(ADDU), T9)
        .addReg(TargetReg)
        .addReg(ZERO);
BuildMI(MBB, I, I->getDebugLoc(), get(ADDU), LR)
    .addReg(TargetReg)
    .addReg(ZERO);
BuildMI(MBB, I, I->getDebugLoc(), get(ADDU), SP).addReg(SP).addReg(OffsetReg);
expandRetLR(MBB, I);
}
```

Ibdex/input/ch9_3_detect_exception.cpp

```
bool exceptionOccur = false;
void* returnAddr;

// Even though __builtin_frame_address is useless in this example, I believe
// it will be used in real exception handler implementation. Because in real
// implementation, the exception handler keeps a table and decide which function
// should be triggered for a specific exception and hand over to it.
// The hand over process needs unwinding the stack frame. The stack frame address
// can be gotten by calling __builtin_frame_address in the charged function.
void exception_handler() {
    exceptionOccur = true;
    int frameaddr = (int) __builtin_frame_address(0);
    __builtin_eh_return(0, returnAddr); // no warning, eh_return never returns.
}

__attribute__ ((weak))
int test_detect_exception(bool exception) {
    exceptionOccur = false;
    void* handler = (void*)(&exception_handler);
    if (exception) {
        returnAddr = __builtin_return_address(0);
        __builtin_eh_return(0, handler); // no warning, eh_return never returns.
    }
    else {
        return 0;
    }
}
```

```
114-37-150-48:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_detect_exception.cpp -emit-llvm -o ch9_3_detect_exception.bc
114-37-150-48:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch9_3_detect_exception.bc -o -
; ModuleID = 'ch9_3_detect_exception.bc'
target datalayout = "E-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64"
```

(continues on next page)

(continued from previous page)

```
target triple = "mips-unknown-linux-gnu"

@exceptionOccur = global i8 0, align 1
@returnAddr = global i8* null, align 4

; Function Attrs: nounwind
define void @_Z17exception_handlerv() #0 {
    %frameaddr = alloca i32, align 4
    store i8 1, i8* @exceptionOccur, align 1
    %1 = call i8* @llvm.frameaddress(i32 0)
    %2 = ptrtoint i8* %1 to i32
    store i32 %2, i32* %frameaddr, align 4
    %3 = load i8*, i8** @returnAddr, align 4
    call void @llvm.eh.return.i32(i32 0, i8* %3)
    unreachable
                                ; No predecessors!
    ret void
}

; Function Attrs: nounwind readnone
declare i8* @llvm.frameaddress(i32) #1

; Function Attrs: nounwind
declare void @llvm.eh.return.i32(i32, i8*) #2

define weak i32 @_Z21test_detect_exceptionb(i1 zeroext %exception) #3 {
    %1 = alloca i8, align 1
    %handler = alloca i8*, align 4
    %2 = zext i1 %exception to i8
    store i8 %2, i8* %1, align 1
    store i8 0, i8* @exceptionOccur, align 1
    store i8* bitcast (void ()* @_Z17exception_handlerv to i8*), i8** %handler, align 4
    %3 = load i8, i8* %1, align 1
    %4 = trunc i8 %3 to i1
    br i1 %4, label %5, label %8

; <label>:5                                ; preds = %0
    %6 = call i8* @llvm.returnaddress(i32 0)
    store i8* %6, i8** %returnAddr, align 4
    %7 = load i8*, i8** %handler, align 4
    call void @llvm.eh.return.i32(i32 0, i8* %7)
    unreachable

; <label>:8                                ; preds = %0
    ret i32 0
}

; Function Attrs: nounwind readnone
declare i8* @llvm.returnaddress(i32) #1

attributes #0 = { nounwind ... }
attributes #1 = { nounwind readnone }
```

(continues on next page)

(continued from previous page)

```

attributes #2 = { nounwind }
attributes #3 = { "less-precise-fpmad"="false" ... }
...

114-37-150-48:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
ch9_3_detect_exception.bc -o -
.text
.section .mdebug.abi032
.previous
.file "ch9_3_detect_exception.bc"
.globl _Z17exception_handlerv
.align 2
.type _Z17exception_handlerv,@function
.ent _Z17exception_handlerv # @_Z17exception_handlerv
_Z17exception_handlerv:
.frame $fp,16,$lr
.mask 0x00001000,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -16
st $fp, 12($sp)           # 4-byte Folded Spill
st $4, 4($fp)
st $5, 0($fp)
move $fp, $sp
lui $2, %got_hi(exceptionOccur)
addu $2, $2, $gp
ld $2, %got_lo(exceptionOccur) ($2)
addiu $3, $zero, 1
sb $3, 0($2)
st $fp, 8($fp)
lui $2, %got_hi(returnAddr)
addu $2, $2, $gp
ld $2, %got_lo(returnAddr) ($2)
ld $2, 0($2)
addiu $3, $zero, 0
move $sp, $fp
ld $4, 4($fp)
ld $5, 0($fp)
ld $fp, 12($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 16
move $t9, $2
move $lr, $2
addu $sp, $sp, $3
ret $lr
nop
.set macro
.set reorder
.end _Z17exception_handlerv
$func_end0:

```

(continues on next page)

(continued from previous page)

```
.size _Z17exception_handlerv, ($func_end0)-_Z17exception_handlerv

.weak _Z21test_detect_exceptionb
.align 2
.type _Z21test_detect_exceptionb,@function
.ent _Z21test_detect_exceptionb # @_Z21test_detect_exceptionb
_Z21test_detect_exceptionb:
.cfi_startproc
.frame $fp,24,$lr
.mask 0x00001000,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -24
$tmp0:
.cfi_def_cfa_offset 24
st $fp, 20($sp)           # 4-byte Folded Spill
$tmp1:
.cfi_offset 12, -4
st $4, 8($fp)
st $5, 4($fp)
$tmp2:
.cfi_offset 4, -16
$tmp3:
.cfi_offset 5, -20
move $fp, $sp
$tmp4:
.cfi_def_cfa_register 12
sb $4, 16($fp)
lui $2, %got_hi(exceptionOccur)
addu $2, $2, $gp
ld $2, %got_lo(exceptionOccur)($2)
addiu $3, $zero, 0
sb $3, 0($2)
lui $2, %got_hi(_Z17exception_handlerv)
addu $2, $2, $gp
ld $2, %got_lo(_Z17exception_handlerv)($2)
st $2, 12($fp)
lbu $2, 16($fp)
andi $2, $2, 1
beq $2, $zero, .LBB1_2
nop
jmp .LBB1_1
nop
.LBB1_2:
addiu $2, $zero, 0
move $sp, $fp
ld $4, 8($fp)
ld $5, 4($fp)
ld $fp, 20($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 24
```

(continues on next page)

(continued from previous page)

```

ret $lr
nop
.LBB1_1:
    lui $2, %got_hi(returnAddr)
    addu $2, $2, $gp
    ld $2, %got_lo(returnAddr)($2)
    st $lr, 0($2)
    ld $2, 12($fp)
    addiu $3, $zero, 0
    move $sp, $fp
    ld $4, 8($fp)
    ld $5, 4($fp)
    ld $fp, 20($sp)           # 4-byte Folded Reload
    addiu $sp, $sp, 24
    move $t9, $2
    move $lr, $2
    addu $sp, $sp, $3
    ret $lr
nop
.set macro
.set reorder
.end _Z21test_detect_exceptionb
$func_end1:
.size _Z21test_detect_exceptionb, ($func_end1)-_Z21test_detect_exceptionb
.cfi_endproc

.type exceptionOccur,@object  # @exceptionOccur
.bss
.globl exceptionOccur
exceptionOccur:
.byte 0                      # 0x0
.size exceptionOccur, 1

.type returnAddr,@object      # @returnAddr
.globl returnAddr
.align 2
returnAddr:
.4byte 0
.size returnAddr, 4
...

```

If you disable `__attribute__((weak))` in the C file, then the IR will have `nounwind` in attributes #3. The side effect in the ASM output is that no `.cfi_offset` is issued, as seen in the function `exception_handler()`.

This example code of exception handler implementation can get the frame, return address, and call the exception handler by calling `__builtin_xxx` in Clang using the C language, without introducing any assembly instruction. This example can be verified in the chapter “Cpu0 ELF linker” of the other book “LLVM Tool Chain for Cpu0”¹⁴.

By examining the global variable `exceptionOccur`, which is true or false, the program will set the control flow to `exception_handler()` or skip it accordingly.

¹⁴ <http://jonathan2251.github.io/lbt/lld.html>

eh.dwarf intrinsic

Besides `lowerADD()` in `Cpu0ISelLowering`, the following code is only for supporting `eh.dwarf`. It can be run with the input `eh-dwarf-cfa.ll` as shown below.

Ibindex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
// if framepointer enabled, set it to point to the stack pointer.
if (hasFP(MF)) {
    if (Cpu0FI->callsEhDwarf()) {
        BuildMI(MBB, MBBI, dl, TII.get(ADDu), Cpu0::V0).addReg(FP).addReg(ZERO)
            .setMIFlag(MachineInstr::FrameSetup);
    }
}

...
```

Ibindex/input/eh-dwarf-cfa.ll

```
; RUN: llc -march=cpu0el -mcpu=cpu032II < %s | FileCheck %s

declare i8* @llvm.eh.dwarf.cfa(i32) nounwind
declare i8* @llvm.frameaddress(i32) nounwind readnone

define i8* @f1() nounwind {
entry:
    %x = alloca [32 x i8], align 1
    %0 = call i8* @llvm.eh.dwarf.cfa(i32 0)
    ret i8* %0

; CHECK:         addiu    $sp, $sp, -40
; CHECK:         addu     $2,  $zero, $fp
}

define i8* @f2() nounwind {
entry:
    %x = alloca [65536 x i8], align 1
    %0 = call i8* @llvm.eh.dwarf.cfa(i32 0)
    ret i8* %0

; check stack size (65536 + 8)
; CHECK:         lui      ${[R0:[a-z0-9]+]}, 65535
; CHECK:         addiu   ${[R0]}, ${[R0]}, -8
; CHECK:         addu    $sp, $sp, ${[R0]}

; check return value ($sp + stack size)
; CHECK:         addu    $2,  $zero, $fp
}

define i32 @f3() nounwind {
entry:
```

(continues on next page)

(continued from previous page)

```
%x = alloca [32 x i8], align 1
%0 = call i8* @llvm.eh.dwarf.cfa(i32 0)
%1 = ptrtoint i8* %0 to i32
%2 = call i8* @llvm.frameaddress(i32 0)
%3 = ptrtoint i8* %2 to i32
%add = add i32 %1, %3
ret i32 %add

; CHECK:      addiu    $sp, $sp, -40

; check return value ($fp + stack size + $fp)
; CHECK:      move     $fp, $sp
; CHECK:      addu     $2, $fp, $fp
}
```

bswap intrinsic

Cpu0 supports the LLVM intrinsic `bswap`¹⁵.

Ibdex/chapters/Chapter12_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
: TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {

    setOperationAction(ISD::BSWAP, MVT::i32, Expand);
    setOperationAction(ISD::BSWAP, MVT::i64, Expand);

    ...
}
```

Ibdex/input/ch9_3_bswap.cpp

```
int test_bswap16() {
    volatile int a = 0x1234;
    int result = (__builtin_bswap16(a) ^ 0x3412);

    return result;
}

int test_bswap32() {
    volatile int a = 0x1234;
    int result = (__builtin_bswap32(a) ^ 0x34120000);

    return result;
}
```

(continues on next page)

¹⁵ <http://llvm.org/docs/LangRef.html#llvm-bswap-intrinsics>

(continued from previous page)

```
int test_bswap64() {
    volatile int a = 0x1234;
    int result = (__builtin_bswap64(a) ^ 0x3412000000000000);

    return result;
}

int test_bswap() {
    int result = test_bswap16() + test_bswap32() + test_bswap64();

    return result;
}
```

```
114-37-150-48:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_bswap.cpp -emit-llvm -o ch9_3_bswap.bc
114-37-150-48:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch9_3_bswap.bc -o -
...
define i32 @_Z12test_bswap16v() #0 {
    %a = alloca i32, align 4
    %result = alloca i32, align 4
    store volatile i32 4660, i32* %a, align 4
    %1 = load volatile i32, i32* %a, align 4
    %2 = trunc i32 %1 to i16
    %3 = call i16 @llvm.bswap.i16(i16 %2)
    %4 = zext i16 %3 to i32
    %5 = xor i32 %4, 13330
    store i32 %5, i32* %result, align 4
    %6 = load i32, i32* %result, align 4
    ret i32 %6
}
...
```

9.6.6 Add specific backend intrinsic function

LLVM intrinsic functions are designed to extend LLVM IRs for hardware acceleration in compiler design¹⁶. Many CPUs implement their own intrinsic functions for hardware-specific instructions that improve performance.

Some GPUs use the LLVM infrastructure as their OpenGL/OpenCL backend compiler and rely on many LLVM-extended intrinsic functions.

To demonstrate how to use backend proprietary intrinsic functions to support specific instructions for performance improvement in domain-specific languages, Cpu0 adds an intrinsic function `@llvm.cpu0.gcd` for its greatest common divisor (GCD) instruction.

This instruction demonstrates how to implement a custom intrinsic in LLVM; however, it is not implemented in the Verilog Cpu0 hardware.

The code is as follows,

¹⁶ <https://llvm.org/docs/ExtendingLLVM.html>

Ibdex/llvm/modify/llvm/include/llvm/IR/Intrinsics.td

```
...
include "llvm/IR/IntrinsicsCpu0.td"
...
```

Ibdex/llvm/modify/llvm/include/llvm/IR/IntrinsicsCpu0.td

```
===== IntrinsicsCpu0.td - Defines Mips intrinsics -----*- tablegen -*=====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This file defines all of the CPU0-specific intrinsics.
//
//=====

// __builtin_cpu0_gcd defined in
// https://github.com/Jonathan2251/lbt/blob/master/exlbt/clang/include/clang/Basic/
// BuiltinsCpu0.def
def int_cpu0_gcd : GCCBuiltIn<"__builtin_cpu0_gcd">,
    Intrinsic<[llvm_i32_ty], [llvm_i32_ty, llvm_i32_ty],
    [Commutative, IntrNoMem]>;
```

Ibdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
class IntrinsicArithLogicR<bits<8> op, string instr_asm, SDPatternOperator OpNode,
    InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
    FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
        !strconcat(instr_asm, "\t$ra, $rb, $rc"),
        [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], itin> {
    let shamt = 0;
    let isCommutable = isComm; // e.g. add rb rc = add rc rb
    let isReMaterializable = 1;
}
```

```
def GCD : IntrinsicArithLogicR<0x60, "gcd", int_cpu0_gcd, IIAlu, CPURegs, 1>;
```

When running `llc` with `cpu0_gcd.ll`, it generates the `gcd` machine instruction. Meanwhile, running `cpu0_gcd_soft.ll` results in a call to the `cpu0_gcd_soft` function.

In other words, `@llvm.cpu0.gcd` is an intrinsic function mapped to the `gcd` machine instruction, while `@cpu0_gcd_soft` is a regular function implemented in software.

For undefined intrinsic functions in Cpu0, such as `fmul float %0, %1`, LLVM will compile them into function calls like `jsub fmul` for Cpu0¹⁷.

The file `test_memcpy.ll` is an example of an `IntrWriteMem` instruction, which prevents the operation from being optimized out.

¹⁷ file:///Users/cschen/git/lbd/build/html/othertype.html#float-and-double

9.7 Summary

Now, the Cpu0 backend can handle both integer function calls and control statements, similar to the example code in the LLVM frontend tutorial.

It can also translate some of the C++ object-oriented programming language into Cpu0 instructions without much additional backend effort, because the frontend handles most of the complexity for meeting C++ requirement.

LLVM is a well-structured system that follows compiler theory closely. Any backend of LLVM benefits from this structure.

The best part of the three-tier compiler architecture is that backends will automatically support more languages as the frontend expands its language support, as long as no new IRs are introduced.

ELF SUPPORT

- *ELF format*
 - *ELF header and Section header table*
 - *Relocation Record*
 - *Cpu0 ELF related files*
- *llvm-objdump*
 - *llvm-objdump -t -r*
 - *llvm-objdump -d*
- *Disassembler Structure*

Cpu0 backend generated the ELF format of object files.

The ELF (Executable and Linkable Format) is a common standard file format for executables, object code, shared libraries and core dumps. First published in the System V Application Binary Interface specification, and later in the Tool Interface Standard, it was quickly accepted among different vendors of Unixsystems. In 1999 it was chosen as the standard binary file format for Unix and Unix-like systems on x86 by the x86open project. Please reference¹.

The binary encoding of the Cpu0 instruction set in object files has been verified in previous chapters. However, we did not delve into the ELF file format, such as the ELF header and relocation records, at that time.

In this chapter, you will learn how to use tools such as `llvm-objdump`, `llvm-readelf`, and others to analyze ELF files generated by Cpu0. Through these tools, you will also understand the ELF file format itself.

This chapter introduces these tools to readers because understanding the popular ELF format and analysis tools is valuable. An LLVM compiler engineer is responsible for ensuring that their backend generates correct object files.

With these tools, you can verify the correctness of the generated ELF format.

The Cpu0 author has published a book titled “System Software,” which introduces topics such as assemblers, linkers, loaders, compilers, and operating systems in both concept and practice. It demonstrates how to analyze ELF files using binutils and gcc, and includes example code. This is a Chinese-language book on “System Software.”

The book “System Software”² written by Beck is a well-known resource for explaining what the compiler, linker, and loader produce, and how they work together conceptually. You may refer to it to understand how **Relocation Records** work if you need a refresher or are learning this topic for the first time.

¹ http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

² Leland Beck, System Software: An Introduction to Systems Programming.

³, ⁴, ⁵ are Chinese documents about this topic, available on the Cpu0 author's website.

10.1 ELF format

ELF is a format used in both object and executable files. Therefore, there are two views of it, as shown in Fig. 10.1.

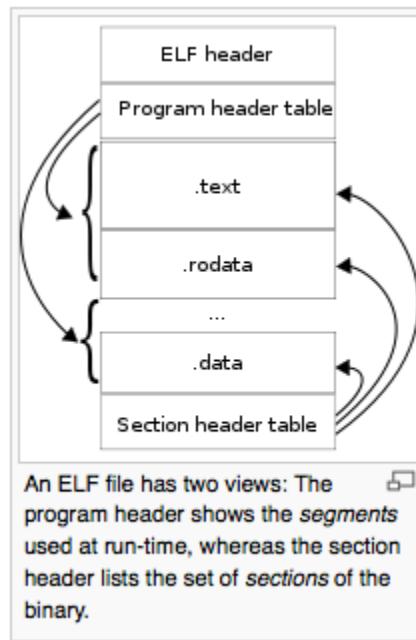


Fig. 10.1: ELF file format overview

As shown in Fig. 10.1, the “Section header table” includes sections .text, .rodata, ..., .data, which are used for code, read-only data, and read/write data, respectively. The “Program header table” includes segments used at run time for code and data.

The definition of segments describes the run-time layout of code and data, while sections describe the link-time layout.

10.1.1 ELF header and Section header table

Let's run Chapter9_3/ with ch6_1.cpp, and dump ELF header information using `llvm-readelf -h` to see what the ELF header contains.

```
input$ ~/llvm/test/build/bin/llc -march=cpu0
-relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o

input$ llvm-readelf -h ch6_1.cpu0.o
Magic: 7f 45 4c 46 01 02 01 03 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, big endian
Version: 1 (current)
OS/ABI: UNIX - GNU
ABI Version: 0
```

(continues on next page)

³ <http://ccckmit.wikidot.com/lk:aout>

⁴ <http://ccckmit.wikidot.com/lk:objfile>

⁵ <http://ccckmit.wikidot.com/lk:elffile>

(continued from previous page)

```
Type: REL (Relocatable file)
Machine: <unknown>: 0xc9
Version: 0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 176 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 40 (bytes)
Number of section headers: 8
Section header string table index: 5
input$
```

```
input$ ~/llvm/test/build/bin/llc
-march=mips -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.mips.o
```

```
input$ llvm-readelf -h ch6_1.mips.o
ELF Header:
  Magic: 7f 45 4c 46 01 02 01 03 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, big endian
  Version: 1 (current)
  OS/ABI: UNIX - GNU
  ABI Version: 0
  Type: REL (Relocatable file)
  Machine: MIPS R3000
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 200 (bytes into file)
  Flags: 0x50001007, noreorder, pic, cpic, o32, mips32
  Size of this header: 52 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 40 (bytes)
  Number of section headers: 9
  Section header string table index: 6
input$
```

```
input$ llvm-readelf -l ch6_1.cpu0.o
```

```
There are no program headers in this file.
input$
```

As shown in the ELF header above, it contains information such as the magic number, version, ABI, and more. The *Machine* field for Cpu0 is listed as unknown, whereas MIPS is recognized as *MIPSR3000*.

This happens because Cpu0 is a unknown CPU supported by the *llvm-readelf* utility.

Let's check the ELF segments information with the following command:

```
input$ llvm-readelf -l ch6_1.cpu0.o
```

```
There are no program headers in this file.  
input$
```

This result is expected because the Cpu0 object file is meant for linking only, not execution because we don't implement linker at this point yet. Therefore, the segment table is empty.

Next, let's check the ELF sections. Each section includes offset and size information.

```
input$ llvm-readelf -S ch6_1.cpu0.o  
There are 10 section headers, starting at offset 0xd4:  
  
Section Headers:  
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al  
[ 0] NULL 00000000 000000 000000 00 0 0 0 0  
[ 1] .text PROGBITS 00000000 000034 000034 00 AX 0 0 4  
[ 2] .rel.text REL 00000000 000310 000018 08 8 1 4  
[ 3] .data PROGBITS 00000000 000068 000004 00 WA 0 0 4  
[ 4] .bss NOBITS 00000000 00006c 000000 00 WA 0 0 4  
[ 5] .eh_frame PROGBITS 00000000 00006c 000028 00 A 0 0 4  
[ 6] .rel.eh_frame REL 00000000 000328 000008 08 8 5 4  
[ 7] .shstrtab STRTAB 00000000 000094 00003e 00 0 0 1  
[ 8] .symtab SYMTAB 00000000 000264 000090 10 9 6 4  
[ 9] .strtab STRTAB 00000000 0002f4 00001b 00 0 0 1  
  
Key to Flags:  
W (write), A (alloc), X (execute), M (merge), S (strings)  
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
O (extra OS processing required) o (OS specific), p (processor specific)  
input$
```

10.1.2 Relocation Record

Cpu0 backend translates global variables as follows:

```
input$ clang -target mips-unknown-linux-gnu -c ch6_1.cpp  
-emit-llvm -o ch6_1.bc  
input$ ~/llvm/test/build/  
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch6_1.bc -o ch6_1.cpu0.s  
input$ cat ch6_1.cpu0.s  
.section .mdebug.abi32  
.previous  
.file "ch6_1.bc"  
.text  
...  
.cfi_startproc  
.frame $sp,8,$lr  
.mask 0x00000000,0  
.set noreorder  
.cupload $t9  
...  
lui $2, %got_hi(gI)  
addu $2, $2, $gp
```

(continues on next page)

(continued from previous page)

```

ld $2, %got_lo(gI)($2)
...
.type gI,@object          # @gI
.data
.globl gI
.align 2
gI:
    .4byte 100             # 0x64
    .size gI, 4

input$ ~/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o
input$ llvm-objdump -s ch6_1.cpu0.o

ch6_1.cpu0.o:      file format elf32-big

Contents of section .text:
// .cupload machine instruction
0000 0fa00000 0daa0000 13aa6000 ..... .
...
0020 002a0000 00220000 012d0000 0ddd0008 .*...".-.....
...
input$


input$ llvm-readelf -tr ch6_1.cpu0.o
There are 8 section headers, starting at offset 0xb0:

Section Headers:
[Nr] Name           Type        Addr     Off      Size     ES   Lk Inf Al
Flags
[ 0]
    NULL           PROGBITS    00000000 000000 000000 00   0   0   0
    [00000000]:
[ 1] .text          PROGBITS    00000000 000034 000044 00   0   0   4
    [00000006]: ALLOC, EXEC
[ 2] .rel.text       REL         00000000 0002a8 000020 08   6   1   4
    [00000000]:
[ 3] .data          PROGBITS    00000000 000078 000008 00   0   0   4
    [00000003]: WRITE, ALLOC
[ 4] .bss           NOBITS     00000000 000080 000000 00   0   0   4
    [00000003]: WRITE, ALLOC
[ 5] .shstrtab      STRTAB     00000000 000080 000030 00   0   0   1
    [00000000]:
[ 6] .symtab        SYMTAB     00000000 0001f0 000090 10   7   5   4

```

(continues on next page)

(continued from previous page)

```
[00000000]:
[ 7] .strtab
    STRTAB      00000000 000280 000025 00    0    0    1
[00000000]:

Relocation section '.rel.text' at offset 0x2a8 contains 4 entries:
Offset     Info      Type          Sym.Value  Sym. Name
00000000  00000805 unrecognized: 5        00000000  _gp_disp
00000004  00000806 unrecognized: 6        00000000  _gp_disp
00000020  00000616 unrecognized: 16       00000004  gI
00000028  00000617 unrecognized: 17       00000004  gI

input$ llvm-readelf -tr ch6_1.mips.o
There are 9 section headers, starting at offset 0xc8:

Section Headers:
[Nr] Name
Type          Addr      Off     Size   ES   Lk Inf Al
Flags
[ 0]
NULL         00000000 000000 000000 00    0    0    0
[00000000]:
[ 1] .text
PROGBITS     00000000 000034 000038 00    0    0    4
[00000006]: ALLOC, EXEC
[ 2] .rel.text
REL          00000000 0002f8 000018 08    7    1    4
[00000000]:
[ 3] .data
PROGBITS     00000000 00006c 000008 00    0    0    4
[00000003]: WRITE, ALLOC
[ 4] .bss
NOBITS       00000000 000074 000000 00    0    0    4
[00000003]: WRITE, ALLOC
[ 5] .reginfo
MIPS_REGINFO 00000000 000074 000018 00    0    0    1
[00000002]: ALLOC
[ 6] .shstrtab
STRTAB       00000000 00008c 000039 00    0    0    1
[00000000]:
[ 7] .symtab
SYMTAB       00000000 000230 0000a0 10    8    6    4
[00000000]:
[ 8] .strtab
STRTAB       00000000 0002d0 000025 00    0    0    1
[00000000]:

Relocation section '.rel.text' at offset 0x2f8 contains 3 entries:
Offset     Info      Type          Sym.Value  Sym. Name
00000000  00000905 R_MIPS_HI16      00000000  _gp_disp
00000004  00000906 R_MIPS_LO16      00000000  _gp_disp
```

(continues on next page)

(continued from previous page)

00000001c	00000709	R_MIPS_GOT16	00000004	gI
-----------	----------	--------------	----------	----

As depicted in [subsection Global Variables Accessing In PIC Addressing Mode](#), it translates “`.cupload %reg`” into the following.

```
// Lower ".cupload $reg" to
// "lui    $gp, %hi(_gp_disp)"
// "ori   $gp, $gp, %lo(_gp_disp)"
// "addu  $gp, $gp, $t9"
```

The `_gp_disp` value is determined by the loader, so it’s undefined in the obj file. You can find both the relocation records for offset 0 and 4 of the `.text` section referring to the `_gp_disp` symbol.

The offset 0 and 4 of the `.text` section correspond to the instructions `lui $gp, %hi(_gp_disp)` and `ori $gp, $gp, %lo(_gp_disp)`, whose encoded object representations are `0fa00000` and `0daa0000`, respectively.

The object file sets the `%hi(_gp_disp)` and `%lo(_gp_disp)` fields to zero, since the loader will determine the actual `_gp_disp` value at runtime and patch these two relocation entries accordingly.

You can verify the correctness of Cpu0’s handling of `%hi(_gp_disp)` and `%lo(_gp_disp)` by comparing them to the MIPS relocation records `R_MIPS_HI(_gp_disp)` and `R_MIPS_LO(_gp_disp)`, even though Cpu0 is not a recognized CPU target by the `llvm-readelf` utility.

The instruction `ld $2, %got(gI)($gp)` behaves similarly. Because the actual address of the `.data` section variable `gI` is unknown at compile time, Cpu0 sets its address to 0 and creates a relocation record at offset 0x00000020 of the `.text` section.

The linker or loader will patch this address at link time (for static linking) or load time (for dynamic linking), depending on how the program is built.

10.1.3 Cpu0 ELF related files

The files `Cpu0ELFObjectWriter.cpp` and `Cpu0OMC*.cpp` are responsible for generating object files (`.o`) in ELF format for the Cpu0 backend.

Most instruction-specific encoding logic is defined in the `Cpu0InstrInfo.td` and `Cpu0RegisterInfo.td` TableGen files. Based on these `.td` descriptions, LLVM automatically translates Cpu0 instructions into the correct binary format for object files.

10.2 llvm-objdump

10.2.1 llvm-objdump -t -r

The `llvm-objdump -tr` command displays symbol table and relocation record information, similar to the output of `llvm-readelf -tr`.

To examine the differences, try running `llvm-objdump` with and without enabling the Cpu0 backend, as shown in the following example:

```
118-165-83-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3.cpp -emit-llvm -o ch9_3.bc
118-165-83-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch9_3.bc -o
ch9_3.cpu0.o

118-165-78-12:input Jonathan$ objdump -t -r ch9_3.cpu0.o
```

(continues on next page)

(continued from previous page)

```
ch9_3.cpu0.o:      file format elf32-big

SYMBOL TABLE:
00000000 1    df *ABS*      00000000 ch9_3.bc
00000000 1    d .text       00000000 .text
00000000 1    d .data       00000000 .data
00000000 1    d .bss        00000000 .bss
00000000 g    F .text       00000084 _Z5sum_iiz
00000084 g    F .text       00000080 main
00000000     *UND*        00000000 _gp_disp

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE          VALUE
00000084 UNKNOWN      _gp_disp
00000088 UNKNOWN      _gp_disp
000000e0 UNKNOWN      _Z5sum_iiz

118-165-83-10:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llvm-objdump -t -r ch9_3.cpu0.o

ch9_3.cpu0.o: file format ELF32-CPU0

RELOCATION RECORDS FOR [.text]:
132 R_CPU0_HI16 _gp_disp
136 R_CPU0_LO16 _gp_disp
224 R_CPU0_CALL16 _Z5sum_iiz

SYMBOL TABLE:
00000000 1    df *ABS*      00000000 ch9_3.bc
00000000 1    d .text       00000000 .text
00000000 1    d .data       00000000 .data
00000000 1    d .bss        00000000 .bss
00000000 g    F .text       00000084 _Z5sum_iiz
00000084 g    F .text       00000080 main
00000000     *UND*        00000000 _gp_disp
```

The *llvm-objdump* tool can correctly display the file format and relocation record information, whereas the GNU *objdump* cannot. This is because the Cpu0-specific relocation record definitions have been added to *ELF.h* within LLVM's source code, enabling *llvm-objdump* to recognize and interpret them properly.

include/llvm/support/ELF.h

```
// Machine architectures
enum {
  ...
  EM_CPU0          = 998, // Document LLVM Backend Tutorial Cpu0
  EM_CPU0_LE       = 999 // EM_CPU0_LE: little endian; EM_CPU0: big endian
}
```

lib/object/ELF.cpp

```
...
StringRef getELFRelocationTypeName(uint32_t Machine, uint32_t Type) {
    switch (Machine) {
        ...
        case ELF::EM_CPU0:
            switch (Type) {
# include "llvm/Support/ELFRelocs/Cpu0.def"
                default:
                    break;
            }
            break;
        ...
    }
}
```

include/llvm/Support/ELFRelocs/Cpu0.def

```
#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif

ELF_RELOC(R_CPU0_NONE, 0)
ELF_RELOC(R_CPU0_32, 2)
ELF_RELOC(R_CPU0_HI16, 5)
ELF_RELOC(R_CPU0_LO16, 6)
ELF_RELOC(R_CPU0_GPREL16, 7)
ELF_RELOC(R_CPU0_LITERAL, 8)
ELF_RELOC(R_CPU0_GOT16, 9)
ELF_RELOC(R_CPU0_PC16, 10)
ELF_RELOC(R_CPU0_CALL16, 11)
ELF_RELOC(R_CPU0_GPREL32, 12)
ELF_RELOC(R_CPU0_PC24, 13)
ELF_RELOC(R_CPU0_GOT_HI16, 22)
ELF_RELOC(R_CPU0_GOT_LO16, 23)
ELF_RELOC(R_CPU0_RELGOT, 36)
ELF_RELOC(R_CPU0_TLS_GD, 42)
ELF_RELOC(R_CPU0_TLS_LDM, 43)
ELF_RELOC(R_CPU0_TLS_DTP_HI16, 44)
ELF_RELOC(R_CPU0_TLS_DTP_LO16, 45)
ELF_RELOC(R_CPU0_TLS_GOTTPREL, 46)
ELF_RELOC(R_CPU0_TLS_TPREL32, 47)
ELF_RELOC(R_CPU0_TLS_TP_HI16, 49)
ELF_RELOC(R_CPU0_TLS_TP_LO16, 50)
ELF_RELOC(R_CPU0_GLOB_DAT, 51)
ELF_RELOC(R_CPU0_JUMP_SLOT, 127)
```

include/llvm/Object/ELFOBJECTFILE.H

```
template<support::endianness target_endianness, bool is64Bits>
error_code ELFOBJECTFILE<target_endianness, is64Bits>
    ::getRelocationValueString(DataRefImpl Rel,
                               SmallVectorImpl<char> &Result) const {
    ...
    case ELF::EM_CPU0: // llvm-objdump -t -r
        res = symname;
        break;
    ...
}

template<support::endianness target_endianness, bool is64Bits>
StringRef ELFOBJECTFILE<target_endianness, is64Bits>
    ::getFileFormatName() const {
    switch(Header->e_ident[ELF::EI_CLASS]) {
    case ELF::ELFCLASS32:
        switch(Header->e_machine) {
        ...
        case ELF::EM_CPU0: // llvm-objdump -t -r
            return "ELF32-CPU0";
        ...
    }
}

template<support::endianness target_endianness, bool is64Bits>
unsigned ELFOBJECTFILE<target_endianness, is64Bits>::getArch() const {
    switch(Header->e_machine) {
    ...
    case ELF::EM_CPU0: // llvm-objdump -t -r
        return (target_endianness == support::little) ?
            Triple::cpu0el : Triple::cpu0;
    ...
}
```

In addition to `llvm-objdump -t -r`, the `llvm-readobj -h` command can be used to display the Cpu0 ELF header information, thanks to the `EM_CPU0` definition added earlier.

10.2.2 llvm-objdump -d

Run the example code from the previous chapter using the command `llvm-objdump -d` to disassemble the ELF file and view its contents in hexadecimal format as shown below:

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch8_1_1.cpp -emit-llvm -o ch8_1_1.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch8_1_1.bc
-o ch8_1_1.cpu0.o
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-objdump -d ch8_1_1.cpu0.o

ch8_1_1.cpu0.o: file format ELF32-unknown

Disassembly of section .text:error: no disassembler for target cpu0-unknown-
```

(continues on next page)

(continued from previous page)

unknown

To support `llvm-objdump`, the following code is added in `Chapter10_1/`. (Note: The `DecoderMethod` for `brtarget24` was added in a previous chapter.)

Ibdex/chapters/Chapter10_1/CMakeLists.txt

```
tablegen(LLVM Cpu0GenDisassemblerTables.inc -gen-disassembler)
```

```
Cpu0Disassembler
```

```
add_subdirectory(Disassembler)
```

Ibdex/chapters/Chapter10_1/Cpu0InstrInfo.td

```
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
    FL<op, (outs), (ins RC:$ra),
        !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
    let rb = 0;
    let imm16 = 0;
//#if CH >= CH10_1 1.5
    let DecoderMethod = "DecodeJumpFR";
//#endif
}
```

```
class JumpLink<bits<8> op, string instr_asm>:
    FJ<op, (outs), (ins calltarget:$target, variable_ops),
        !strconcat(instr_asm, "\t$target"), [(Cpu0JmpLink imm:$target)],
        IIBranch> {
//#if CH >= CH10_1 2
    let DecoderMethod = "DecodeJumpTarget";
//#endif
}
```

Ibdex/chapters/Chapter10_1/Disassembler/CMakeLists.txt

```
add_llvm_component_library(LLVMCpu0Disassembler
    Cpu0Disassembler.cpp

    LINK_COMPONENTS
    MCDisassembler
    Cpu0Info
    Support

    ADD_TO_COMPONENT
    Cpu0
)
```

Ibdex/chapters/Chapter10_1/Disassembler/Cpu0Disassembler.cpp

```
===== Cpu0Disassembler.cpp - Disassembler for Cpu0 -----*- C++ -*=====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// This file is part of the Cpu0 Disassembler.  
//  
//=====//  
  
#include "Cpu0.h"  
  
#include "Cpu0RegisterInfo.h"  
#include "Cpu0Subtarget.h"  
#include "llvm/MC/MCDisassembler/MCDisassembler.h"  
#include "llvm/MC/MCFixedLenDisassembler.h"  
#include "llvm/MC/MCInst.h"  
#include "llvm/MC/MCSubtargetInfo.h"  
#include "llvm/Support/MathExtras.h"  
#include "llvm/Support/TargetRegistry.h"  
  
using namespace llvm;  
  
#define DEBUG_TYPE "cpu0-disassembler"  
  
typedef MCDisassembler::DecodeStatus DecodeStatus;  
  
namespace {  
  
/// Cpu0DisassemblerBase - a disassembler class for Cpu0.  
class Cpu0DisassemblerBase : public MCDisassembler {  
public:  
    /// Constructor - Initializes the disassembler.  
    ///  
    Cpu0DisassemblerBase(const MCSubtargetInfo &STI, MCContext &Ctx,  
                         bool bigEndian) :  
        MCDisassembler(STI, Ctx),  
        IsBigEndian(bigEndian) {}  
  
    virtual ~Cpu0DisassemblerBase() {}  
  
protected:  
    bool IsBigEndian;  
};  
  
/// Cpu0Disassembler - a disassembler class for Cpu032.  
class Cpu0Disassembler : public Cpu0DisassemblerBase {  
public:
```

(continues on next page)

(continued from previous page)

```

/// Constructor      - Initializes the disassembler.
///
Cpu0Disassembler(const MCSubtargetInfo &STI, MCContext &Ctx, bool bigEndian)
    : Cpu0DisassemblerBase(STI, Ctx, bigEndian) {
}

/// getInstruction - See MCDisassembler.
DecodeStatus getInstruction(MCInst &Instr, uint64_t &Size,
                            ArrayRef<uint8_t> Bytes, uint64_t Address,
                            raw_ostream &CStream) const override;
};

} // end anonymous namespace

// Decoder tables for GPR register
static const unsigned CPUREgsTable[] = {
    Cpu0::ZERO, Cpu0::AT, Cpu0::V0, Cpu0::V1,
    Cpu0::A0, Cpu0::A1, Cpu0::T9, Cpu0::T0,
    Cpu0::T1, Cpu0::S0, Cpu0::S1, Cpu0::GP,
    Cpu0::FP, Cpu0::SP, Cpu0::LR, Cpu0::SW
};

// Decoder tables for co-processor 0 register
static const unsigned C0RegsTable[] = {
    Cpu0::PC, Cpu0::EPC
};

static DecodeStatus DecodeCPUREgsRegisterClass(MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder);
static DecodeStatus DecodeGPROutRegisterClass(MCInst &Inst,
                                             unsigned RegNo,
                                             uint64_t Address,
                                             const void *Decoder);
static DecodeStatus DecodeSRRegisterClass(MCInst &Inst,
                                         unsigned RegNo,
                                         uint64_t Address,
                                         const void *Decoder);
static DecodeStatus DecodeC0RegsRegisterClass(MCInst &Inst,
                                             unsigned RegNo,
                                             uint64_t Address,
                                             const void *Decoder);
static DecodeStatus DecodeBranch16Target(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder);
static DecodeStatus DecodeBranch24Target(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder);
static DecodeStatus DecodeJumpTarget(MCInst &Inst,
                                     ...

```

(continues on next page)

(continued from previous page)

```
        unsigned Insn,
        uint64_t Address,
        const void *Decoder);
static DecodeStatus DecodeJumpFR(MCInst &Inst,
                                 unsigned Insn,
                                 uint64_t Address,
                                 const void *Decoder);

static DecodeStatus DecodeMem(MCInst &Inst,
                             unsigned Insn,
                             uint64_t Address,
                             const void *Decoder);
static DecodeStatus DecodeSimm16(MCInst &Inst,
                                 unsigned Insn,
                                 uint64_t Address,
                                 const void *Decoder);

namespace llvm {
extern Target TheCpu0elTarget, TheCpu0Target, TheCpu064Target,
              TheCpu064elTarget;
}

static MCDisassembler *createCpu0Disassembler(
    const Target &T,
    const MCSubtargetInfo &STI,
    MCContext &Ctx) {
    return new Cpu0Disassembler(STI, Ctx, true);
}

static MCDisassembler *createCpu0elDisassembler(
    const Target &T,
    const MCSubtargetInfo &STI,
    MCContext &Ctx) {
    return new Cpu0Disassembler(STI, Ctx, false);
}

extern "C" void LLVMInitializeCpu0Disassembler() {
    // Register the disassembler.
    TargetRegistry::RegisterMCDisassembler(TheCpu0Target,
                                            createCpu0Disassembler);
    TargetRegistry::RegisterMCDisassembler(TheCpu0elTarget,
                                            createCpu0elDisassembler);
}

#if 0
#undef LLVM_DEBUG
#define LLVM_DEBUG(X) X
#endif
#include "Cpu0GenDisassemblerTables.inc"

/// Read four bytes from the ArrayRef and return 32 bit word sorted
/// according to the given endianess
```

(continues on next page)

(continued from previous page)

```

static DecodeStatus readInstruction32(ArrayRef<uint8_t> Bytes, uint64_t Address,
                                     uint64_t &Size, uint32_t &Insn,
                                     bool IsBigEndian) {
    // We want to read exactly 4 Bytes of data.
    if (Bytes.size() < 4) {
        Size = 0;
        return MCDisassembler::Fail;
    }

    if (IsBigEndian) {
        // Encoded as a big-endian 32-bit word in the stream.
        Insn = (Bytes[3] << 0) |
               (Bytes[2] << 8) |
               (Bytes[1] << 16) |
               (Bytes[0] << 24);
    }
    else {
        // Encoded as a small-endian 32-bit word in the stream.
        Insn = (Bytes[0] << 0) |
               (Bytes[1] << 8) |
               (Bytes[2] << 16) |
               (Bytes[3] << 24);
    }

    return MCDisassembler::Success;
}

DecodeStatus
Cpu0Disassembler::getInstruction(MCInst &Instr, uint64_t &Size,
                                  ArrayRef<uint8_t> Bytes,
                                  uint64_t Address,
                                  raw_ostream &CStream) const {
    uint32_t Insn;

    DecodeStatus Result;

    Result = readInstruction32(Bytes, Address, Size, Insn, IsBigEndian);

    if (Result == MCDisassembler::Fail)
        return MCDisassembler::Fail;

    // Calling the auto-generated decoder function.
    Result = decodeInstruction(DecoderTableCpu032, Instr, Insn, Address,
                               this, STI);
    if (Result != MCDisassembler::Fail) {
        Size = 4;
        return Result;
    }

    return MCDisassembler::Fail;
}

```

(continues on next page)

(continued from previous page)

```
static DecodeStatus DecodeCPURegsRegisterClass (MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder) {
    if (RegNo > 15)
        return MCDisassembler::Fail;

    Inst.addOperand(MCOperand::createReg (CPURegsTable[RegNo]));
    return MCDisassembler::Success;
}

static DecodeStatus DecodeGPROutRegisterClass (MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder) {
    return DecodeCPURegsRegisterClass(Inst, RegNo, Address, Decoder);
}

static DecodeStatus DecodeSRRegisterClass (MCInst &Inst,
                                          unsigned RegNo,
                                          uint64_t Address,
                                          const void *Decoder) {
    return DecodeCPURegsRegisterClass(Inst, RegNo, Address, Decoder);
}

static DecodeStatus DecodeC0RegsRegisterClass (MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder) {
    if (RegNo > 1)
        return MCDisassembler::Fail;

    Inst.addOperand(MCOperand::createReg (C0RegsTable[RegNo]));
    return MCDisassembler::Success;
}

// @DecodeMem {
static DecodeStatus DecodeMem (MCInst &Inst,
                             unsigned Insn,
                             uint64_t Address,
                             const void *Decoder) {
// @DecodeMem body {
    int Offset = SignExtend32<16>(Insn & 0xffff);
    int Reg = (int)fieldFromInstruction(Insn, 20, 4);
    int Base = (int)fieldFromInstruction(Insn, 16, 4);

    Inst.addOperand(MCOperand::createReg (CPURegsTable[Reg]));
    Inst.addOperand(MCOperand::createReg (CPURegsTable[Base]));
    Inst.addOperand(MCOperand::createImm(Offset));

    return MCDisassembler::Success;
}
```

(continues on next page)

(continued from previous page)

```

static DecodeStatus DecodeBranch16Target (MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder) {
    int BranchOffset = fieldFromInstruction(Insn, 0, 16);
    if (BranchOffset > 0x8fff)
        BranchOffset = -1*(0x10000 - BranchOffset);
    Inst.addOperand(MCOperand::createImm(BranchOffset));
    return MCDisassembler::Success;
}

/* CBranch instruction define $ra and then imm24; The printOperand() print
   operand 1 (operand 0 is $ra and operand 1 is imm24), so we Create register
   operand first and create imm24 next, as follows,

// Cpu0InstrInfo.td
class CBranch<bits<8> op, string instr_asm, RegisterClass RC,
               list<Register> UseRegs>:
    FJ<op, (outs), (ins RC:$ra, brtarget:$addr),
    !strconcat(instr_asm, "\t$addr"),
    [(brcond RC:$ra, bb:$addr)], IIBranch> {

// Cpu0AsmWriter.inc
void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {
...
    case 3:
        // CMP, JEQ, JGE, JGT, JLE, JLT, JNE
        printOperand(MI, 1, O);
        break;
}
static DecodeStatus DecodeBranch24Target (MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder) {
    int BranchOffset = fieldFromInstruction(Insn, 0, 24);
    if (BranchOffset > 0xffff)
        BranchOffset = -1*(0x1000000 - BranchOffset);
    Inst.addOperand(MCOperand::createReg(Cpu0::SW));
    Inst.addOperand(MCOperand::createImm(BranchOffset));
    return MCDisassembler::Success;
}

static DecodeStatus DecodeJumpTarget (MCInst &Inst,
                                     unsigned Insn,
                                     uint64_t Address,
                                     const void *Decoder) {

    unsigned JumpOffset = fieldFromInstruction(Insn, 0, 24);
    Inst.addOperand(MCOperand::createImm(JumpOffset));
    return MCDisassembler::Success;
}

```

(continues on next page)

(continued from previous page)

```
static DecodeStatus DecodeJumpFR(MCInst &Inst,
                                unsigned Insn,
                                uint64_t Address,
                                const void *Decoder) {
    int Reg_a = (int)fieldFromInstruction(Insn, 20, 4);
    Inst.addOperand(MCOperand::createReg(CPURegsTable[Reg_a]));
// exapin in http://jonathan2251.github.io/lbd/llvmstructure.html#jr-note
    if (CPURegsTable[Reg_a] == Cpu0::LR)
        Inst.setOpcode(Cpu0::RET);
    else
        Inst.setOpcode(Cpu0::JR);
    return MCDisassembler::Success;
}

static DecodeStatus DecodeSimm16(MCInst &Inst,
                                unsigned Insn,
                                uint64_t Address,
                                const void *Decoder) {
    Inst.addOperand(MCOperand::createImm(SignExtend32<16>(Insn)));
    return MCDisassembler::Success;
}
```

As shown in the above code, it adds the `Disassembler` directory to handle the reverse translation from `obj` to assembly. Therefore, `Disassembler/Cpu0Disassembler.cpp` is added, and the `CMakeLists.txt` is modified to build the `Disassembler` directory and enable the disassembler table generated by setting `has_disassembler = 1`. Most of the code is handled by the table defined in `*.td` files.

Not every instruction in the `*.td` files can be disassembled without trouble, even though they can be successfully translated into assembly and `obj`. For those that cannot be disassembled, LLVM provides the "let `DecoderMethod`" keyword to allow programmers to implement their own decode functions.

For example, in `Cpu0`, we define functions such as `DecodeBranch24Target()`, `DecodeJumpTarget()`, and `DecodeJumpFR()` in `Cpu0Disassembler.cpp`. We then inform `llvm-tblgen` by writing "let `DecoderMethod = ...`" in the corresponding instruction definitions or ISD nodes of `Cpu0InstrInfo.td`.

LLVM will call these `DecoderMethods` when the user uses disassembler tools, such as `llvm-objdump -d`.

Finally, `cpu032II` includes all instructions from `cpu032I` and adds some new instructions. When `llvm-objdump -d` is invoked, the function `selectCpu0ArchFeature()` will be called through `createCpu0MCSubtargetInfo()`. Since `llvm-objdump` cannot set CPU options like `l1c -mcpu=cpu032I`, the variable `CPU` in `selectCpu0ArchFeature()` is empty when invoked by `llvm-objdump -d`. To ensure that all instructions are disassembled, we set `Cpu0ArchFeature` to "`+cpu032II`" so that it can disassemble all instructions from `cpu032II` (which includes all instructions from `cpu032I` and adds new ones).

Ibdex/chapters/Chapter10_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
/// Select the Cpu0 Architecture Feature for the given triple and cpu name.
/// The function will be called at command 'llvm-objdump -d' for Cpu0 elf input.
static std::string selectCpu0ArchFeature(const Triple &TT, StringRef CPU) {
    std::string Cpu0ArchFeature;
    if (CPU.empty() || CPU == "generic") {
        if (TT.getArch() == Triple::cpu0 || TT.getArch() == Triple::cpu0el) {
```

(continues on next page)

(continued from previous page)

```

if (CPU.empty() || CPU == "cpu032II") {
    Cpu0ArchFeature = "+cpu032II";
}
else {
    if (CPU == "cpu032I") {
        Cpu0ArchFeature = "+cpu032I";
    }
}
}
return Cpu0ArchFeature;
}

```

Now, run Chapter10_1/ with command `llvm-objdump -d ch8_1_1.cpu0.o` will get the following result.

```

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj
ch8_1_1.bc -o ch8_1_1.cpu0.o
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-objdump -d ch8_1_1.cpu0.o

ch8_1_1.cpu0.o:      file format ELF32-CPU0

Disassembly of section .text:
_Z13test_controlv:
 0: 09 dd ff d8          addiu $sp, $sp, -40
 4: 09 30 00 00          addiu $3, $zero, 0
 8: 02 3d 00 24          st   $3, 36($sp)
 c: 09 20 00 01          addiu $2, $zero, 1
10: 02 2d 00 20          st   $2, 32($sp)
14: 09 40 00 02          addiu $4, $zero, 2
18: 02 4d 00 1c          st   $4, 28($sp)
...

```

10.3 Disassembler Structure

The flow of disassembly is shown in Fig. 10.2.

- After `getInstruction()` of `Cpu0Disassembler.cpp`, `disassembleObject()` of `llvm-objdump.cpp` call `printInst()` of `Cpu0InstPrinter.cpp` to print (address: binary assembly) for the instruction, for example “(4: 09 30 00 00 addiu \$3, \$zero, 0)”.
 - `printInst()` of `Cpu0InstPrinter.cpp`: reference Fig. 3.8.
- Bytes: 4-byte (32-bits) for Cpu0. `insn`: Convert Bytes to big or little endian of 32-bit (unsigned int) binary instruction.

List `DecoderTableCpu032` and `decodeInstruction()` as follows:

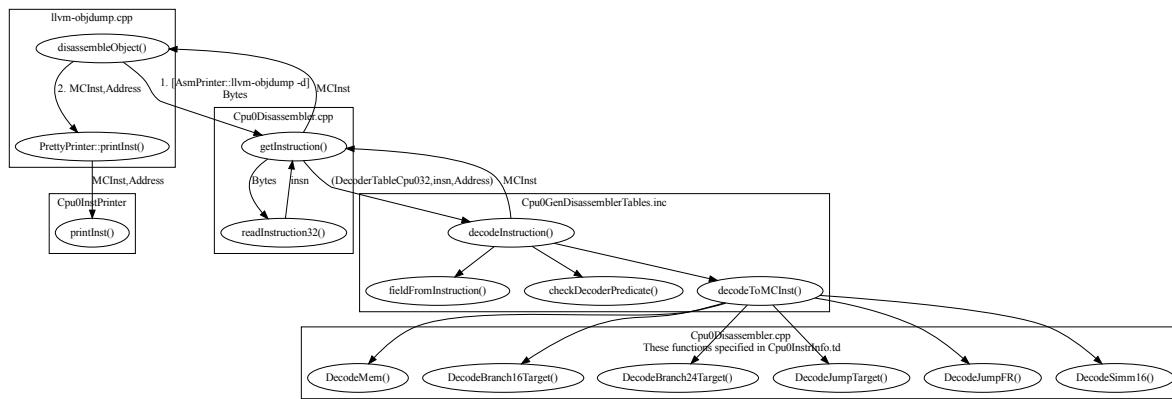


Fig. 10.2: The flow of disassembly.

build/lib/Target/Cpu0/Cpu0GenDisassemblerTables.inc

```

static const uint8_t DecoderTableCpu032[] = {
/* 0 */      MCD::OPC_ExtractField, 24, 8, // Inst{31-24} ...
/* 3 */      MCD::OPC_FilterValue, 0, 11, 0, 0, // Skip to: 19
/* 8 */      MCD::OPC_CheckField, 0, 24, 0, 149, 4, 0, // Skip to: 1188
/* 15 */     MCD::OPC_Decode, 178, 2, 0, // Opcode: NOP
/* 19 */     MCD::OPC_FilterValue, 1, 4, 0, 0, // Skip to: 28
/* 24 */     MCD::OPC_Decode, 161, 2, 1, // Opcode: LD
/* 28 */     MCD::OPC_FilterValue, 2, 4, 0, 0, // Skip to: 37
/* 33 */     MCD::OPC_Decode, 201, 2, 1, // Opcode: ST
/* 37 */     MCD::OPC_FilterValue, 3, 9, 0, 0, // Skip to: 51
/* 42 */     MCD::OPC_CheckPredicate, 0, 117, 4, 0, // Skip to: 1188
/* 47 */     MCD::OPC_Decode, 159, 2, 1, // Opcode: LB
/* 51 */     MCD::OPC_FilterValue, 4, 9, 0, 0, // Skip to: 65
/* 56 */     MCD::OPC_CheckPredicate, 0, 103, 4, 0, // Skip to: 1188
/* 61 */     MCD::OPC_Decode, 160, 2, 1, // Opcode: LBu
/* 65 */     MCD::OPC_FilterValue, 5, 9, 0, 0, // Skip to: 79
/* 70 */     MCD::OPC_CheckPredicate, 0, 89, 4, 0, // Skip to: 1188
/* 75 */     MCD::OPC_Decode, 187, 2, 1, // Opcode: SB
/* 79 */     MCD::OPC_FilterValue, 6, 9, 0, 0, // Skip to: 93
/* 84 */     MCD::OPC_CheckPredicate, 0, 75, 4, 0, // Skip to: 1188
/* 89 */     MCD::OPC_Decode, 163, 2, 1, // Opcode: LH
/* 93 */     MCD::OPC_FilterValue, 7, 9, 0, 0, // Skip to: 107
/* 98 */     MCD::OPC_CheckPredicate, 0, 61, 4, 0, // Skip to: 1188
/* 103 */    MCD::OPC_Decode, 164, 2, 1, // Opcode: LHu
/* 107 */    MCD::OPC_FilterValue, 8, 9, 0, 0, // Skip to: 121
/* 112 */    MCD::OPC_CheckPredicate, 0, 47, 4, 0, // Skip to: 1188
...
}
  
```

```

template <typename InsnType>
static DecodeStatus decodeInstruction(const uint8_t DecodeTable[], MCInst &MI,
                                      InsnType insn, uint64_t Address,
                                      const void *DisAsm,
                                      const MCSubtargetInfo &STI) {
  
```

(continues on next page)

(continued from previous page)

```

const FeatureBitset &Bits = STI.getFeatureBits();

const uint8_t *Ptr = DecodeTable;
InsnType CurFieldValue = 0;
DecodeStatus S = MCDisassembler::Success;
while (true) {
    ptrdiff_t Loc = Ptr - DecodeTable;
    switch (*Ptr) {
    default:
        errs() << Loc << ": Unexpected decode table opcode!\n";
        return MCDisassembler::Fail;
    case MCD::OPC_ExtractField: {
        unsigned Start = *++Ptr;
        unsigned Len = *++Ptr;
        ++Ptr;
        CurFieldValue = fieldFromInstruction(insn, Start, Len);
        LLVM_DEBUG(dbgs() << Loc << ": OPC_ExtractField(" << Start << ", "
            << Len << "): " << CurFieldValue << "\n");
        break;
    }
    case MCD::OPC_FilterValue: {
        // Decode the field value.
        unsigned Len;
        InsnType Val = decodeULEB128(++Ptr, &Len);
        Ptr += Len;
        // NumToSkip is a plain 24-bit integer.
        unsigned NumToSkip = *Ptr++;
        NumToSkip |= (*Ptr++) << 8;
        NumToSkip |= (*Ptr++) << 16;

        // Perform the filter operation.
        if (Val != CurFieldValue)
            Ptr += NumToSkip;
        LLVM_DEBUG(dbgs() << Loc << ": OPC_FilterValue(" << Val << ", "
            << NumToSkip << "): " << ((Val != CurFieldValue) ? "FAIL:" : "PASS:")
            << " continuing at " << (Ptr - DecodeTable) << "\n");

        break;
    }
    case MCD::OPC_CheckField: {
        unsigned Start = *++Ptr;
        unsigned Len = *++Ptr;
        InsnType FieldValue = fieldFromInstruction(insn, Start, Len);
        // Decode the field value.
        InsnType ExpectedValue = decodeULEB128(++Ptr, &Len);
        Ptr += Len;
        // NumToSkip is a plain 24-bit integer.
        unsigned NumToSkip = *Ptr++;
        NumToSkip |= (*Ptr++) << 8;
        NumToSkip |= (*Ptr++) << 16;

        // If the actual and expected values don't match, skip.
    }
}

```

(continues on next page)

(continued from previous page)

```

if (ExpectedValue != FieldValue)
    Ptr += NumToSkip;
LLVM_DEBUG(dbgs() << Loc << ": OPC_CheckField(" << Start << ", "
    << Len << ", " << ExpectedValue << ", " << NumToSkip
    << "): FieldValue = " << FieldValue << ", ExpectedValue = "
    << ExpectedValue << ":" "
    << ((ExpectedValue == FieldValue) ? "PASS\n" : "FAIL\n"));
break;
}
case MCD::OPC_CheckPredicate: {
    unsigned Len;
    // Decode the Predicate Index value.
    unsigned PIdx = decodeULEB128(++Ptr, &Len);
    Ptr += Len;
    // NumToSkip is a plain 24-bit integer.
    unsigned NumToSkip = *Ptr++;
    NumToSkip |= (*Ptr++) << 8;
    NumToSkip |= (*Ptr++) << 16;
    // Check the predicate.
    bool Pred;
    if (!Pred = checkDecoderPredicate(PIdx, Bits))
        Ptr += NumToSkip;
    (void)Pred;
    LLVM_DEBUG(dbgs() << Loc << ": OPC_CheckPredicate(" << PIdx << "): "
        << (Pred ? "PASS\n" : "FAIL\n"));

    break;
}
case MCD::OPC_Decode: {
    unsigned Len;
    // Decode the Opcode value.
    unsigned Opc = decodeULEB128(++Ptr, &Len);
    Ptr += Len;
    unsigned DecodeIdx = decodeULEB128(Ptr, &Len);
    Ptr += Len;

    MI.clear();
    MI.setOpcode(Opc);
    bool DecodeComplete;
    S = decodeToMCInst(S, DecodeIdx, insn, MI, Address, DisAsm, DecodeComplete);
    assert(DecodeComplete);

    LLVM_DEBUG(dbgs() << Loc << ": OPC_Decode: opcode " << Opc
        << ", using decoder " << DecodeIdx << ":" "
        << (S != MCDisassembler::Fail ? "PASS" : "FAIL") << "\n");
    return S;
}
case MCD::OPC_TryDecode: {
    unsigned Len;
    // Decode the Opcode value.
    unsigned Opc = decodeULEB128(++Ptr, &Len);
    Ptr += Len;
}

```

(continues on next page)

(continued from previous page)

```

unsigned DecodeIdx = decodeULEB128(Ptr, &Len);
Ptr += Len;
// NumToSkip is a plain 24-bit integer.
unsigned NumToSkip = *Ptr++;
NumToSkip |= (*Ptr++) << 8;
NumToSkip |= (*Ptr++) << 16;

// Perform the decode operation.
MCInst TmpMI;
TmpMI.setOpcode(Opc);
bool DecodeComplete;
S = decodeToMCInst(S, DecodeIdx, insn, TmpMI, Address, DisAsm, DecodeComplete);
LLVM_DEBUG(dbgs() << Loc << ": OPC_TryDecode: opcode " << Opc
<< ", using decoder " << DecodeIdx << ":" );

if (DecodeComplete) {
    // Decoding complete.
    LLVM_DEBUG(dbgs() << (S != MCDisassembler::Fail ? "PASS" : "FAIL") << "\n");
    MI = TmpMI;
    return S;
} else {
    assert(S == MCDisassembler::Fail);
    // If the decoding was incomplete, skip.
    Ptr += NumToSkip;
    LLVM_DEBUG(dbgs() << "FAIL: continuing at " << (Ptr - DecodeTable) << "\n");
    // Reset decode status. This also drops a SoftFail status that could be
    // set before the decode attempt.
    S = MCDisassembler::Success;
}
break;
}
case MCD::OPC_SoftFail: {
    // Decode the mask values.
    unsigned Len;
    InsnType PositiveMask = decodeULEB128(++Ptr, &Len);
    Ptr += Len;
    InsnType NegativeMask = decodeULEB128(Ptr, &Len);
    Ptr += Len;
    bool Fail = (insn & PositiveMask) || (~insn & NegativeMask);
    if (Fail)
        S = MCDisassembler::SoftFail;
    LLVM_DEBUG(dbgs() << Loc << ": OPC_SoftFail: " << (Fail ? "FAIL\n" : "PASS\n"));
    break;
}
case MCD::OPC_Fail: {
    LLVM_DEBUG(dbgs() << Loc << ": OPC_Fail\n");
    return MCDisassembler::Fail;
}
}
}
}
llvm_unreachable("bogosity detected in disassembler state machine!");
}

```

(continues on next page)

(continued from previous page)

List the tracing of `decodeInstruction()` by enabling “#if 1” in `Cpu0Disassembler.cpp` and running `llvm-objdump` as follows:

Ibdex/chapters/Chapter10_1/Disassembler/Cpu0Disassembler.cpp

```
#if 1
#undef LLVM_DEBUG(X)
#define LLVM_DEBUG(X) X
#endif
#include "Cpu0GenDisassemblerTables.inc"
```

```
input % ~/llvm/debug/build/bin/clang -target mips-unknown-linux-gnu -c ch3.cpp -emit-  
→ llvm -o ch3.bc  
input % ~/llvm/test/build/bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch3.  
→ bc -o ch3.cpu0.o  
input % ~/llvm/test/build/bin/llvm-objdump -d ch3.cpu0.o  
  
ch3.cpu0.o:      file format elf32-cpu0  
  
Disassembly of section .text:  
  
00000000 /* main: */  
0: OPC_ExtractField(24, 8): 9  
3: OPC_FilterValue(0, 11): FAIL: continuing at 19  
19: OPC_FilterValue(1, 4): FAIL: continuing at 28  
28: OPC_FilterValue(2, 4): FAIL: continuing at 37  
37: OPC_FilterValue(3, 9): FAIL: continuing at 51  
51: OPC_FilterValue(4, 9): FAIL: continuing at 65  
65: OPC_FilterValue(5, 9): FAIL: continuing at 79  
79: OPC_FilterValue(6, 9): FAIL: continuing at 93  
93: OPC_FilterValue(7, 9): FAIL: continuing at 107  
107: OPC_FilterValue(8, 9): FAIL: continuing at 121  
121: OPC_FilterValue(9, 4): PASS: continuing at 126  
126: OPC_Decode: opcode 264, using decoder 2: PASS  
     0: 09 dd ff f8    addiu   $sp, $sp, -8  
0: OPC_ExtractField(24, 8): 2  
3: OPC_FilterValue(0, 11): FAIL: continuing at 19  
19: OPC_FilterValue(1, 4): FAIL: continuing at 28  
28: OPC_FilterValue(2, 4): PASS: continuing at 33  
33: OPC_Decode: opcode 329, using decoder 1: PASS  
     4: 02 cd 00 04    st      $fp, 4($sp)  
0: OPC_ExtractField(24, 8): 17  
3: OPC_FilterValue(0, 11): FAIL: continuing at 19  
19: OPC_FilterValue(1, 4): FAIL: continuing at 28  
28: OPC_FilterValue(2, 4): FAIL: continuing at 37  
37: OPC_FilterValue(3, 9): FAIL: continuing at 51  
51: OPC_FilterValue(4, 9): FAIL: continuing at 65  
65: OPC_FilterValue(5, 9): FAIL: continuing at 79  
79: OPC_FilterValue(6, 9): FAIL: continuing at 93
```

(continues on next page)

(continued from previous page)

```
93: OPC_FilterValue(7, 9): FAIL: continuing at 107
107: OPC_FilterValue(8, 9): FAIL: continuing at 121
121: OPC_FilterValue(9, 4): FAIL: continuing at 130
130: OPC_FilterValue(10, 16): FAIL: continuing at 151
151: OPC_FilterValue(11, 16): FAIL: continuing at 172
172: OPC_FilterValue(12, 9): FAIL: continuing at 186
186: OPC_FilterValue(13, 9): FAIL: continuing at 200
200: OPC_FilterValue(14, 9): FAIL: continuing at 214
214: OPC_FilterValue(15, 16): FAIL: continuing at 235
235: OPC_FilterValue(17, 11): PASS: continuing at 240
240: OPC_CheckField(0, 1, 0, 941): FieldValue = 0, ExpectedValue = 0: PASS
247: OPC_Decode: opcode 265, using decoder 6: PASS
    8: 11 cd 00 00 move    $fp, $sp
0: OPC_ExtractField(24, 8): 9
3: OPC_FilterValue(0, 11): FAIL: continuing at 19
19: OPC_FilterValue(1, 4): FAIL: continuing at 28
28: OPC_FilterValue(2, 4): FAIL: continuing at 37
37: OPC_FilterValue(3, 9): FAIL: continuing at 51
51: OPC_FilterValue(4, 9): FAIL: continuing at 65
65: OPC_FilterValue(5, 9): FAIL: continuing at 79
79: OPC_FilterValue(6, 9): FAIL: continuing at 93
93: OPC_FilterValue(7, 9): FAIL: continuing at 107
107: OPC_FilterValue(8, 9): FAIL: continuing at 121
121: OPC_FilterValue(9, 4): PASS: continuing at 126
126: OPC_Decode: opcode 264, using decoder 2: PASS
    c: 09 20 00 00 addiu   $2, $zero, 0
0: OPC_ExtractField(24, 8): 2
3: OPC_FilterValue(0, 11): FAIL: continuing at 19
19: OPC_FilterValue(1, 4): FAIL: continuing at 28
28: OPC_FilterValue(2, 4): PASS: continuing at 33
33: OPC_Decode: opcode 329, using decoder 1: PASS
    10: 02 2c 00 00 st      $2, 0($fp)
0: OPC_ExtractField(24, 8): 17
3: OPC_FilterValue(0, 11): FAIL: continuing at 19
19: OPC_FilterValue(1, 4): FAIL: continuing at 28
28: OPC_FilterValue(2, 4): FAIL: continuing at 37
37: OPC_FilterValue(3, 9): FAIL: continuing at 51
51: OPC_FilterValue(4, 9): FAIL: continuing at 65
65: OPC_FilterValue(5, 9): FAIL: continuing at 79
79: OPC_FilterValue(6, 9): FAIL: continuing at 93
93: OPC_FilterValue(7, 9): FAIL: continuing at 107
107: OPC_FilterValue(8, 9): FAIL: continuing at 121
121: OPC_FilterValue(9, 4): FAIL: continuing at 130
130: OPC_FilterValue(10, 16): FAIL: continuing at 151
151: OPC_FilterValue(11, 16): FAIL: continuing at 172
172: OPC_FilterValue(12, 9): FAIL: continuing at 186
186: OPC_FilterValue(13, 9): FAIL: continuing at 200
200: OPC_FilterValue(14, 9): FAIL: continuing at 214
214: OPC_FilterValue(15, 16): FAIL: continuing at 235
235: OPC_FilterValue(17, 11): PASS: continuing at 240
240: OPC_CheckField(0, 1, 0, 941): FieldValue = 0, ExpectedValue = 0: PASS
247: OPC_Decode: opcode 265, using decoder 6: PASS
```

(continues on next page)

(continued from previous page)

```
14: 11 dc 00 00    move    $sp, $fp
0: OPC_ExtractField(24, 8): 1
3: OPC_FilterValue(0, 11): FAIL: continuing at 19
19: OPC_FilterValue(1, 4): PASS: continuing at 24
24: OPC_Decode: opcode 289, using decoder 1: PASS
    18: 01 cd 00 04    ld      $fp, 4($sp)
0: OPC_ExtractField(24, 8): 9
3: OPC_FilterValue(0, 11): FAIL: continuing at 19
19: OPC_FilterValue(1, 4): FAIL: continuing at 28
28: OPC_FilterValue(2, 4): FAIL: continuing at 37
37: OPC_FilterValue(3, 9): FAIL: continuing at 51
51: OPC_FilterValue(4, 9): FAIL: continuing at 65
65: OPC_FilterValue(5, 9): FAIL: continuing at 79
79: OPC_FilterValue(6, 9): FAIL: continuing at 93
93: OPC_FilterValue(7, 9): FAIL: continuing at 107
107: OPC_FilterValue(8, 9): FAIL: continuing at 121
121: OPC_FilterValue(9, 4): PASS: continuing at 126
126: OPC_Decode: opcode 264, using decoder 2: PASS
    1c: 09 dd 00 08    addiu   $sp, $sp, 8
0: OPC_ExtractField(24, 8): 60
3: OPC_FilterValue(0, 11): FAIL: continuing at 19
19: OPC_FilterValue(1, 4): FAIL: continuing at 28
28: OPC_FilterValue(2, 4): FAIL: continuing at 37
37: OPC_FilterValue(3, 9): FAIL: continuing at 51
51: OPC_FilterValue(4, 9): FAIL: continuing at 65
65: OPC_FilterValue(5, 9): FAIL: continuing at 79
79: OPC_FilterValue(6, 9): FAIL: continuing at 93
93: OPC_FilterValue(7, 9): FAIL: continuing at 107
107: OPC_FilterValue(8, 9): FAIL: continuing at 121
121: OPC_FilterValue(9, 4): FAIL: continuing at 130
130: OPC_FilterValue(10, 16): FAIL: continuing at 151
151: OPC_FilterValue(11, 16): FAIL: continuing at 172
172: OPC_FilterValue(12, 9): FAIL: continuing at 186
186: OPC_FilterValue(13, 9): FAIL: continuing at 200
200: OPC_FilterValue(14, 9): FAIL: continuing at 214
214: OPC_FilterValue(15, 16): FAIL: continuing at 235
235: OPC_FilterValue(17, 11): FAIL: continuing at 251
251: OPC_FilterValue(18, 11): FAIL: continuing at 267
267: OPC_FilterValue(19, 11): FAIL: continuing at 283
283: OPC_FilterValue(20, 11): FAIL: continuing at 299
299: OPC_FilterValue(21, 16): FAIL: continuing at 320
320: OPC_FilterValue(22, 16): FAIL: continuing at 341
341: OPC_FilterValue(23, 16): FAIL: continuing at 362
362: OPC_FilterValue(24, 16): FAIL: continuing at 383
383: OPC_FilterValue(25, 16): FAIL: continuing at 404
404: OPC_FilterValue(26, 16): FAIL: continuing at 425
425: OPC_FilterValue(27, 16): FAIL: continuing at 446
446: OPC_FilterValue(28, 16): FAIL: continuing at 467
467: OPC_FilterValue(29, 16): FAIL: continuing at 488
488: OPC_FilterValue(30, 16): FAIL: continuing at 509
509: OPC_FilterValue(31, 16): FAIL: continuing at 530
530: OPC_FilterValue(32, 16): FAIL: continuing at 551
```

(continues on next page)

(continued from previous page)

```

551: OPC_FilterValue(33, 16): FAIL: continuing at 572
572: OPC_FilterValue(34, 16): FAIL: continuing at 593
593: OPC_FilterValue(35, 16): FAIL: continuing at 614
614: OPC_FilterValue(36, 16): FAIL: continuing at 635
635: OPC_FilterValue(37, 16): FAIL: continuing at 656
656: OPC_FilterValue(38, 4): FAIL: continuing at 665
665: OPC_FilterValue(39, 4): FAIL: continuing at 674
674: OPC_FilterValue(40, 11): FAIL: continuing at 690
690: OPC_FilterValue(41, 11): FAIL: continuing at 706
706: OPC_FilterValue(42, 11): FAIL: continuing at 722
722: OPC_FilterValue(43, 11): FAIL: continuing at 738
738: OPC_FilterValue(48, 4): FAIL: continuing at 747
747: OPC_FilterValue(49, 4): FAIL: continuing at 756
756: OPC_FilterValue(50, 4): FAIL: continuing at 765
765: OPC_FilterValue(51, 4): FAIL: continuing at 774
774: OPC_FilterValue(52, 4): FAIL: continuing at 783
783: OPC_FilterValue(53, 4): FAIL: continuing at 792
792: OPC_FilterValue(54, 9): FAIL: continuing at 806
806: OPC_FilterValue(55, 4): FAIL: continuing at 815
815: OPC_FilterValue(56, 4): FAIL: continuing at 824
824: OPC_FilterValue(57, 23): FAIL: continuing at 852
852: OPC_FilterValue(58, 4): FAIL: continuing at 861
861: OPC_FilterValue(59, 9): FAIL: continuing at 875
875: OPC_FilterValue(60, 11): PASS: continuing at 880
880: OPC_CheckField(0, 1, 0, 301): FieldValue = 0, ExpectedValue = 0: PASS
887: OPC_Decode: opcode 285, using decoder 16: PASS
    20: 3c e0 00 00    ret      $lr
0: OPC_ExtractField(24, 8): 0
3: OPC_FilterValue(0, 11): PASS: continuing at 8
8: OPC_CheckField(0, 1, 0, 1173): FieldValue = 0, ExpectedValue = 0: PASS
15: OPC_Decode: opcode 306, using decoder 0: PASS
    24: 00 00 00 00    nop

```

Based on the debug log above, pick the example “addiu \$sp, \$sp, -8”, which has an opcode of 9, to explain *decodeInstruction()* as shown in the table and explanation below:

Table 10.1: The state transformation of decodeInstruction() for “addiu \$sp, \$sp, -8”

state	result
OPC_ExtractField	CurFieldValue <- Opcode:9
OPC_FilterValue	Match entries of DecodeTable == CurFieldValue
OPC_Decode	setOpcode(9) and decode operands by calling decodeToMCInst()

- For “move \$fp, \$sp” and “ret \$lr”, they have state OPC_CheckField before OPC_Decode since they are R type of Cpu0 instruction format and “let shamt = 0;” is set in “class ArithLogic” of Cpu0InstrInfo.td.
 - For “move \$fp, \$sp”, fieldFromInstruction(0x11cd0000, 0, 12) = (0x11cd0000 & 0x00000fff). Check bits(20..31) is 0.
- DecodeBranch16Target() and DecodeBranch24Target(): decode immediate value to MCInst.operand and set the type of MCInst.operand to immediate type, with value being either positive or negative. Operand of MCInst can be either immediate or register type.

ASSEMBLER

- *AsmParser support*
- *Assembler structure*
- *Inline assembly*

This chapter covers the assembly programming support in the Cpu0 backend.

When it comes to assembly language programming, there are two types of usage in C/C++ as follows:

ordinary assembly

```
asm("ld      $2, 8($sp)");
```

inline assembly

```
int foo = 10;
const int bar = 15;

__asm__ __volatile__("addu %0,%1,%2"
                     : "=r"(foo) // 5
                     : "r"(foo), "r"(bar)
                     );
```

In LLVM, the first method is supported by the LLVM AsmParser, and the second by the inline assembly handler.

With AsmParser and inline assembly support in the Cpu0 backend, we can hand-code assembly language in a C/C++ file and translate it into an object file (in ELF format).

11.1 AsmParser support

This section lists all the AsmParser code for the Cpu0 backend, with only brief explanations. Please refer to¹ for a more detailed explanation of AsmParser.

Running Chapter10_1/ with ch11_1.cpp will produce the following error message.

Run Chapter10_1/ with ch11_1.cpp will get the following error message.

¹ <http://www.embecosm.com/appnotes/ean10/ean10-howto-llvmas-1.0.html>

Ibdex/input/ch11_1.cpp

```
asm("ld      $2, 8($sp)");
asm("st      $0, 4($sp)");
asm("addiu $3,      $ZERO, 0");
asm("add $v1, $at, $v0");
asm("sub $3, $2, $3");
asm("mul $2, $1, $3");
asm("div $3, $2");
asm("divu $2, $3");
asm("and $2, $1, $3");
asm("or $3, $1, $2");
asm("xor $1, $2, $3");
asm("mult $4, $3");
asm("multu $3, $2");
asm("mfhi $3");
asm("mflo $2");
asm("mthi $2");
asm("mtlo $2");
asm("sra $2, $2, 2");
asm("rol $2, $1, 3");
asm("ror $3, $3, 4");
asm("shl $2, $2, 2");
asm("shr $2, $3, 5");
asm("cmp $sw, $2, $3");
asm("jeq $sw, 20");
asm("jne $sw, 16");
asm("jlt $sw, -20");
asm("jle $sw, -16");
asm("jgt $sw, -4");
asm("jge $sw, -12");
asm("jsub 0x000010000");
asm("jr $4");
asm("ret $lr");
asm("jalr $t9");
asm("li $3, 0x00700000");
asm("la $3, 0x00800000($6)");
asm("la $3, 0x00900000");
```

```
JonathanekiiMac:input Jonathan$ clang -c ch11_1.cpp -emit-llvm -o
ch11_1.bc
JonathanekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc
-o ch11_1.cpu0.o
LLVM ERROR: Inline asm not supported by this streamer because we don't have
an asm parser for this target
```

Since we haven't implemented the Cpu0 assembler, the error message shown above occurs. Cpu0 can translate LLVM IR into assembly and object files directly, but it cannot translate hand-written assembly instructions into an object file.

Chapter11_1/ includes the AsmParser implementation as follows:

Ibdex/chapters/Chapter11_1/AsmParser/Cpu0AsmParser.cpp

```
//===== Cpu0AsmParser.cpp - Parse Cpu0 assembly to MCInst instructions =====//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

#include "Cpu0.h"
#if CH >= CH11_1

#include "MCTargetDesc/Cpu0MCExpr.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/ADT/APInt.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCInstBuilder.h"
#include "llvm/MC/MCParser/MCAsmLexer.h"
#include "llvm/MC/MCParser/MCParsedAsmOperand.h"
#include "llvm/MC/MCParser/MCTargetAsmParser.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/MC/MCParser/MCAsmLexer.h"
#include "llvm/MC/MCParser/MCParsedAsmOperand.h"
#include "llvm/MC/MCValue.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/MathExtras.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-asm-parser"

namespace {
class Cpu0AssemblerOptions {
public:
    Cpu0AssemblerOptions():
        reorder(true), macro(true) {
    }

    bool isReorder() {return reorder;}
    void setReorder() {reorder = true;}
    void setNoreorder() {reorder = false;}

    bool isMacro() {return macro;}
    void setMacro() {macro = true;}
}
}
```

(continues on next page)

(continued from previous page)

```
void setNomacro() {macro = false;}

private:
    bool reorder;
    bool macro;
};

namespace {
class Cpu0AsmParser : public MCTargetAsmParser {
    MCAsmParser &Parser;
    Cpu0AssemblerOptions Options;

#define GET_ASSEMBLER_HEADER
#include "Cpu0GenAsmMatcher.inc"

    bool MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                OperandVector &Operands, MCStreamer &Out,
                                uint64_t &ErrorInfo,
                                bool MatchingInlineAsm) override;

    bool ParseRegister(unsigned &RegNo, SMLoc &StartLoc, SMLoc &EndLoc) override;

    OperandMatchResultTy tryParseRegister(unsigned &RegNo, SMLoc &StartLoc,
                                         SMLoc &EndLoc) override;

    bool ParseInstruction(ParseInstructionInfo &Info, StringRef Name,
                          SMLoc NameLoc, OperandVector &Operands) override;

    bool ParseDirective(AsmToken DirectiveID) override;

    OperandMatchResultTy parseMemOperand(OperandVector &);

    bool ParseOperand(OperandVector &Operands, StringRef Mnemonic);

    int tryParseRegister(StringRef Mnemonic);

    bool tryParseRegisterOperand(OperandVector &Operands,
                               StringRef Mnemonic);

    bool needsExpansion(MCInst &Inst);

    void expandInstruction(MCInst &Inst, SMLoc IDLoc,
                          SmallVectorImpl<MCInst> &Instructions);
    void expandLoadImm(MCInst &Inst, SMLoc IDLoc,
                       SmallVectorImpl<MCInst> &Instructions);
    void expandLoadAddressImm(MCInst &Inst, SMLoc IDLoc,
                             SmallVectorImpl<MCInst> &Instructions);
    void expandLoadAddressReg(MCInst &Inst, SMLoc IDLoc,
                             SmallVectorImpl<MCInst> &Instructions);
    bool reportParseError(StringRef ErrorMsg);
```

(continues on next page)

(continued from previous page)

```
bool parseMemOffset(const MCExpr *&Res);
bool parseRelocOperand(const MCExpr *&Res);

const MCExpr *evaluateRelocExpr(const MCExpr *Expr, StringRef RelocStr);

bool parseDirectiveSet();

bool parseSetAtDirective();
bool parseSetNoAtDirective();
bool parseSetMacroDirective();
bool parseSetNoMacroDirective();
bool parseSetReorderDirective();
bool parseSetNoReorderDirective();

int matchRegisterName(StringRef Symbol);

int matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic);

unsigned getReg(int RC, int RegNo);

public:
    Cpu0AsmParser(const MCSubtargetInfo &sti, MCAsmParser &parser,
                  const MCInstrInfo &MII, const MCTargetOptions &Options)
        : MCTargetAsmParser(Options, sti, MII), Parser(parser) {
        // Initialize the set of available features.
        setAvailableFeatures(ComputeAvailableFeatures(getSTI().getFeatureBits()));
    }

    MCAsmParser &getParser() const { return Parser; }
    MCAsmLexer &getLexer() const { return Parser.getLexer(); }

};

}

namespace {

/// Cpu0Operand - Instances of this class represent a parsed Cpu0 machine
/// instruction.
class Cpu0Operand : public MCParsedAsmOperand {

    enum KindTy {
        k_Immediate,
        k_Memory,
        k_Register,
        k_Token
    } Kind;

public:
    Cpu0Operand(KindTy K) : MCParsedAsmOperand(), Kind(K) {}

    struct Token {
```

(continues on next page)

(continued from previous page)

```
const char *Data;
unsigned Length;
};

struct PhysRegOp {
    unsigned RegNum; // Register Number
};

struct ImmOp {
    const MCExpr *Val;
};

struct MemOp {
    unsigned Base;
    const MCExpr *Off;
};

union {
    struct Token Tok;
    struct PhysRegOp Reg;
    struct ImmOp Imm;
    struct MemOp Mem;
};

SMLoc StartLoc, EndLoc;

public:
    void addRegOperands(MCInst &Inst, unsigned N) const {
        assert(N == 1 && "Invalid number of operands!");
        Inst.addOperand(MCOperand::createReg(getReg()));
    }

    void addExpr(MCInst &Inst, const MCExpr *Expr) const{
        // Add as immediate when possible. Null MCExpr = 0.
        if (Expr == 0)
            Inst.addOperand(MCOperand::createImm(0));
        else if (const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(Expr))
            Inst.addOperand(MCOperand::createImm(CE->getValue()));
        else
            Inst.addOperand(MCOperand::createExpr(Expr));
    }

    void addImmOperands(MCInst &Inst, unsigned N) const {
        assert(N == 1 && "Invalid number of operands!");
        const MCExpr *Expr = getImm();
        addExpr(Inst, Expr);
    }

    void addMemOperands(MCInst &Inst, unsigned N) const {
        assert(N == 2 && "Invalid number of operands!");

        Inst.addOperand(MCOperand::createReg(getMemBase()));

        const MCExpr *Expr = getMemOff();
        addExpr(Inst, Expr);
    }
}
```

(continues on next page)

(continued from previous page)

```

}

bool isReg() const override { return Kind == k_Register; }
bool isImm() const override { return Kind == k_Immediate; }
bool isToken() const override { return Kind == k_Token; }
bool isMem() const override { return Kind == k_Memory; }

StringRef getToken() const {
    assert(Kind == k_Token && "Invalid access!");
    return StringRef(Tok.Data, Tok.Length);
}

unsigned getReg() const override {
    assert((Kind == k_Register) && "Invalid access!");
    return Reg.RegNum;
}

const MCExpr *getImm() const {
    assert((Kind == k_Immediate) && "Invalid access!");
    return Imm.Val;
}

unsigned getMemBase() const {
    assert((Kind == k_Memory) && "Invalid access!");
    return Mem.Base;
}

const MCExpr *getMemOff() const {
    assert((Kind == k_Memory) && "Invalid access!");
    return Mem.Off;
}

static std::unique_ptr<Cpu0Operand> CreateToken(StringRef Str, SMLoc S) {
    auto Op = std::make_unique<Cpu0Operand>(k_Token);
    Op->Tok.Data = Str.data();
    Op->Tok.Length = Str.size();
    Op->StartLoc = S;
    Op->EndLoc = S;
    return Op;
}

/// Internal constructor for register kinds
static std::unique_ptr<Cpu0Operand> CreateReg(unsigned RegNum, SMLoc S,
                                               SMLoc E) {
    auto Op = std::make_unique<Cpu0Operand>(k_Register);
    Op->Reg.RegNum = RegNum;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
}

static std::unique_ptr<Cpu0Operand> CreateImm(const MCExpr *Val, SMLoc S, SMLoc E) {
}

```

(continues on next page)

(continued from previous page)

```
auto Op = std::make_unique<Cpu0Operand>(k_Immediate);
Op->Imm.Val = Val;
Op->StartLoc = S;
Op->EndLoc = E;
return Op;
}

static std::unique_ptr<Cpu0Operand> CreateMem(unsigned Base, const MCExpr *Off,
                                              SMLoc S, SMLoc E) {
    auto Op = std::make_unique<Cpu0Operand>(k_Memory);
    Op->Mem.Base = Base;
    Op->Mem.Off = Off;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
}

/// getStartLoc - Get the location of the first token of this operand.
SMLoc getStartLoc() const override { return StartLoc; }
/// getEndLoc - Get the location of the last token of this operand.
SMLoc getEndLoc() const override { return EndLoc; }

void print(raw_ostream &OS) const override {
    switch (Kind) {
    case k_Immediate:
        OS << "Imm<";
        OS << *Imm.Val;
        OS << ">";
        break;
    case k_Memory:
        OS << "Mem<";
        OS << Mem.Base;
        OS << ", ";
        OS << *Mem.Off;
        OS << ">";
        break;
    case k_Register:
        OS << "Register<" << Reg.RegNum << ">";
        break;
    case k_Token:
        OS << Tok.Data;
        break;
    }
}
};

void printCpu0Operands(OperandVector &Operands) {
    for (size_t i = 0; i < Operands.size(); i++) {
        Cpu0Operand* op = static_cast<Cpu0Operand*>(&*Operands[i]);
        assert(op != nullptr);
        LLVM_DEBUG(dbgs() << "<" << *op << ">");
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    LLVM_DEBUG(dbgs() << "\n");
}

//@1 {
bool Cpu0AsmParser::needsExpansion(MCInst &Inst) {

    switch(Inst.getOpcode()) {
        case Cpu0::LoadImm32Reg:
        case Cpu0::LoadAddr32Imm:
        case Cpu0::LoadAddr32Reg:
            return true;
        default:
            return false;
    }
}

void Cpu0AsmParser::expandInstruction(MCInst &Inst, SMLoc IDLoc,
                                      SmallVectorImpl<MCInst> &Instructions) {
    switch(Inst.getOpcode()) {
        case Cpu0::LoadImm32Reg:
            return expandLoadImm(Inst, IDLoc, Instructions);
        case Cpu0::LoadAddr32Imm:
            return expandLoadAddressImm(Inst, IDLoc, Instructions);
        case Cpu0::LoadAddr32Reg:
            return expandLoadAddressReg(Inst, IDLoc, Instructions);
    }
}
//@1 }

void Cpu0AsmParser::expandLoadImm(MCInst &Inst, SMLoc IDLoc,
                                  SmallVectorImpl<MCInst> &Instructions) {
    MCInst tmpInst;
    const MCOperand &ImmOp = Inst.getOperand(1);
    assert(ImmOp.isImm() && "expected immediate operand kind");
    const MCOperand &RegOp = Inst.getOperand(0);
    assert(RegOp.isReg() && "expected register operand kind");

    int ImmValue = ImmOp.getImm();
    tmpInst.setLoc(IDLoc);
    if (0 <= ImmValue && ImmValue <= 65535) {
        // for 0 <= j <= 65535.
        // li d,j => ori d,$zero,j
        tmpInst.setOpcode(Cpu0::ORI);
        tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
        tmpInst.addOperand(
            MCOperand::createReg(Cpu0::ZERO));
        tmpInst.addOperand(MCOperand::createImm(ImmValue));
        Instructions.push_back(tmpInst);
    } else if (ImmValue < 0 && ImmValue >= -32768) {
        // for -32768 <= j < 0.
        // li d,j => addiu d,$zero,j
    }
}

```

(continues on next page)

(continued from previous page)

```

tmpInst.setOpcode(Cpu0::ADDiu); //TODO: no ADDiu64 in td files?
tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
tmpInst.addOperand(
    MCOperand::createReg(Cpu0::ZERO));
tmpInst.addOperand(MCOperand::createImm(ImmValue));
Instructions.push_back(tmpInst);
} else {
    // for any other value of j that is representable as a 32-bit integer.
    // li d,j => lui d,hi16(j)
    //             ori d,d,lo16(j)
    tmpInst.setOpcode(Cpu0::LUI);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm((ImmValue & 0xffff0000) >> 16));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
    tmpInst.setOpcode(Cpu0::ORi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue & 0xffff));
    tmpInst.setLoc(IDLoc);
    Instructions.push_back(tmpInst);
}
}

void Cpu0AsmParser::expandLoadAddressReg(MCInst &Inst, SMLoc IDLoc,
                                         SmallVectorImpl<MCInst> &Instructions) {
MCInst tmpInst;
const MCOperand &ImmOp = Inst.getOperand(2);
assert(ImmOp.isImm() && "expected immediate operand kind");
const MCOperand &SrcRegOp = Inst.getOperand(1);
assert(SrcRegOp.isReg() && "expected register operand kind");
const MCOperand &DstRegOp = Inst.getOperand(0);
assert(DstRegOp.isReg() && "expected register operand kind");
int ImmValue = ImmOp.getImm();
if ( -32768 <= ImmValue && ImmValue <= 32767) {
    // for -32768 <= j < 32767.
    // la d,j(s) => addiu d,s,j
    tmpInst.setOpcode(Cpu0::ADDiu); //TODO: no ADDiu64 in td files?
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(SrcRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
} else {
    // for any other value of j that is representable as a 32-bit integer.
    // la d,j(s) => lui d,hi16(j)
    //             ori d,d,lo16(j)
    //             add d,d,s
    tmpInst.setOpcode(Cpu0::LUI);
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm((ImmValue & 0xffff0000) >> 16));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
}

```

(continues on next page)

(continued from previous page)

```

tmpInst.setOpcode(Cpu0::ORi);
tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::createImm(ImmValue & 0xffff));
Instructions.push_back(tmpInst);
tmpInst.clear();
tmpInst.setOpcode(Cpu0::ADD);
tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::createReg(SrcRegOp.getReg()));
Instructions.push_back(tmpInst);
}
}

void Cpu0AsmParser::expandLoadAddressImm(MCInst &Inst, SMLoc IDLoc,
                                         SmallVectorImpl<MCInst> &Instructions) {
MCInst tmpInst;
const MCOperand &ImmOp = Inst.getOperand(1);
assert(ImmOp.isImm() && "expected immediate operand kind");
const MCOperand &RegOp = Inst.getOperand(0);
assert(RegOp.isReg() && "expected register operand kind");
int ImmValue = ImmOp.getImm();
if (-32768 <= ImmValue && ImmValue <= 32767) {
    // for -32768 <= j < 32767.
    // la d,j => addiu d,$zero,j
    tmpInst.setOpcode(Cpu0::ADDiu);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(
        MCOperand::createReg(Cpu0::ZERO));
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
} else {
    // for any other value of j that is representable as a 32-bit integer.
    // la d,j => lui d,hi16(j)
    //           ori d,d,lo16(j)
    tmpInst.setOpcode(Cpu0::LUI);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm((ImmValue & 0xffff0000) >> 16));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
    tmpInst.setOpcode(Cpu0::ORi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue & 0xffff));
    Instructions.push_back(tmpInst);
}
}

//@2 {
bool Cpu0AsmParser::MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                             OperandVector &Operands,
                                             MCStreamer &Out,

```

(continues on next page)

```
        uint64_t &ErrorInfo,
        bool MatchingInlineAsm) {

printCpu0Operands(Operands);
MCInst Inst;
unsigned MatchResult = MatchInstructionImpl(Operands, Inst, ErrorInfo,
                                             MatchingInlineAsm);
switch (MatchResult) {
default: break;
case Match_Success: {
    if (needsExpansion(Inst)) {
        SmallVector<MCInst, 4> Instructions;
        expandInstruction(Inst, IDLoc, Instructions);
        for(unsigned i =0; i < Instructions.size(); i++){
            Out.emitInstruction(Instructions[i], getSTI());
        }
    } else {
        Inst.setLoc(IDLoc);
        Out.emitInstruction(Inst, getSTI());
    }
    return false;
}
//@2 }
case Match_MissingFeature:
    Error(IDLoc, "instruction requires a CPU feature not currently enabled");
    return true;
case Match_InvalidOperand: {
    SMLoc ErrorLoc = IDLoc;
    if (ErrorInfo != ~0U) {
        if (ErrorInfo >= Operands.size())
            return Error(IDLoc, "too few operands for instruction");

        ErrorLoc = ((Cpu0Operand &)*Operands[ErrorInfo]).getStartLoc();
        if (ErrorLoc == SMLoc()) ErrorLoc = IDLoc;
    }

    return Error(ErrorLoc, "invalid operand for instruction");
}
case Match_MnemonicFail:
    return Error(IDLoc, "invalid instruction");
}
return true;
}

int Cpu0AsmParser::matchRegisterName(StringRef Name) {

int CC;
CC = StringSwitch<unsigned>(Name)
.Case("zero", Cpu0::ZERO)
.Case("at", Cpu0::AT)
.Case("v0", Cpu0::V0)
.Case("v1", Cpu0::V1)
.Case("a0", Cpu0::A0)
.Case("a1", Cpu0::A1)
```

(continues on next page)

(continued from previous page)

```

.Case("t9",    Cpu0::T9)
.Case("t0",    Cpu0::T0)
.Case("t1",    Cpu0::T1)
.Case("s0",    Cpu0::S0)
.Case("s1",    Cpu0::S1)
.Case("sw",    Cpu0::SW)
.Case("gp",    Cpu0::GP)
.Case("fp",    Cpu0::FP)
.Case("sp",    Cpu0::SP)
.Case("lr",    Cpu0::LR)
.Case("pc",    Cpu0::PC)
.Case("hi",    Cpu0::HI)
.Case("lo",    Cpu0::LO)
.Case("epc",   Cpu0::EPC)
.Default(-1);

if (CC != -1)
    return CC;

return -1;
}

unsigned Cpu0AsmParser::getReg(int RC,int RegNo) {
    return *(getContext().getRegisterInfo()->getRegClass(RC).begin() + RegNo);
}

int Cpu0AsmParser::matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic) {
    if (RegNum > 15)
        return -1;

    return getReg(Cpu0::CPUREgsRegClassID, RegNum);
}

int Cpu0AsmParser::tryParseRegister(StringRef Mnemonic) {
    const AsmToken &Tok = Parser.getTok();
    int RegNum = -1;

    if (Tok.is(AsmToken::Identifier)) {
        std::string lowerCase = Tok.getString().lower();
        RegNum = matchRegisterName(lowerCase);
    } else if (Tok.is(AsmToken::Integer))
        RegNum = matchRegisterByNumber(static_cast<unsigned>(Tok.getIntVal()),
                                       Mnemonic.lower());
    else
        return RegNum; //error
    return RegNum;
}

bool Cpu0AsmParser::
tryParseRegisterOperand(OperandVector &Operands,
                      StringRef Mnemonic){

```

(continues on next page)

(continued from previous page)

```
SMLoc S = Parser.getTok().getLoc();
int RegNo = -1;

    RegNo = tryParseRegister(Mnemonic);
if (RegNo == -1)
    return true;

Operands.push_back(Cpu0Operand::CreateReg(RegNo, S,
    Parser.getTok().getLoc()));
Parser.Lex(); // Eat register token.
return false;
}

bool Cpu0AsmParser::ParseOperand(OperandVector &Operands,
                               StringRef Mnemonic) {
LLVM_DEBUG(dbgs() << "ParseOperand\n");
// Check if the current operand has a custom associated parser, if so, try to
// custom parse the operand, or fallback to the general approach.
OperandMatchResultTy ResTy = MatchOperandParserImpl(Operands, Mnemonic);
if (ResTy == MatchOperand_Success)
    return false;
// If there wasn't a custom match, try the generic matcher below. Otherwise,
// there was a match, but an error occurred, in which case, just return that
// the operand parsing failed.
if (ResTy == MatchOperand_ParseFail)
    return true;

LLVM_DEBUG(dbgs() << "... Generic Parser\n");

switch (getLexer().getKind()) {
default:
    Error(Parser.getTok().getLoc(), "unexpected token in operand");
    return true;
case AsmToken::Dollar:
    // parse register
    SMLoc S = Parser.getTok().getLoc();
    Parser.Lex(); // Eat dollar token.
    // parse register operand
    if (!tryParseRegisterOperand(Operands, Mnemonic)) {
        if (getLexer().is(AsmToken::LParen)) {
            // check if it is indexed addressing operand
            Operands.push_back(Cpu0Operand::CreateToken("(", S));
            Parser.Lex(); // eat parenthesis
            if (getLexer().isNot(AsmToken::Dollar))
                return true;

            Parser.Lex(); // eat dollar
            if (tryParseRegisterOperand(Operands, Mnemonic))
                return true;

            if (!getLexer().is(AsmToken::RParen))
                return true;
    }
}
```

(continues on next page)

(continued from previous page)

```

S = Parser.getTok().getLoc();
Operands.push_back(Cpu0Operand::CreateToken(")", S));
Parser.Lex();
}
return false;
}
// maybe it is a symbol reference
StringRef Identifier;
if (Parser.parseIdentifier(Identifier))
    return true;

SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

MCSymbol *Sym = getContext().getOrCreateSymbol("$" + Identifier);

// Otherwise create a symbol ref.
const MCExpr *Res = MCSymbolRefExpr::create(Sym, MCSymbolRefExpr::VK_None,
                                              getContext());

Operands.push_back(Cpu0Operand::CreateImm(Res, S, E));
return false;
}
case AsmToken::Identifier:
case AsmToken::LParen:
case AsmToken::Minus:
case AsmToken::Plus:
case AsmToken::Integer:
case AsmToken::String: {
    // quoted label names
    const MCExpr *IdVal;
    SMLoc S = Parser.getTok().getLoc();
    if (getParser().parseExpression(IdVal))
        return true;
    SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);
    Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
    return false;
}
case AsmToken::Percent: {
    // it is a symbol reference or constant expression
    const MCExpr *IdVal;
    SMLoc S = Parser.getTok().getLoc(); // start location of the operand
    if (parseRelocOperand(IdVal))
        return true;

    SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

    Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
    return false;
} // case AsmToken::Percent
} // switch(getLexer().getKind())
return true;

```

(continues on next page)

(continued from previous page)

```

}

const MCExpr *Cpu0AsmParser::evaluateRelocExpr(const MCExpr *Expr,
                                               StringRef RelocStr) {
    Cpu0MCExpr::Cpu0ExprKind Kind =
        StringSwitch<Cpu0MCExpr::Cpu0ExprKind>(RelocStr)
            .Case("call16", Cpu0MCExpr::CEK_GOT_CALL)
            .Case("call_hi", Cpu0MCExpr::CEK_CALL_HI16)
            .Case("call_lo", Cpu0MCExpr::CEK_CALL_LO16)
            .Case("dtp_hi", Cpu0MCExpr::CEK_DTP_HI)
            .Case("dtp_lo", Cpu0MCExpr::CEK_DTP_LO)
            .Case("got", Cpu0MCExpr::CEK_GOT)
            .Case("got_hi", Cpu0MCExpr::CEK_GOT_HI16)
            .Case("got_lo", Cpu0MCExpr::CEK_GOT_LO16)
            .Case("gottprel", Cpu0MCExpr::CEK_GOTTPREL)
            .Case("gp_rel", Cpu0MCExpr::CEK_GPREL)
            .Case("hi", Cpu0MCExpr::CEK_ABS_HI)
            .Case("lo", Cpu0MCExpr::CEK_ABS_LO)
            .Case("tlsldm", Cpu0MCExpr::CEK_TLSLDM)
            .Case("tp_hi", Cpu0MCExpr::CEK_TP_HI)
            .Case("tp_lo", Cpu0MCExpr::CEK_TP_LO)
            .Default(Cpu0MCExpr::CEK_None);

    assert(Kind != Cpu0MCExpr::CEK_None);
    return Cpu0MCExpr::create(Kind, Expr, getContext());
}

bool Cpu0AsmParser::parseRelocOperand(const MCExpr *&Res) {

    Parser.Lex(); // eat % token
    const AsmToken &Tok = Parser.getTok(); // get next token, operation
    if (Tok.isNot(AsmToken::Identifier))
        return true;

    std::string Str = Tok.getIdentifier().str();

    Parser.Lex(); // eat identifier
    // now make expression from the rest of the operand
    const MCExpr *IdVal;
    SMLoc EndLoc;

    if (getLexer().getKind() == AsmToken::LParen) {
        while (1) {
            Parser.Lex(); // eat '(' token
            if (getLexer().getKind() == AsmToken::Percent) {
                Parser.Lex(); // eat % token
                const AsmToken &nextTok = Parser.getTok();
                if (nextTok.isNot(AsmToken::Identifier))
                    return true;
                Str += "%";
                Str += nextTok.getIdentifier();
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

Parser.Lex(); // eat identifier
if (getLexer().getKind() != AsmToken::LParen)
    return true;
} else
    break;
}
if (getParser().parseParenExpression(IdVal, EndLoc))
    return true;

while (getLexer().getKind() == AsmToken::RParen)
    Parser.Lex(); // eat ')' token

} else
    return true; // parenthesis must follow reloc operand

Res = evaluateRelocExpr(IdVal, Str);
return false;
}

bool Cpu0AsmParser::ParseRegister(unsigned &RegNo, SMLoc &StartLoc,
                                  SMLoc &EndLoc) {

StartLoc = Parser.getTok().getLoc();
RegNo = tryParseRegister("");
EndLoc = Parser.getTok().getLoc();
return (RegNo == (unsigned)-1);
}

OperandMatchResultTy Cpu0AsmParser::tryParseRegister(unsigned &RegNo,
                                                    SMLoc &StartLoc,
                                                    SMLoc &EndLoc) {
StartLoc = Parser.getTok().getLoc();
RegNo = tryParseRegister("");
EndLoc = Parser.getTok().getLoc();
return (RegNo == (unsigned)-1) ? MatchOperand_NoMatch
                               : MatchOperand_Success;
}

bool Cpu0AsmParser::parseMemOffset(const MCExpr *&Res) {
switch(getLexer().getKind()) {
default:
    return true;
case AsmToken::Integer:
case AsmToken::Minus:
case AsmToken::Plus:
    return (getParser().parseExpression(Res));
case AsmToken::Percent:
    return parseRelocOperand(Res);
case AsmToken::LParen:
    return false; // it's probably assuming 0
}
return true;
}

```

(continues on next page)

(continued from previous page)

```
}

// eg, 12($sp) or 12(la)
OperandMatchResultTy Cpu0AsmParser::parseMemOperand(
    OperandVector &Operands) {

    const MCExpr *IdVal = 0;
    SMLoc S;
    // first operand is the offset
    S = Parser.getTok().getLoc();

    if (parseMemOffset(IdVal))
        return MatchOperand_ParseFail;

    const AsmToken &Tok = Parser.getTok(); // get next token
    if (Tok.isNot(AsmToken::LParen)) {
        Cpu0Operand &Mnemonic = static_cast<Cpu0Operand &>(*Operands[0]);
        if (Mnemonic.getToken() == "la") {
            SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);
            Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
            return MatchOperand_Success;
        }
        Error(Parser.getTok().getLoc(), "'(' expected");
        return MatchOperand_ParseFail;
    }

    Parser.Lex(); // Eat '(' token.

    const AsmToken &Tok1 = Parser.getTok(); // get next token
    if (Tok1.is(AsmToken::Dollar)) {
        Parser.Lex(); // Eat '$' token.
        if (tryParseRegisterOperand(Operands, ""))
            Error(Parser.getTok().getLoc(), "unexpected token in operand");
        return MatchOperand_ParseFail;
    }

    } else {
        Error(Parser.getTok().getLoc(), "unexpected token in operand");
        return MatchOperand_ParseFail;
    }

    const AsmToken &Tok2 = Parser.getTok(); // get next token
    if (Tok2.isNot(AsmToken::RParen)) {
        Error(Parser.getTok().getLoc(), "')' expected");
        return MatchOperand_ParseFail;
    }

    SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

    Parser.Lex(); // Eat ')' token.

    if (!IdVal)
```

(continues on next page)

(continued from previous page)

```

IdVal = MCConstantExpr::create(0, getContext());

// Replace the register operand with the memory operand.
std::unique_ptr<Cpu0Operand> op(
    static_cast<Cpu0Operand *>(Operands.back().release()));
int RegNo = op->getReg();
// remove register from operands
Operands.pop_back();
// and add memory operand
Operands.push_back(Cpu0Operand::CreateMem(RegNo, IdVal, S, E));
return MatchOperand_Success;
}

bool Cpu0AsmParser::
ParseInstruction(ParseInstructionInfo &Info, StringRef Name, SMLoc NameLoc,
                 OperandVector &Operands) {

    // Create the leading tokens for the mnemonic, split by '.' characters.
    size_t Start = 0, Next = Name.find('.');
    StringRef Mnemonic = Name.slice(Start, Next);
    // Refer to the explanation in source code of function DecodeJumpFR(...) in
    // Cpu0Disassembler.cpp
    if (Mnemonic == "ret")
        Mnemonic = "jr";

    Operands.push_back(Cpu0Operand::CreateToken(Mnemonic, NameLoc));

    // Read the remaining operands.
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        // Read the first operand.
        if (ParseOperand(Operands, Name)) {
            SMLoc Loc = getLexer().getLoc();
            Parser.eatToEndOfStatement();
            return Error(Loc, "unexpected token in argument list");
        }

        while (getLexer().is(AsmToken::Comma) ) {
            Parser.Lex(); // Eat the comma.

            // Parse and remember the operand.
            if (ParseOperand(Operands, Name)) {
                SMLoc Loc = getLexer().getLoc();
                Parser.eatToEndOfStatement();
                return Error(Loc, "unexpected token in argument list");
            }
        }
    }

    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        SMLoc Loc = getLexer().getLoc();
        Parser.eatToEndOfStatement();
        return Error(Loc, "unexpected token in argument list");
    }
}

```

(continues on next page)

(continued from previous page)

```
}

Parser.Lex(); // Consume the EndOfStatement
return false;
}

bool Cpu0AsmParser::reportParseError(StringRef ErrorMsg) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, ErrorMsg);
}

bool Cpu0AsmParser::parseSetReorderDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        reportParseError("unexpected token in statement");
        return false;
    }
    Options.setReorder();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetNoReorderDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        reportParseError("unexpected token in statement");
        return false;
    }
    Options.setNoreorder();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetMacroDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        reportParseError("unexpected token in statement");
        return false;
    }
    Options.setMacro();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetNoMacroDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
```

(continues on next page)

(continued from previous page)

```
reportParseError("`noreorder' must be set before `nomacro'");  
    return false;  
}  
if (Options.isReorder()) {  
    reportParseError("`noreorder' must be set before `nomacro'");  
    return false;  
}  
Options.setNomacro();  
Parser.Lex(); // Consume the EndOfStatement  
return false;  
}  
bool Cpu0AsmParser::parseDirectiveSet() {  
  
    // get next token  
    const AsmToken &Tok = Parser.getTok();  
  
    if (Tok.getString() == "reorder") {  
        return parseSetReorderDirective();  
    } else if (Tok.getString() == "noreorder") {  
        return parseSetNoReorderDirective();  
    } else if (Tok.getString() == "macro") {  
        return parseSetMacroDirective();  
    } else if (Tok.getString() == "nomacro") {  
        return parseSetNoMacroDirective();  
    }  
    return true;  
}  
  
bool Cpu0AsmParser::ParseDirective(AsmToken DirectiveID) {  
  
    if (DirectiveID.getString() == ".ent") {  
        // ignore this directive for now  
        Parser.Lex();  
        return false;  
    }  
  
    if (DirectiveID.getString() == ".end") {  
        // ignore this directive for now  
        Parser.Lex();  
        return false;  
    }  
  
    if (DirectiveID.getString() == ".frame") {  
        // ignore this directive for now  
        Parser.eatToEndOfStatement();  
        return false;  
    }  
  
    if (DirectiveID.getString() == ".set") {  
        return parseDirectiveSet();  
    }  
}
```

(continues on next page)

(continued from previous page)

```
if (DirectiveID.getString() == ".fmask") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
}

if (DirectiveID.getString() == ".mask") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
}

if (DirectiveID.getString() == ".gpword") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
}

return true;
}

extern "C" void LLVMInitializeCpu0AsmParser() {
    RegisterMCAsmParser<Cpu0AsmParser> X(TheCpu0Target);
    RegisterMCAsmParser<Cpu0AsmParser> Y(TheCpu0elTarget);
}

#define GET_REGISTER_MATCHER
#define GET_MATCHER_IMPLEMENTATION
#include "Cpu0GenAsmMatcher.inc"

#else // #if CH >= CH11_1
extern "C" void LLVMInitializeCpu0AsmParser() {}
#endif
```

Index/chapters/Chapter11_1/AsmParser/CMakeLists.txt

```
add_llvm_component_library(LLVMCpu0AsmParser
    Cpu0AsmParser.cpp

    LINK_COMPONENTS
        MC
        MCParse
        Cpu0Desc
        Cpu0Info
        Support

    ADD_TO_COMPONENT
        Cpu0
    )
```

The *Cpu0AsmParser.cpp* file contains around one thousand lines of code that handle assembly language parsing. With a little patience, you can understand it.

To enable building the files in the AsmParser directory, modify *CMakeLists.txt* as follows:

Ibdex/chapters/Chapter11_1/CMakeLists.txt

```
set(LLVM_TARGET_DEFINITIONS Cpu0Asm.td)
tablegen(LLVM Cpu0GenAsmMatcher.inc -gen-asn-matcher)

Cpu0AsmParser

add_subdirectory(AsmParser)
```

Ibdex/chapters/Chapter11_1/Cpu0Asm.td

```
//===== Cpu0Asm.td - Describe the Cpu0 Target Machine -----*- tablegen -*=====//
//                                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====-----=====
// This is the top level entry point for the Cpu0 target.
//=====-----=====

//=====-----=====
// Target-independent interfaces
//=====-----=====

include "llvm/Target/Target.td"

//=====-----=====
// Target-dependent interfaces
//=====-----=====

include "Cpu0RegisterInfo.td"
include "Cpu0RegisterInfoGPROutForAsm.td"
include "Cpu0.td"
```

Ibdex/chapters/Chapter11_1/Cpu0RegisterInfoGPROutForAsm.td

```
//=====-----///
// Register Classes
//=====-----///

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add CPURegs)>;
```

The *CMakeLists.txt* modification shown above generates *Cpu0GenAsmMatcher.inc*, which is used by *Cpu0AsmParser.cpp*.

Cpu0Asm.td includes *Cpu0RegisterInfoGPROutForAsm.td*, which defines *GPROut* as mapping to *CPURegs*. Meanwhile, *Cpu0Other.td* includes *Cpu0RegisterInfoGPROutForOther.td*, which defines *GPROut* to map to *CPURegs* excluding *SW*.

Cpu0Other.td is used when translating LLVM IR to Cpu0 instructions. In this case, the *SW* register is reserved for storing CPU status and must not be allocated as a general-purpose register.

For example, if *GPROut* includes *SW*, compiling the C statement *a = (b & c);* might generate the instruction *and \$sw, \$1, \$2.* This would overwrite the interrupt status in *\$sw.*

However, when programming in assembly, instructions like *andi \$sw, \$sw, 0xffff* are allowed. This kind of assembly is accepted, and the Cpu0 backend considers it safe. An assembler programmer may use *andi \$sw, \$sw, 0xffff* to disable trace debug messages, and *ori \$sw, \$sw, 0x0020* to enable them.

Additionally, interrupt bits can also be enabled or disabled using *ori* and *andi* instructions.

The *EPC* must be set to *CPURegs* as shown below. Otherwise, *MatchInstructionImpl()* in *MatchAndEmitInstruction()* will fail for the instruction *asm("mfco \$pc, \$epc");*

Ibdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
def CPUREgs : RegisterClass<"Cpu0", [i32], 32, (add
...
, PC, EPC)>;
```

Ibdex/chapters/Chapter11_1/Cpu0.td

```
def Cpu0AsmParser : AsmParser {
    let ShouldEmitMatchRegisterName = 0;
}

def Cpu0AsmParserVariant : AsmParserVariant {
    int Variant = 0;

    // Recognize hard coded registers.
    string RegisterPrefix = "$";
}
```

```
def Cpu0 : Target {
    ...
}
```

```
    let AssemblyParsers = [Cpu0AsmParser];
    let AssemblyParserVariants = [Cpu0AsmParserVariant];
```

```
}
```

Ibdex/chapters/Chapter11_1/Cpu0InstrFormats.td

```
// Pseudo-instructions for alternate assembly syntax (never used by codegen).
// These are aliases that require C++ handling to convert to the target
// instruction, while InstAliases can be handled directly by tblgen.
class Cpu0AsmPseudoInst<dag outs, dag ins, string asmstr>:
    Cpu0Inst<outs, ins, asmstr, [], IIPseudo, Pseudo> {
        let isPseudo = 1;
        let Pattern = [];
    }
```

Ibdex/chapters/Chapter11_1/Cpu0InstrInfo.td

```
def Cpu0MemAsmOperand : AsmOperandClass {
    let Name = "Mem";
    let ParserMethod = "parseMemOperand";
}

// Address operand
def mem : Operand<i32> {
    ...
    let ParserMatchClass = Cpu0MemAsmOperand;
}
...
```

```
//=====
// Pseudo Instruction definition
=====

let Predicates = [Ch11_1] in {
class LoadImm32< string instr_asm, Operand Od, RegisterClass RC> :
    Cpu0AsmPseudoInst<(outs RC:$ra), (ins Od:$imm32),
        !strconcat(instr_asm, "\t$ra, $imm32")> ;
def LoadImm32Reg : LoadImm32<"li", shamt, GPROut>;

class LoadAddress<string instr_asm, Operand MemOpnd, RegisterClass RC> :
    Cpu0AsmPseudoInst<(outs RC:$ra), (ins MemOpnd:$addr),
        !strconcat(instr_asm, "\t$ra, $addr")> ;
def LoadAddr32Reg : LoadAddress<"la", mem, GPROut>;

class LoadAddressImm<string instr_asm, Operand Od, RegisterClass RC> :
    Cpu0AsmPseudoInst<(outs RC:$ra), (ins Od:$imm32),
        !strconcat(instr_asm, "\t$ra, $imm32")> ;
def LoadAddr32Imm : LoadAddressImm<"la", shamt, GPROut>;
}
```

In the *Cpu0InstrInfo.td* file, the directive **let ParserMethod = “parseMemOperand”** declares that the method *parseMemOperand()* will be used to handle the **mem** operand in Cpu0 instructions such as *ld* and *st*.

For example, in the instruction *ld \$2, 4(\$sp)*, the **mem** operand is *4(\$sp)*.

Together with the directive **let ParserMatchClass = Cpu0MemAsmOperand;**, LLVM will invoke the *parseMemOperand()* function in *Cpu0AsmParser.cpp* whenever it encounters a **mem** operand like *4(\$sp)* in assembly code.

With the above **let** assignments, TableGen will generate the corresponding structure and functions in *Cpu0GenAsmMatcher.inc*.

build/lib/Target/Cpu0/Cpu0GenAsmMatcher.inc

```
enum OperandMatchResultTy {
    MatchOperand_Success,      // operand matched successfully
    MatchOperand_NoMatch,     // operand did not match
    MatchOperand_ParseFail    // operand matched but had errors
};
OperandMatchResultTy MatchOperandParserImpl(
```

(continues on next page)

(continued from previous page)

```

OperandVector &Operands,
StringRef Mnemonic);
OperandMatchResultTy tryCustomParseOperand(
    OperandVector &Operands,
    unsigned MCK);
...

Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::
tryCustomParseOperand(OperandVector &Operands,
    unsigned MCK) {

    switch(MCK) {
        case MCK_Mem:
            return parseMemOperand(Operands);
        default:
            return MatchOperand_NoMatch;
    }
    return MatchOperand_NoMatch;
}

Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::
MatchOperandParserImpl(OperandVector &Operands,
    StringRef Mnemonic) {
    ...
}

/// MatchClassKind - The kinds of classes which participate in
/// instruction matching.
enum MatchClassKind {
    ...
    MCK_Mem, // user defined class 'Cpu0MemAsmOperand'
    ...
};

```

The three pseudo instruction definitions in *Cpu0InstrInfo.td*, such as *LoadImm32Reg*, are handled by *Cpu0AsmParser.cpp* as follows:

Ibdex/chapters/Chapter11_1/AsmParser/Cpu0AsmParser.cpp

```

bool Cpu0AsmParser::needsExpansion(MCInst &Inst) {

    switch(Inst.getOpcode()) {
        case Cpu0::LoadImm32Reg:
        case Cpu0::LoadAddr32Imm:
        case Cpu0::LoadAddr32Reg:
            return true;
        default:
            return false;
    }
}

void Cpu0AsmParser::expandInstruction(MCInst &Inst, SMLoc IDLoc,
    SmallVectorImpl<MCInst> &Instructions) {

```

(continues on next page)

(continued from previous page)

```

switch(Inst.getOpcode()) {
    case Cpu0::LoadImm32Reg:
        return expandLoadImm(Inst, IDLoc, Instructions);
    case Cpu0::LoadAddr32Imm:
        return expandLoadAddressImm(Inst, IDLoc, Instructions);
    case Cpu0::LoadAddr32Reg:
        return expandLoadAddressReg(Inst, IDLoc, Instructions);
}
}
}

```

```

bool Cpu0AsmParser::MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                            OperandVector &Operands,
                                            MCStreamer &Out,
                                            uint64_t &ErrorInfo,
                                            bool MatchingInlineAsm) {
    printCpu0Operands(Operands);
    MCInst Inst;
    unsigned MatchResult = MatchInstructionImpl(Operands, Inst, ErrorInfo,
                                                MatchingInlineAsm);
    switch (MatchResult) {
    default: break;
    case Match_Success: {
        if (needsExpansion(Inst)) {
            SmallVector<MCInst, 4> Instructions;
            expandInstruction(Inst, IDLoc, Instructions);
            for (unsigned i = 0; i < Instructions.size(); i++) {
                Out.emitInstruction(Instructions[i], getSTI());
            }
        } else {
            Inst.setLoc(IDLoc);
            Out.emitInstruction(Inst, getSTI());
        }
        return false;
    }
}
}

```

```

...
}
}

```

Finally, remember that the *CPURegs* list below must follow the order of register numbers, because the AsmParser uses this order when encoding register numbers.

Ibdex/chapters/Chapter11_1/Cpu0RegisterInfo.td

```

//=====
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"
// @ All registers definition
let Namespace = "Cpu0" in {
    // @ General Purpose Registers
    def ZERO : Cpu0GPRReg<0, "zero">, DwarfRegNum<[0]>;
    def AT : Cpu0GPRReg<1, "1">, DwarfRegNum<[1]>;
    def V0 : Cpu0GPRReg<2, "2">, DwarfRegNum<[2]>;
}

```

(continues on next page)

(continued from previous page)

```

def V1 : Cpu0GPRReg<3, "3">, DwarfRegNum<[3]>;
def A0 : Cpu0GPRReg<4, "4">, DwarfRegNum<[4]>;
def A1 : Cpu0GPRReg<5, "5">, DwarfRegNum<[5]>;
def T9 : Cpu0GPRReg<6, "t9">, DwarfRegNum<[6]>;
def T0 : Cpu0GPRReg<7, "7">, DwarfRegNum<[7]>;
def T1 : Cpu0GPRReg<8, "8">, DwarfRegNum<[8]>;
def S0 : Cpu0GPRReg<9, "9">, DwarfRegNum<[9]>;
def S1 : Cpu0GPRReg<10, "10">, DwarfRegNum<[10]>;
def GP : Cpu0GPRReg<11, "gp">, DwarfRegNum<[11]>;
def FP : Cpu0GPRReg<12, "fp">, DwarfRegNum<[12]>;
def SP : Cpu0GPRReg<13, "sp">, DwarfRegNum<[13]>;
def LR : Cpu0GPRReg<14, "lr">, DwarfRegNum<[14]>;
def SW : Cpu0GPRReg<15, "sw">, DwarfRegNum<[15]>;
// def MAR : Register< 16, "mar">, DwarfRegNum<[16]>;
// def MDR : Register< 17, "mdr">, DwarfRegNum<[17]>;

//#if CH >= CH4_1
// Hi/Lo registers number and name
def HI : Cpu0Reg<0, "ac0">, DwarfRegNum<[18]>;
def LO : Cpu0Reg<0, "ac0">, DwarfRegNum<[19]>;
//#endif
def PC : Cpu0C0Reg<0, "pc">, DwarfRegNum<[20]>;
def EPC : Cpu0C0Reg<1, "epc">, DwarfRegNum<[21]>;
}

//=====
//@Register Classes
//=====

def CPUREgs : RegisterClass<"Cpu0", [i32], 32, (add
    // Reserved
    ZERO, AT,
    // Return Values and Arguments
    V0, V1, A0, A1,
    // Not preserved across procedure calls
    T9, T0, T1,
    // Callee save
    S0, S1,
    // Reserved
    GP, FP,
    SP, LR, SW)>;

```

Run *Chapter11_1*/with *ch11_1.cpp* to get the correct result as shown below:

```

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc -o
ch11_1.cpu0.o
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/
llvm-objdump -d ch11_1.cpu0.o

ch11_1.cpu0.o: file format ELF32-unknown

```

(continues on next page)

(continued from previous page)

```
Disassembly of section .text:
.text:
 0: 01 2d 00 08          ld    $2, 8($sp)
 4: 02 0d 00 04          st    $zero, 4($sp)
 8: 09 30 00 00          addiu $3, $zero, 0
 c: 13 31 20 00          add   $3, $1, $2
10: 14 32 30 00          sub   $3, $2, $3
...
```

The instructions *cmp* and *jeg* display the *\$sw* register explicitly in both assembly and disassembly. You can modify the code in the *AsmParser* and *Disassembler* (discussed in the last chapter) to hide *\$sw* in these instructions—for example, displaying *jeq 20* instead of *jeq \$sw, 20*.

Both *AsmParser* and *Cpu0AsmParser* inherit from *MCAsmParser* as follows:

Ilvm/lib/MC/MCParser/AsmParser.cpp

```
class AsmParser : public MCAsmParser {
  ...
}
```

AsmParser will call the *ParseInstruction()* and *MatchAndEmitInstruction()* functions of *Cpu0AsmParser* as follows:

Ilvm/lib/MC/MCParser/AsmParser.cpp

```
bool AsmParser::parseStatement(ParseStatementInfo &Info) {
  ...
  // Directives start with "."
  if (IDVal[0] == '.' && IDVal != ".") {
    // First query the target-specific parser. It will return 'true' if it
    // isn't interested in this directive.
    if (!getTargetParser().ParseDirective(ID))
      return false;
    ...
  }
  ...
  bool HadError = getTargetParser().ParseInstruction(IInfo, OpcodeStr, IDLoc,
                                                    Info.ParsedOperands);
  ...
  // If parsing succeeded, match the instruction.
  if (!HadError) {
    unsigned ErrorInfo;
    getTargetParser().MatchAndEmitInstruction(IDLoc, Info.Opcode,
                                              Info.ParsedOperands, Out,
                                              ErrorInfo, ParsingInlineAsm);
  }
  ...
}
```

11.2 Assembler structure

Run `llc` with the option `-debug-only=asm-matcher,cpu0-asm-parser` to observe how the Cpu0 assembler works.

The `AsmParser` directory handles the translation from assembly to object files.

The assembling Data Flow Diagram (DFD) is shown in Fig. 11.1 and Fig. 11.2.

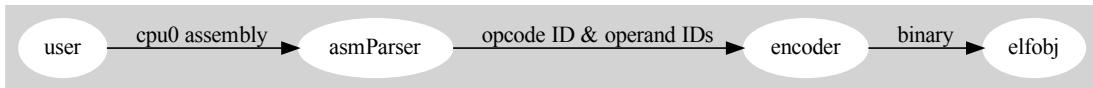


Fig. 11.1: Assembly flow

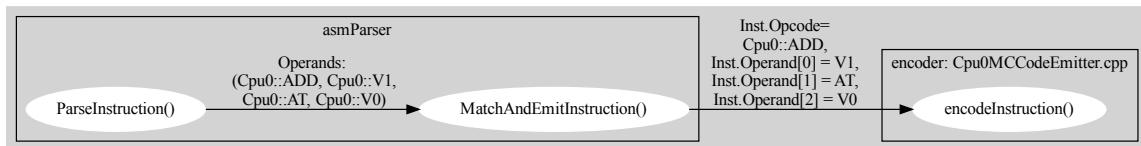


Fig. 11.2: Assembly flow, for instance: `add $v1, $v0, $at`

Given an example assembly instruction `add $v1, $v0, $at`, the LLVM AsmParser core calls the backend `ParseInstruction()` function in `Cpu0AsmParser.cpp` when it detects that the first token at the beginning of the line is an identifier.

`ParseInstruction()` parses a single assembly instruction, creates the operand objects, and returns them to the LLVM AsmParser. Then, the AsmParser calls the backend `MatchAndEmitInstruction()` function to assign the opcode and operands to an `MCInst`. The encoder then encodes the binary instruction from the `MCInst` using information from `Cpu0InstrInfo.td`, which includes the binary values for the opcode ID and operand IDs of the instruction.

Below is a list of the key functions and data structures in `MatchAndEmitInstruction()` and `encodeInstruction()`, with explanations provided in comments beginning with `///`.

llvm/build/lib/Target/Cpu0/Cpu0GenAsmMatcher.inc

```

enum InstructionConversionKind {
    Convert_Reg1_0_Reg1_1_Reg1_2,
    Convert_Reg1_0_Reg1_1_Imm1_2,
    ...
    CVT_NUM_SIGNATURES
};

} // end anonymous namespace

struct MatchEntry {
    uint16_t Mnemonic;
    uint16_t Opcode;
}
  
```

(continues on next page)

(continued from previous page)

```

uint8_t ConvertFn;
uint32_t RequiredFeatures;
uint8_t Classes[3];
StringRef getMnemonic() const {
    return StringRef(MnemonicTable + Mnemonic + 1,
                      MnemonicTable[Mnemonic]);
}
};

static const MatchEntry MatchTable0[] = {
{ 0 /* add */, Cpu0::ADD, Convert_Reg1_0_Reg1_1_Reg1_2, 0, { MCK_CPURegs, MCK_
    ↪CPURegs, MCK_CPURegs }, },
{ 4 /* addiu */, Cpu0::ADDIU, Convert_Reg1_0_Reg1_1_Imm1_2, 0, { MCK_CPURegs, ↪
    ↪MCK_CPURegs, MCK_Imm }, },
...
};

unsigned Cpu0AsmParser::
MatchInstructionImpl(const OperandVector &Operands,
                    MCInst &Inst, uint64_t &ErrorInfo,
                    bool matchingInlineAsm, unsigned VariantID) {
...
// Find the appropriate table for this asm variant.
const MatchEntry *Start, *End;
switch (VariantID) {
default: llvm_unreachable("invalid variant!");
case 0: Start = std::begin(MatchTable0); End = std::end(MatchTable0); break;
}
// Search the table.
auto MnemonicRange = std::equal_range(Start, End, Mnemonic, LessOpcode());
...
for (const MatchEntry *it = MnemonicRange.first, *ie = MnemonicRange.second;
      it != ie; ++it) {
...
// We have selected a definite instruction, convert the parsed
// operands into the appropriate MCInst.

    /// For instance ADD , V1, AT, V0
    /// MnemonicRange.first = &MatchTable0[0]
    /// MnemonicRange.second = &MatchTable0[1]
    /// it.ConvertFn = Convert_Reg1_0_Reg1_1_Reg1_2

    convertToMCInst(it->ConvertFn, Inst, it->Opcode, Operands);
...
}
...
}

static const uint8_t ConversionTable[CVT_NUM_SIGNATURES][9] = {
// Convert_Reg1_0_Reg1_1_Reg1_2
{ CVT_95_Reg, 1, CVT_95_Reg, 2, CVT_95_Reg, 3, CVT_Done },
// Convert_Reg1_0_Reg1_1_Imm1_2

```

(continues on next page)

(continued from previous page)

```

{ CVT_95_Reg, 1, CVT_95_Reg, 2, CVT_95_addImmOperands, 3, CVT_Done },
...
};

/// When kind = Convert__Reg1_0__Reg1_1__Reg1_2, ConversionTable[Kind] is equal to_
//→CVT_95_Reg
/// For Operands[1], Operands[2], Operands[3] do the following:
/// static_cast<Cpu0Operand&>(*Operands[OpIdx]).addRegOperands(Inst, 1);
/// Since p = 0, 2, 4, then OpIdx = 1, 2, 3 when OpIdx=*(p+1).
/// Since, Operands[1] = V1, Operands[2] = AT, Operands[3] = V0,
/// for "ADD , V1, AT, V0" which created by ParseInstruction().
/// Inst.Opcode = ADD, Inst.Operand[0] = V1, Inst.Operand[1] = AT,
/// Inst.Operand[2] = V0.
void Cpu0AsmParser:::
convertToMCInst(unsigned Kind, MCInst &Inst, unsigned Opcode,
               const OperandVector &Operands) {
    assert(Kind < CVT_NUM_SIGNATURES && "Invalid signature!");
    const uint8_t *Converter = ConversionTable[Kind];
    unsigned OpIdx;
    Inst.setOpcode(Opcode);
    for (const uint8_t *p = Converter; *p; p+= 2) {
        OpIdx = *(p + 1);
        switch (*p) {
            default: llvm_unreachable("invalid conversion entry!");
            case CVT_Reg:
                static_cast<Cpu0Operand&>(*Operands[OpIdx]).addRegOperands(Inst, 1);
                break;
            ...
        }
    }
}
}

```

Index/chapters/Chapter11_1/AsmParser/Cpu0AsmParser.cpp

```

/// For "ADD , V1, AT, V0", ParseInstruction() set Operands[1].Reg.RegNum = V1,
/// Operands[2].Reg.RegNum = AT, ..., by Cpu0Operand::CreateReg(RegNo, S,
/// Parser.getTok().getLoc()) in calling ParseOperand().
/// So, after (*Operands[1..3]).addRegOperands(Inst, 1),
/// Inst.Opcode = ADD, Inst.Operand[0] = V1, Inst.Operand[1] = AT,
/// Inst.Operand[2] = V0.
class Cpu0Operand : public MCParsedAsmOperand {
    ...
    void addRegOperands(MCInst &Inst, unsigned N) const {
        assert(N == 1 && "Invalid number of operands!");
        Inst.addOperand(MCOperand::createReg(getReg()));
    }
    ...
    unsigned getReg() const override {
        assert((Kind == k_Register) && "Invalid access!");
        return Reg.RegNum;
    }
}

```

(continues on next page)

(continued from previous page)

```
...
}
```

Ibdex/chapters/Chapter11_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
void Cpu0MCCodeEmitter::  
encodeInstruction(const MCInst &MI, raw_ostream &OS,  
                  SmallVectorImpl<MCFixup> &Fixups,  
                  const MCSubtargetInfo &STI) const  
{  
    uint32_t Binary = getBinaryCodeForInstr(MI, Fixups, STI);  
    ...  
    EmitInstruction(Binary, Size, OS);  
}
```

Ilvm/build/lib/Target/Cpu0/Cpu0GenMCCodeEmitter.inc

```
uint64_t Cpu0MCCodeEmitter::getBinaryCodeForInstr(const MCInst &MI,  
                                                SmallVectorImpl<MCFixup> &Fixups,  
                                                const MCSubtargetInfo &STI) const {  
    static const uint64_t InstBits[] = {  
        ...  
        UINT64_C(318767104),           // ADD  /// 318767104=0x13000000  
        ...  
    };  
    ...  
  
    const unsigned opcode = MI.getOpcode();  
    ...  
    switch (opcode) {  
        case Cpu0::ADD:  
            ...  
            // op: ra  
            op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);  
            Value |= (op & UINT64_C(15)) << 20;  
            // op: rb  
            op = getMachineOpValue(MI, MI.getOperand(1), Fixups, STI);  
            Value |= (op & UINT64_C(15)) << 16;  
            // op: rc  
            op = getMachineOpValue(MI, MI.getOperand(2), Fixups, STI);  
            Value |= (op & UINT64_C(15)) << 12;  
            break;  
    }  
    ...  
}  
return Value;  
}
```

MatchTable0 includes all possible combinations of opcodes and operand types.

Even if a user's assembly instruction passes the syntax check in *Cpu0AsmParser*, *MatchAndEmitInstruction()* can still fail. For example, the instruction *asm("move \$3, \$2")*; may succeed, but *asm("move \$3, \$2, \$1")*; will fail.

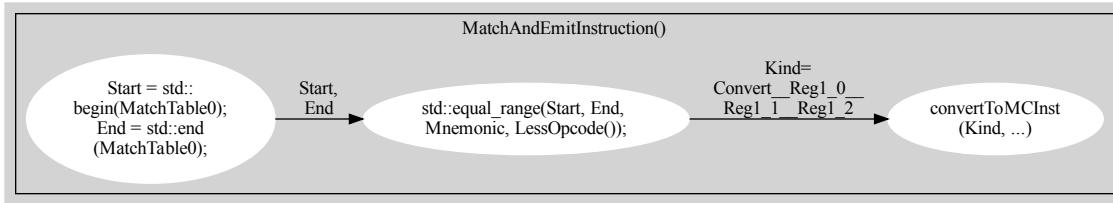


Fig. 11.3: Data flow in `MatchAndEmitInstruction()`, for instance: add \$v1, \$v0, \$at”

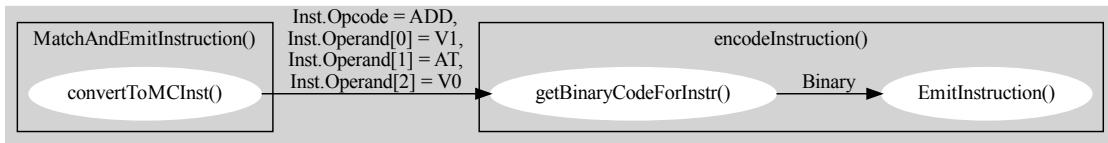


Fig. 11.4: Data flow between `MatchAndEmitInstruction()` and `encodeInstruction()`, for instance: add \$v1, \$v0, \$at

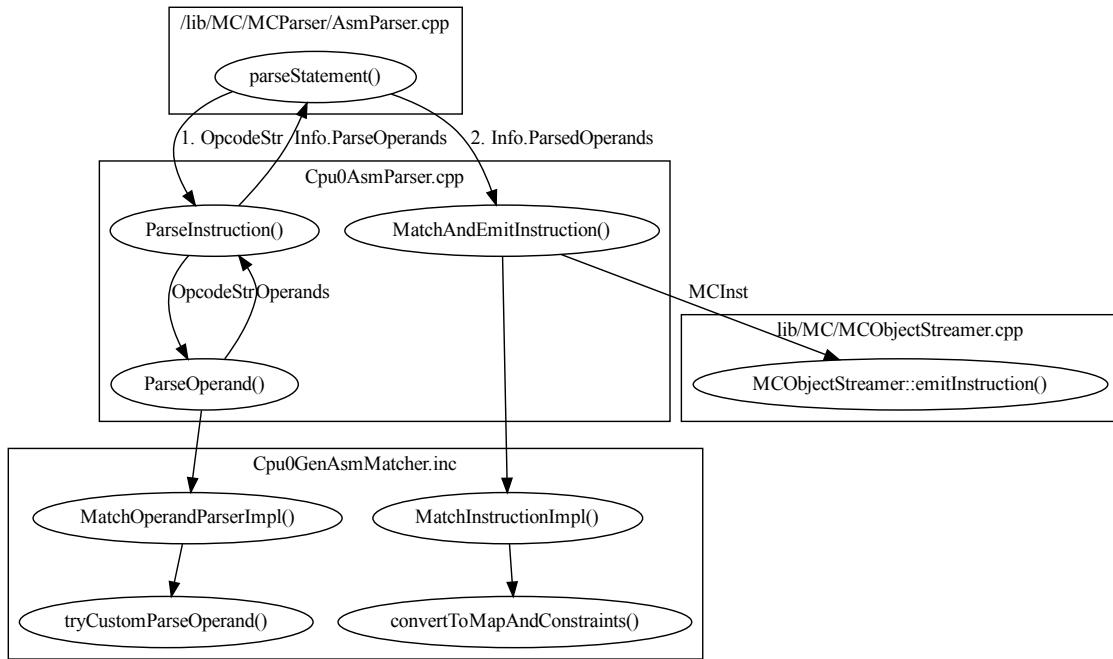
The flow of function calls for *Cpu0AsmParser* is shown in Fig. 11.5.

- After `ParseInstruction()` and `MatchAndEmitInstruction()` are called, an `MCInst` object is produced.
 - In `MatchAndEmitInstruction()`, the assembler calls `MCOObjectStreamer::emitInstruction()` to encode the instruction into binary. See Fig. 5.2 for reference.
- Run `llc` with the option `-debug` or `-debug-only=asm-matcher,cpu0-asm-parser` to display debug messages and trace the assembler flow, as shown below.
- For Cpu0, only memory operands (used in L-type or J-type instructions) will trigger a call to `tryCustomParseOperand()`.

```

input % ~/llvm/test/build/bin/clang -target mips-unknown-linux-gnu -c
ch11_1.cpp -emit-llvm -o ch11_1.bc
input % ~/llvm/test/build/bin/llc -march=cpu0 -relocation-model=pic
-filetype=obj -debug-only=asm-matcher,cpu0-asm-parser ch11_1.bc -o
ch11_1.cpu0.o

ParseOperand
.. Generic Parser
ParseOperand
<ld><Register<19>><Mem<9, 8>>
AsmMatcher: found 1 encodings with mnemonic 'ld'
Trying to match opcode LD
Matching formal operand class MCK_CPURegs against actual operand at index 1
(Register<19>): match success using generic matcher
Matching formal operand class MCK_Mem against actual operand at index 2
(Mem<9, 8>): match success using generic matcher
Matching formal operand class InvalidMatchClass against actual operand at
(continues on next page)
  
```


 Fig. 11.5: Flow of calling functions for `Cpu0AsmParser`.

(continued from previous page)

```

index 3: actual operand index out of range Opcode result: complete match,
selecting this opcode
    
```

The other functions in `Cpu0AsmParser` are called in the following flow:

- `ParseDirective()` → `parseDirectiveSet()` → `parseSetReorderDirective()`, `parseSetNoReorderDirective()`, `parseSetMacroDirective()`, `parseSetNoMacroDirective()` → `reportParseError()`
- `ParseInstruction()` → `ParseOperand()` → `MatchOperandParserImpl()` (in `Cpu0GenAsmMatcher.inc`) → `tryCustomParseOperand()` (in `Cpu0GenAsmMatcher.inc`) → `parseMemOperand()` → `parseMemOffset()`, `tryParseRegisterOperand()`
- `MatchAndEmitInstruction()` → `MatchInstructionImpl()` (in `Cpu0GenAsmMatcher.inc`) → `needsExpansion()` → `expandInstruction()`
- `parseMemOffset()` → `parseRelocOperand()` → `getVariantKind()`
- `tryParseRegisterOperand()` → `tryParseRegister()` → `matchRegisterName()` → `getReg()`, `matchRegisterByName()`
- `expandInstruction()` → `expandLoadImm()`, `expandLoadAddressImm()`, `expandLoadAddressReg()` → `EmitInstruction()` (in `Cpu0AsmPrint.cpp`)

11.3 Inline assembly

Run Chapter11_1 with ch11_2 will get the following error.

Ibdex/input/ch11_2.cpp

```
extern "C" int printf(const char *format, ...);
int inlineasm_addu(void)
{
    int foo = 10;
    const int bar = 15;

//  call i32 asm sideeffect "addu $0,$1,$2", "=r,r,r"(i32 %1, i32 %2) #1, !srcloc !1
__asm__ __volatile__("addu %0,%1,%2"
                     :"=r"(foo) // 5
                     :"r"(foo), "r"(bar)
                     );

    return foo;
}

int inlineasm_longlong(void)
{
    int a, b;
    const long long bar = 0x0000000500000006;
    int* p = (int*)&bar;
//  int* q = (p+1); // Do not set q here.

//  call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %2) #2, !srcloc !2
__asm__ __volatile__("ld %0,%1"
                     :"=r"(a) // 0x700070007000700b
                     :"m"(*p)
                     );
    int* q = (p+1); // Set q just before inline asm refer to avoid register clobbered.
//  call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %6) #2, !srcloc !3
__asm__ __volatile__("ld %0,%1"
                     :"=r"(b) // 11
                     :"m"(*q)
//          Or use :"m"(*(p+1)) to avoid register clobbered.
                     );

    return (a+b);
}

int inlineasm_constraint(void)
{
    int foo = 10;
    const int n_5 = -5;
    const int n5 = 5;
    const int n0 = 0;
    const unsigned int un5 = 5;
    const int n65536 = 0x10000;
    const int n_65531 = -65531;
```

(continues on next page)

(continued from previous page)

```

//    call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,I"(i32 %1, i32 15) #1, !srcloc !2
__asm__ __volatile__("addiu %0,%1,%2"
                     : "=r" (foo) // 15
                     : "r" (foo), "I" (n_5)
                     );
__asm__ __volatile__("addiu %0,%1,%2"
                     : "=r" (foo) // 15
                     : "r" (foo), "J" (n0)
                     );
__asm__ __volatile__("addiu %0,%1,%2"
                     : "=r" (foo) // 10
                     : "r" (foo), "K" (n5)
                     );
__asm__ __volatile__("ori %0,%1,%2"
                     : "=r" (foo) // 10
                     : "r" (foo), "L" (n65536) // 0x10000 = 65536
                     );
__asm__ __volatile__("addiu %0,%1,%2"
                     : "=r" (foo) // 15
                     : "r" (foo), "N" (n_65531)
                     );
__asm__ __volatile__("addiu %0,%1,%2"
                     : "=r" (foo) // 10
                     : "r" (foo), "O" (n_5)
                     );
__asm__ __volatile__("addiu %0,%1,%2"
                     : "=r" (foo) // 15
                     : "r" (foo), "P" (un5)
                     );
return foo;
}

int inlineasm_arg(int u, int v)
{
    int w;

    __asm__ __volatile__("subu %0,%1,%2"
                        : "=r" (w)
                        : "r" (u), "r" (v)
                        );
    return w;
}

```

(continues on next page)

(continued from previous page)

```
int g[3] = {1,2,3};

int inlineasm_global()
{
    int c, d;
    __asm__ __volatile__("ld %0,%1"
        :"=r"(c) // c=3
        :"m"(g[2])
        );
    __asm__ __volatile__("addiu %0,%1,1"
        :"=r"(d) // d=4
        :"r"(c)
        );
    return d;
}

#ifndef TESTSOFTFLOATLIB
// test_float() will call soft float library
int inlineasm_float()
{
    float a = 2.2;
    float b = 3.3;

    int c = (int)(a + b);

    int d;
    __asm__ __volatile__("addiu %0,%1,1"
        :"=r"(d)
        :"r"(c)
        );
    return d;
}
#endif

int test_inlineasm()
{
    int a, b, c, d, e, f;

    a = inlineasm_addu(); // 25
    b = inlineasm_longlong(); // 11
    c = inlineasm_constraint(); // 15
    d = inlineasm_arg(1, 10); // -9
    e = inlineasm_arg(6, 3); // 3
    __asm__ __volatile__("addiu %0,%1,1"
        :"=r"(f) // e=4
        :"r"(e)
        );
    return (a+b+c+d+e+f); // 25+11+15-9+3+4=49
```

(continues on next page)

(continued from previous page)

}

```
1-160-129-73:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -filetype=asm ch11_2.bc -o -
.section .mdebug.abi32
.previous
.file "ch11_2.bc"
error: couldn't allocate output register for constraint 'r'
```

The *ch11_2.cpp* file is an inline assembly example. Clang supports inline assembly similarly to GCC.

Inline assembly is used in C/C++ when a program needs to access a specific allocated register or memory location for a C/C++ variable. For example, the variable *foo* in *ch11_2.cpp* may be allocated by the compiler to register \$2, \$3, or another register.

Inline assembly helps bridge the gap between high-level languages and assembly language. See reference².

Chapter11_2 adds support for inline assembly as follows:

Ibdex/chapters/Chapter11_2/Cpu0AsmPrinter.h

```
bool PrintAsmOperand(const MachineInstr *MI, unsigned OpNo,
                      const char *ExtraCode, raw_ostream &O) override;
bool PrintAsmMemoryOperand(const MachineInstr *MI, unsigned OpNum,
                           const char *ExtraCode, raw_ostream &O) override;
void printOperand(const MachineInstr *MI, int opNum, raw_ostream &O);
```

Ibdex/chapters/Chapter11_2/Cpu0AsmPrinter.cpp

```
// Print out an operand for an inline asm expression.
bool Cpu0AsmPrinter::PrintAsmOperand(const MachineInstr *MI, unsigned OpNum,
                                      const char *ExtraCode, raw_ostream &O) {
    // Does this asm operand have a single letter operand modifier?
    if (ExtraCode && ExtraCode[0]) {
        if (ExtraCode[1] != 0) return true; // Unknown modifier.

        const MachineOperand &MO = MI->getOperand(OpNum);
        switch (ExtraCode[0]) {
            default:
                // See if this is a generic print operand
                return AsmPrinter::PrintAsmOperand(MI, OpNum, ExtraCode, O);
            case 'X': // hex const int
                if ((MO.getType()) != MachineOperand::MO_Immediate)
                    return true;
                O << "0x" << StringRef(utohexstr(MO.getImm())).lower();
                return false;
            case 'x': // hex const int (low 16 bits)
                if ((MO.getType()) != MachineOperand::MO_Immediate)
                    return true;
                O << "0x" << StringRef(utohexstr(MO.getImm() & 0xffff)).lower();
        }
    }
}
```

(continues on next page)

² <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>

(continued from previous page)

```
    return false;
  case 'd': // decimal const int
    if ((MO.getType()) != MachineOperand::MO_Immediate)
      return true;
    O << MO.getImm();
    return false;
  case 'm': // decimal const int minus 1
    if ((MO.getType()) != MachineOperand::MO_Immediate)
      return true;
    O << MO.getImm() - 1;
    return false;
  case 'z': {
    // $0 if zero, regular printing otherwise
    if (MO.getType() != MachineOperand::MO_Immediate)
      return true;
    int64_t Val = MO.getImm();
    if (Val)
      O << Val;
    else
      O << "$0";
    return false;
  }
}
}

printOperand(MI, OpNum, O);
return false;
}

bool Cpu0AsmPrinter::PrintAsmMemoryOperand(const MachineInstr *MI,
                                           unsigned OpNum,
                                           const char *ExtraCode,
                                           raw_ostream &O) {
  int Offset = 0;
  // Currently we are expecting either no ExtraCode or 'D'
  if (ExtraCode) {
    return true; // Unknown modifier.
  }

  const MachineOperand &MO = MI->getOperand(OpNum);
  assert(MO.isReg() && "unexpected inline asm memory operand");
  O << Offset << "(" << Cpu0InstPrinter::getRegisterName(MO.getReg()) << ")";

  return false;
}

void Cpu0AsmPrinter::printOperand(const MachineInstr *MI, int opNum,
                                 raw_ostream &O) {
  const MachineOperand &MO = MI->getOperand(opNum);
  bool closeP = false;

  if (MO.getTargetFlags())
```

(continues on next page)

(continued from previous page)

```

closeP = true;

switch(MO.getTargetFlags()) {
    case Cpu0II::MO_GPREL:    O << "%gp_rel("; break;
    case Cpu0II::MO_GOT_CALL: O << "%call16("; break;
    case Cpu0II::MO_GOT:      O << "%got(";      break;
    case Cpu0II::MO_ABS_HI:   O << "%hi(";       break;
    case Cpu0II::MO_ABS_LO:   O << "%lo(";       break;
    case Cpu0II::MO_GOT_HI16: O << "%got_hi16("; break;
    case Cpu0II::MO_GOT_LO16: O << "%got_lo16("; break;
}

switch (MO.getType()) {
    case MachineOperand::MO_Register:
        O << '$'
        << StringRef(Cpu0InstPrinter::getRegisterName(MO.getReg())).lower();
        break;

    case MachineOperand::MO_Immediate:
        O << MO.getImm();
        break;

    case MachineOperand::MO_MachineBasicBlock:
        O << *MO.getMBB()->getSymbol();
        return;

    case MachineOperand::MO_GlobalAddress:
        O << *getSymbol(MO.getGlobal());
        break;

    case MachineOperand::MO_BlockAddress: {
        MCSymbol *BA = GetBlockAddressSymbol(MO.getBlockAddress());
        O << BA->getName();
        break;
    }

    case MachineOperand::MO_ExternalSymbol:
        O << *GetExternalSymbolSymbol(MO.getSymbolName());
        break;

    case MachineOperand::MO_JumpTableIndex:
        O << MAI->getPrivateGlobalPrefix() << "JTI" << getFunctionNumber()
            << '_' << MO.getIndex();
        break;

    case MachineOperand::MO_ConstantPoolIndex:
        O << MAI->getPrivateGlobalPrefix() << "CPI"
            << getFunctionNumber() << "_" << MO.getIndex();
        if (MO.getOffset())
            O << "+" << MO.getOffset();
        break;
}

```

(continues on next page)

(continued from previous page)

```
default:  
    llvm_unreachable("<unknown operand type>");  
  
}  
  
if (closeP) O << " ");  
}
```

Ibdex/chapters/Chapter11_2/Cpu0InstrInfo.cpp

```
/// Return the number of bytes of code the specified instruction may be.  
unsigned Cpu0InstrInfo::GetInstSizeInBytes(const MachineInstr &MI) const {  
  
    case TargetOpcode::INLINEASM: { // Inline Asm: Variable size.  
        const MachineFunction *MF = MI.getParent()->getParent();  
        const char *AsmStr = MI.getOperand(0).getSymbolName();  
        return getInlineAsmLength(AsmStr, *MF->getTarget().getMCAsmInfo());  
    }  
}
```

Ibdex/chapters/Chapter11_2/Cpu0ISelDAGToDAG.h

```
bool SelectInlineAsmMemoryOperand(const SDValue &Op,  
                                  unsigned ConstraintID,  
                                  std::vector<SDValue> &OutOps) override;
```

Ibdex/chapters/Chapter11_2/Cpu0ISelDAGToDAG.cpp

```
// inlineasm begin  
bool Cpu0DAGToDAGISel::  
SelectInlineAsmMemoryOperand(const SDValue &Op, unsigned ConstraintID,  
                           std::vector<SDValue> &OutOps) {  
    // All memory constraints can at least accept raw pointers.  
    switch(ConstraintID) {  
    default:  
        llvm_unreachable("Unexpected asm memory constraint");  
    case InlineAsm::Constraint_m:  
        OutOps.push_back(Op);  
        return false;  
    }  
    return true;  
}  
// inlineasm end
```

Ibdex/chapters/Chapter11_2/Cpu0ISelLowering.h

```
// Inline asm support  
ConstraintType getConstraintType(StringRef Constraint) const override;  
  
/// Examine constraint string and operand type and determine a weight value.  
/// The operand object must already have been set up with the operand type.
```

(continues on next page)

(continued from previous page)

```

ConstraintWeight getSingleConstraintMatchWeight(
    AsmOperandInfo &info, const char *constraint) const override;

/// This function parses registers that appear in inline-asn constraints.
/// It returns pair (0, 0) on failure.
std::pair<unsigned, const TargetRegisterClass *>
parseRegForInlineAsmConstraint(const StringRef &C, MVT VT) const;

std::pair<unsigned, const TargetRegisterClass *>
getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                             StringRef Constraint, MVT VT) const override;

/// LowerAsmOperandForConstraint - Lower the specified operand into the Ops
/// vector. If it is invalid, don't add anything to Ops. If hasMemory is
/// true it means one of the asm constraint of the inline asm instruction
/// being processed is 'm'.
void LowerAsmOperandForConstraint(SDValue Op,
                                  std::string &Constraint,
                                  std::vector<SDValue> &Ops,
                                  SelectionDAG &DAG) const override;

bool isLegalAddressingMode(const DataLayout &DL, const AddrMode &AM,
                           Type *Ty, unsigned AS,
                           Instruction *I = nullptr) const override;

```

Ibdex/chapters/Chapter11_2/Cpu0ISelLowering.cpp

```

//=====
//          Cpu0 Inline Assembly Support
//=====

/// getConstraintType - Given a constraint letter, return the type of
/// constraint it is for this target.
Cpu0TargetLowering::ConstraintType
Cpu0TargetLowering::getConstraintType(StringRef Constraint) const
{
    // Cpu0 specific constraints
    // GCC config/mips/constraints.md
    // 'c' : A register suitable for use in an indirect
    //       jump. This will always be $t9 for -mabicalls.
    if (Constraint.size() == 1) {
        switch (Constraint[0]) {
            default : break;
            case 'c':
                return C_RegisterClass;
            case 'R':
                return C_Memory;
        }
    }
    return TargetLowering::getConstraintType(Constraint);
}

```

(continues on next page)

(continued from previous page)

```
/// Examine constraint type and operand type and determine a weight value.
/// This object must already have been set up with the operand type
/// and the current alternative constraint selected.
TargetLowering::ConstraintWeight
Cpu0TargetLowering::getSingleConstraintMatchWeight(
    AsmOperandInfo &info, const char *constraint) const {
    ConstraintWeight weight = CW_Invalid;
    Value *CallOperandVal = info.CallOperandVal;
    // If we don't have a value, we can't do a match,
    // but allow it at the lowest weight.
    if (!CallOperandVal)
        return CW_Default;
    Type *type = CallOperandVal->getType();
    // Look at the constraint type.
    switch (*constraint) {
    default:
        weight = TargetLowering::getSingleConstraintMatchWeight(info, constraint);
        break;
    case 'c': // $t9 for indirect jumps
        if (type->isIntegerTy())
            weight = CW_SpecificReg;
        break;
    case 'I': // signed 16 bit immediate
    case 'J': // integer zero
    case 'K': // unsigned 16 bit immediate
    case 'L': // signed 32 bit immediate where lower 16 bits are 0
    case 'N': // immediate in the range of -65535 to -1 (inclusive)
    case 'O': // signed 15 bit immediate (+- 16383)
    case 'P': // immediate in the range of 65535 to 1 (inclusive)
        if (isa<ConstantInt>(CallOperandVal))
            weight = CW_Constant;
        break;
    case 'R':
        weight = CW_Memory;
        break;
    }
    return weight;
}

/// This is a helper function to parse a physical register string and split it
/// into non-numeric and numeric parts (Prefix and Reg). The first boolean flag
/// that is returned indicates whether parsing was successful. The second flag
/// is true if the numeric part exists.
static std::pair<bool, bool>
parsePhysicalReg(constStringRef &C, std::string &Prefix,
                 unsigned long long &Reg) {
    if (C.front() != '{' || C.back() != '}')
        return std::make_pair(false, false);

    // Search for the first numeric character.
    StringRef::const_iterator I, B = C.begin() + 1, E = C.end() - 1;
```

(continues on next page)

(continued from previous page)

```

I = std::find_if(B, E, isdigit);

Prefix.assign(B, I - B);

// The second flag is set to false if no numeric characters were found.
if (I == E)
    return std::make_pair(true, false);

// Parse the numeric characters.
return std::make_pair(!getAsUnsignedInteger(StringRef(I, E - I), 10, Reg),
                     true);
}

std::pair<unsigned, const TargetRegisterClass *> Cpu0TargetLowering::
parseRegForInlineAsmConstraint(const StringRef &C, MVT VT) const {
    const TargetRegisterClass *RC;
    std::string Prefix;
    unsigned long long Reg;

    std::pair<bool, bool> R = parsePhysicalReg(C, Prefix, Reg);

    if (!R.first)
        return std::make_pair(0U, nullptr);
    if (!R.second)
        return std::make_pair(0U, nullptr);

    // Parse $0-$15.
    assert(Prefix == "$");
    RC = getRegClassFor((VT == MVT::Other) ? MVT::i32 : VT);

    assert(Reg < RC->getNumRegs());
    return std::make_pair(*(RC->begin() + Reg), RC);
}

/// Given a register class constraint, like 'r', if this corresponds directly
/// to an LLVM register class, return a register of 0 and the register class
/// pointer.
std::pair<unsigned, const TargetRegisterClass *>
Cpu0TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                                                StringRef Constraint,
                                                MVT VT) const
{
    if (Constraint.size() == 1) {
        switch (Constraint[0]) {
        case 'r':
            if (VT == MVT::i32 || VT == MVT::i16 || VT == MVT::i8) {
                return std::make_pair(0U, &Cpu0::CPURegsRegClass);
            }
            if (VT == MVT::i64)
                return std::make_pair(0U, &Cpu0::CPURegsRegClass);
            // This will generate an error message
            return std::make_pair(0u, static_cast<const TargetRegisterClass*>(0));
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
case 'c': // register suitable for indirect jump
    if (VT == MVT::i32)
        return std::make_pair((unsigned)Cpu0::T9, &Cpu0::CPURegsRegClass);
    assert(0 && "Unexpected type.");
}
}

std::pair<unsigned, const TargetRegisterClass *> R;
R = parseRegForInlineAsmConstraint(Constraint, VT);

if (R.second)
    return R;

return TargetLowering::getRegForInlineAsmConstraint(TRI, Constraint, VT);
}

/// LowerAsmOperandForConstraint - Lower the specified operand into the Ops
/// vector. If it is invalid, don't add anything to Ops.
void Cpu0TargetLowering::LowerAsmOperandForConstraint(SDValue Op,
                                                       std::string &Constraint,
                                                       std::vector<SDValue>&Ops,
                                                       SelectionDAG &DAG) const {
    SDLoc DL(Op);
    SDValue Result;

    // Only support length 1 constraints for now.
    if (Constraint.length() > 1) return;

    char ConstraintLetter = Constraint[0];
    switch (ConstraintLetter) {
    default: break; // This will fall through to the generic implementation
    case 'I': // Signed 16 bit constant
        // If this fails, the parent routine will give an error
        if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
            EVT Type = Op.getValueType();
            int64_t Val = C->getSExtValue();
            if (isInt<16>(Val)) {
                Result = DAG.getTargetConstant(Val, DL, Type);
                break;
            }
        }
        return;
    case 'J': // integer zero
        if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
            EVT Type = Op.getValueType();
            int64_t Val = C->getZExtValue();
            if (Val == 0) {
                Result = DAG.getTargetConstant(0, DL, Type);
                break;
            }
        }
        return;
    }
```

(continues on next page)

(continued from previous page)

```

case 'K': // unsigned 16 bit immediate
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
        EVT Type = Op.getValueType();
        uint64_t Val = (uint64_t)C->getZExtValue();
        if (isUIInt<16>(Val)) {
            Result = DAG.getTargetConstant(Val, DL, Type);
            break;
        }
    }
    return;
case 'L': // signed 32 bit immediate where lower 16 bits are 0
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
        EVT Type = Op.getValueType();
        int64_t Val = C->getSExtValue();
        if ((isInt<32>(Val)) && ((Val & 0xffff) == 0)) {
            Result = DAG.getTargetConstant(Val, DL, Type);
            break;
        }
    }
    return;
case 'N': // immediate in the range of -65535 to -1 (inclusive)
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
        EVT Type = Op.getValueType();
        int64_t Val = C->getSExtValue();
        if ((Val >= -65535) && (Val <= -1)) {
            Result = DAG.getTargetConstant(Val, DL, Type);
            break;
        }
    }
    return;
case 'O': // signed 15 bit immediate
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
        EVT Type = Op.getValueType();
        int64_t Val = C->getSExtValue();
        if ((isInt<15>(Val))) {
            Result = DAG.getTargetConstant(Val, DL, Type);
            break;
        }
    }
    return;
case 'P': // immediate in the range of 1 to 65535 (inclusive)
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
        EVT Type = Op.getValueType();
        int64_t Val = C->getSExtValue();
        if ((Val <= 65535) && (Val >= 1)) {
            Result = DAG.getTargetConstant(Val, DL, Type);
            break;
        }
    }
    return;
}

```

(continues on next page)

(continued from previous page)

```

if (Result.getNode()) {
    Ops.push_back(Result);
    return;
}

TargetLowering::LowerAsmOperandForConstraint(Op, Constraint, Ops, DAG);

bool Cpu0TargetLowering::isLegalAddressingMode(const DataLayout &DL,
                                              const AddrMode &AM, Type *Ty,
                                              unsigned AS, Instruction *I) const {
    // No global is ever allowed as a base.
    if (AM.BaseGV)
        return false;

    switch (AM.Scale) {
    case 0: // "r+i" or just "i", depending on HasBaseReg.
        break;
    case 1:
        if (!AM.HasBaseReg) // allow "r+i".
            break;
        return false; // disallow "r+r" or "r+r+i".
    default:
        return false;
    }

    return true;
}

```

Similar to the backend structure, the structure of inline assembly can also be organized by file name, as shown in the table titled “*The structure of inline assembly*”.

Table 11.1: inline assembly functions

File	Function
Cpu0ISelLowering.cpp	inline asm DAG node create
Cpu0ISelDAGToDAG.cpp	save OP code
Cpu0AsmPrinter.cpp,	inline asm instructions printing
Cpu0InstrInfo.cpp	•

Except for *Cpu0ISelDAGToDAG.cpp*, the other functions are the same as those used in the backend’s normal code compilation.

The inline assembly handling in *Cpu0ISelLowering.cpp* is explained after showing the result of running with *ch11_2.cpp*.

Cpu0ISelDAGToDAG.cpp simply saves the opcode in *SelectInlineAsmMemoryOperand()*. Since the opcode represents a Cpu0 inline assembly instruction, no further LLVM IR DAG translation is needed. The function just saves the opcode directly and returns *false* to notify the LLVM system that the Cpu0 backend has finished processing this inline assembly instruction.

Run *Chapter11_2* with *ch11_2.cpp* to get the following result:

```

1-160-129-73:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch11_2.cpp -emit-llvm -o ch11_2.bc

1-160-129-73:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch11_2.bc -o -
...
target triple = "mips-unknown-linux-gnu"

@g = global [3 x i32] [i32 1, i32 2, i32 3], align 4

; Function Attrs: nounwind
define i32 @_Z14inlineasm_adduv() #0 {
    %foo = alloca i32, align 4
    %bar = alloca i32, align 4
    store i32 10, i32* %foo, align 4
    store i32 15, i32* %bar, align 4
    %1 = load i32* %foo, align 4
    %2 = call i32 asm sideeffect "addu $0,$1,$2", "=r,r,r"(i32 %1, i32 15) #1,
    !srcloc !1
    store i32 %2, i32* %foo, align 4
    %3 = load i32* %foo, align 4
    ret i32 %3
}

; Function Attrs: nounwind
define i32 @_Z18inlineasm_longlongv() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %bar = alloca i64, align 8
    %p = alloca i32*, align 4
    %q = alloca i32*, align 4
    store i64 21474836486, i64* %bar, align 8
    %1 = bitcast i64* %bar to i32*
    store i32* %1, i32** %p, align 4
    %2 = load i32** %p, align 4
    %3 = call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %2) #1, !srcloc !2
    store i32 %3, i32* %a, align 4
    %4 = load i32** %p, align 4
    %5 = getelementptr inbounds i32* %4, i32 1
    store i32* %5, i32** %q, align 4
    %6 = load i32** %q, align 4
    %7 = call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %6) #1, !srcloc !3
    store i32 %7, i32* %b, align 4
    %8 = load i32* %a, align 4
    %9 = load i32* %b, align 4
    %10 = add nsw i32 %8, %9
    ret i32 %10
}

; Function Attrs: nounwind
define i32 @_Z20inlineasm_constraintv() #0 {
    %foo = alloca i32, align 4
    %n_5 = alloca i32, align 4

```

(continues on next page)

(continued from previous page)

```
%n5 = alloca i32, align 4
%n0 = alloca i32, align 4
%un5 = alloca i32, align 4
%n65536 = alloca i32, align 4
%n_65531 = alloca i32, align 4
store i32 10, i32* %foo, align 4
store i32 -5, i32* %n_5, align 4
store i32 5, i32* %n5, align 4
store i32 0, i32* %n0, align 4
store i32 5, i32* %un5, align 4
store i32 65536, i32* %n65536, align 4
store i32 -65531, i32* %n_65531, align 4
%1 = load i32* %foo, align 4
%2 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,I"(i32 %1, i32 -5) #1,
!srcloc !4
store i32 %2, i32* %foo, align 4
%3 = load i32* %foo, align 4
%4 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,J"(i32 %3, i32 0) #1,
!srcloc !5
store i32 %4, i32* %foo, align 4
%5 = load i32* %foo, align 4
%6 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,K"(i32 %5, i32 5) #1,
!srcloc !6
store i32 %6, i32* %foo, align 4
%7 = load i32* %foo, align 4
%8 = call i32 asm sideeffect "ori $0,$1,$2", "=r,r,L"(i32 %7, i32 65536) #1,
!srcloc !7
store i32 %8, i32* %foo, align 4
%9 = load i32* %foo, align 4
%10 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,N"(i32 %9, i32 -65531)
#1, !srcloc !8
store i32 %10, i32* %foo, align 4
%11 = load i32* %foo, align 4
%12 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,O"(i32 %11, i32 -5) #1,
!srcloc !9
store i32 %12, i32* %foo, align 4
%13 = load i32* %foo, align 4
%14 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,P"(i32 %13, i32 5) #1,
!srcloc !10
store i32 %14, i32* %foo, align 4
%15 = load i32* %foo, align 4
ret i32 %15
}

; Function Attrs: nounwind
define i32 @_Z13inlineasm_argii(i32 %u, i32 %v) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %w = alloca i32, align 4
    store i32 %u, i32* %1, align 4
    store i32 %v, i32* %2, align 4
    %3 = load i32* %1, align 4
```

(continues on next page)

(continued from previous page)

```
%4 = load i32* %2, align 4
%5 = call i32 asm sideeffect "subu $0,$1,$2", "=r,r,r"(i32 %3, i32 %4) #1,
!srcloc !11
store i32 %5, i32* %w, align 4
%6 = load i32* %w, align 4
ret i32 %6
}

; Function Attrs: nounwind
define i32 @_Z16inlineasm_globalv() #0 {
    %c = alloca i32, align 4
    %d = alloca i32, align 4
    %1 = call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* getelementptr inbounds
([3 x i32]* @g, i32 0, i32 2)) #1, !srcloc !12
store i32 %1, i32* %c, align 4
%2 = load i32* %c, align 4
%3 = call i32 asm sideeffect "addiu $0,$1,1", "=r,r"(i32 %2) #1, !srcloc !13
store i32 %3, i32* %d, align 4
%4 = load i32* %d, align 4
ret i32 %4
}

; Function Attrs: nounwind
define i32 @_Z14test_inlineasmv() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    %d = alloca i32, align 4
    %e = alloca i32, align 4
    %f = alloca i32, align 4
    %g = alloca i32, align 4
    %1 = call i32 @_Z14inlineasm_adduv()
store i32 %1, i32* %a, align 4
%2 = call i32 @_Z18inlineasm_longlongv()
store i32 %2, i32* %b, align 4
%3 = call i32 @_Z20inlineasm_constraintv()
store i32 %3, i32* %c, align 4
%4 = call i32 @_Z13inlineasm_argii(i32 1, i32 10)
store i32 %4, i32* %d, align 4
%5 = call i32 @_Z13inlineasm_argii(i32 6, i32 3)
store i32 %5, i32* %e, align 4
%6 = load i32* %e, align 4
%7 = call i32 asm sideeffect "addiu $0,$1,1", "=r,r"(i32 %6) #1, !srcloc !14
store i32 %7, i32* %f, align 4
%8 = call i32 @_Z16inlineasm_globalv()
store i32 %8, i32* %g, align 4
%9 = load i32* %a, align 4
%10 = load i32* %b, align 4
%11 = add nsw i32 %9, %10
%12 = load i32* %c, align 4
%13 = add nsw i32 %11, %12
%14 = load i32* %d, align 4
```

(continues on next page)

(continued from previous page)

```

%15 = add nsw i32 %13, %14
%16 = load i32* %e, align 4
%17 = add nsw i32 %15, %16
%18 = load i32* %f, align 4
%19 = add nsw i32 %17, %18
%20 = load i32* %g, align 4
%21 = add nsw i32 %19, %20
ret i32 %21
}

...
1-160-129-73:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -filetype=asm ch11_2.bc -o -
.section .mdebug.abi32
.previous
.file "ch11_2.bc"
.text
.globl _Z14inlineasm_adduv
.align 2
.type _Z14inlineasm_adduv,@function
.ent _Z14inlineasm_adduv      # @_Z14inlineasm_adduv
_Z14inlineasm_adduv:
.frame $fp,16,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -16
st $fp, 12($sp)           # 4-byte Folded Spill
addu $fp, $sp, $zero
addiu $2, $zero, 10
st $2, 8($fp)
addiu $2, $zero, 15
st $2, 4($fp)
ld $3, 8($fp)
#APP
addu $2,$3,$2
#NO_APP
st $2, 8($fp)
addu $sp, $fp, $zero
ld $fp, 12($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
.set macro
.set reorder
.end _Z14inlineasm_adduv
$tmp3:
.size _Z14inlineasm_adduv, ($tmp3)-_Z14inlineasm_adduv

.globl _Z18inlineasm_longlongv
.align 2
.type _Z18inlineasm_longlongv,@function

```

(continues on next page)

(continued from previous page)

```

.ent _Z18inlineasm_longlongv # @_Z18inlineasm_longlongv
_Z18inlineasm_longlongv:
.frame $fp,32,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -32
st $fp, 28($sp)           # 4-byte Folded Spill
addu $fp, $sp, $zero
addiu $2, $zero, 6
st $2, 12($fp)
addiu $2, $zero, 5
st $2, 8($fp)
addiu $2, $fp, 8
st $2, 4($fp)
#APP
ld $2,0($2)
#NO_APP
st $2, 24($fp)
ld $2, 4($fp)
addiu $2, $2, 4
st $2, 0($fp)
#APP
ld $2,0($2)
#NO_APP
st $2, 20($fp)
ld $3, 24($fp)
addu $2, $3, $2
addu $sp, $fp, $zero
ld $fp, 28($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 32
ret $lr
.set macro
.set reorder
.end _Z18inlineasm_longlongv
$tmp7:
.size _Z18inlineasm_longlongv, ($tmp7)-_Z18inlineasm_longlongv

.globl _Z20inlineasm_constraintv
.align 2
.type _Z20inlineasm_constraintv,@function
.ent _Z20inlineasm_constraintv # @_Z20inlineasm_constraintv
_Z20inlineasm_constraintv:
.frame $fp,32,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -32
st $fp, 28($sp)           # 4-byte Folded Spill
addu $fp, $sp, $zero

```

(continues on next page)

(continued from previous page)

```
addiu $2, $zero, 10
st $2, 24($fp)
addiu $2, $zero, -5
st $2, 20($fp)
addiu $2, $zero, 5
st $2, 16($fp)
addiu $3, $zero, 0
st $3, 12($fp)
st $2, 8($fp)
lui $2, 1
st $2, 4($fp)
lui $2, 65535
ori $2, $2, 5
st $2, 0($fp)
ld $2, 24($fp)
#APP
addiu $2,$2,-5
#NO_APP
st $2, 24($fp)
#APP
addiu $2,$2,0
#NO_APP
st $2, 24($fp)
#APP
addiu $2,$2,5
#NO_APP
st $2, 24($fp)
#APP
addiu $2,$2,65536
#NO_APP
st $2, 24($fp)
#APP
addiu $2,$2,-65531
#NO_APP
st $2, 24($fp)
#APP
addiu $2,$2,-5
#NO_APP
st $2, 24($fp)
#APP
addiu $2,$2,5
#NO_APP
st $2, 24($fp)
addu $sp, $fp, $zero
ld $fp, 28($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 32
ret $lr
nop
.set macro
.set reorder
.end _Z20inlineasm_constraintv
$tmp11:
```

(continues on next page)

(continued from previous page)

```

.size _Z20inlineasm_constraintv, ($tmp11)-_Z20inlineasm_constraintv

.globl _Z13inlineasm_argii
.align 2
.type _Z13inlineasm_argii,@function
.ent _Z13inlineasm_argii      # @_Z13inlineasm_argii
_Z13inlineasm_argii:
.frame $fp,16,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -16
st $fp, 12($sp)           # 4-byte Folded Spill
addu $fp, $sp, $zero
ld $2, 16($fp)
st $2, 8($fp)
ld $2, 20($fp)
st $2, 4($fp)
ld $3, 8($fp)
#APP
subu $2,$3,$2
#NO_APP
st $2, 0($fp)
addu $sp, $fp, $zero
ld $fp, 12($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
.set macro
.set reorder
.end _Z13inlineasm_argii
$tmp15:
.size _Z13inlineasm_argii, ($tmp15)-_Z13inlineasm_argii

.globl _Z16inlineasm_globalv
.align 2
.type _Z16inlineasm_globalv,@function
.ent _Z16inlineasm_globalv  # @_Z16inlineasm_globalv
_Z16inlineasm_globalv:
.frame $fp,16,$lr
.mask 0x00001000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -16
st $fp, 12($sp)           # 4-byte Folded Spill
addu $fp, $sp, $zero
lui $2, %hi(g)
ori $2, $2, %lo(g)
addiu $2, $2, 8
#APP

```

(continues on next page)

(continued from previous page)

```
ld $2,0($2)
#NO_APP
st $2, 8($fp)
#APP
addiu $2,$2,1
#NO_APP
st $2, 4($fp)
addu $sp, $fp, $zero
ld $fp, 12($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
.set macro
.set reorder
.end _Z16inlineasm_globalv
$tmp19:
.size _Z16inlineasm_globalv, ($tmp19)-_Z16inlineasm_globalv

.globl _Z14test_inlineasmv
.align 2
.type _Z14test_inlineasmv,@function
.ent _Z14test_inlineasmv      # @_Z14test_inlineasmv
_Z14test_inlineasmv:
.frame $fp,48,$lr
.mask 0x00005000,-4
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -48
st $lr, 44($sp)           # 4-byte Folded Spill
st $fp, 40($sp)           # 4-byte Folded Spill
addu $fp, $sp, $zero
jsub _Z14inlineasm_adduv
nop
st $2, 36($fp)
jsub _Z18inlineasm_longlongv
nop
st $2, 32($fp)
jsub _Z20inlineasm_constraintv
nop
st $2, 28($fp)
addiu $2, $zero, 10
st $2, 4($sp)
addiu $2, $zero, 1
st $2, 0($sp)
jsub _Z13inlineasm_argii
nop
st $2, 24($fp)
addiu $2, $zero, 3
st $2, 4($sp)
addiu $2, $zero, 6
st $2, 0($sp)
```

(continues on next page)

(continued from previous page)

```

jsub _Z13inlineasm_argi
nop
st $2, 20($fp)
#APP
addiu $2,$2,1
#NO_APP
st $2, 16($fp)
jsub _Z16inlineasm_globalv
nop
st $2, 12($fp)
ld $3, 32($fp)
ld $4, 36($fp)
addu $3, $4, $3
ld $4, 28($fp)
addu $3, $3, $4
ld $4, 24($fp)
addu $3, $3, $4
ld $4, 20($fp)
addu $3, $3, $4
ld $4, 16($fp)
addu $3, $3, $4
addu $2, $3, $2
addu $sp, $fp, $zero
ld $fp, 40($sp)           # 4-byte Folded Reload
ld $lr, 44($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 48
ret $lr
nop
.set macro
.set reorder
.end _Z14test_inlineasmv
$tmp23:
.size _Z14test_inlineasmv, ($tmp23)-_Z14test_inlineasmv

.type g,@object          # @g
.data
.globl g
.align 2
g:
.4byte 1                 # 0x1
.4byte 2                 # 0x2
.4byte 3                 # 0x3
.size g, 12

```

Clang first translates GCC-style inline assembly (`__asm__`) into LLVM IR Inline Assembler Expressions³. Then, during the `llc` register allocation stage, it replaces the SSA-form variable registers with physical registers.

In the above example, the functions `LowerAsmOperandForConstraint()` and `getSingleConstraintMatchWeight()` in `Cpu0ISelLowering.cpp` generate different ranges of constant operands based on constraint codes *I*, *J*, *K*, *L*, *N*, *O*, or *P*, and register operands based on *r*.

For instance, the following `__asm__` generates the corresponding LLVM inline assembly immediately after it:

³ <http://llvm.org/docs/LangRef.html#inline-assembler-expressions>

```
__asm__ __volatile__("addiu %0,%1,%2"
    :"=r"(foo) // 15
    :"r"(foo), "I"(n_5)
);
```

```
%2 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,I"(i32 %1, i32 -5) #0, !srcloc !1
```

```
__asm__ __volatile__("addiu %0,%1,%2"
    :"=r"(foo) // 15
    :"r"(foo), "N"(n_65531)
);
```

```
%10 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,N"(i32 %9, i32 -65531) #0, !
→srcloc !5
```

```
__asm__ __volatile__("addiu %0,%1,%2"
    :"=r"(foo) // 15
    :"r"(foo), "P"(un5)
);
```

```
%14 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,P"(i32 %13, i32 5) #0, !srcloc !
→7
```

The *r* constraint in `__asm__` will generate a register operand (e.g., `%I`) in LLVM IR inline assembly. The *I* constraint will generate a constant operand (e.g., `-5`) in LLVM IR inline assembly.

Note that `LowerAsmOperandForConstraint()` limits the range of positive and negative constant operand values to 16 bits, since the FL-type immediate operand in Cpu0 instructions is 16 bits.

As a result:

- The range of *N* is from `-65535` to `-1`.
- The range of *P* is from `1` to `65535`.

Any value outside of these ranges is treated as an error in `LowerAsmOperandForConstraint()`, due to the 16-bit limitation of the FL instruction format.

C++ SUPPORT

- *Exception Handling*
- *Thread variable*
- *C++ Memory Order*
 - *Source Code Compatibility*
 - *The Problem Before C++11*
 - *C++11 Memory Model Solution*
- *Cpu0 implementation for memory-order*

This chapter supports some C++ compiler features.

12.1 Exception Handling

Chapter11_2 can be built and run using the C++ polymorphism example code in `ch12_inherit.cpp` as follows:

Ibdex/input/ch12_inherit.cpp

```
...
class CPolygon { // _ZTVN10__cxxabiv117__class_type_infoE for parent class
...
#ifndef COUT_TEST
// generate IR invoke, landing, resume and unreachable on iMac
{ cout << this->area() << endl; }
#else
{ printf("%d\n", this->area()); }
#endif
};
```

If you use `cout` instead of `printf` in `ch12_inherit.cpp`, it will not generate exception handling IR on Linux. However, it will generate exception handling IRs such as `invoke`, `landingpad`, `resume`, and `unreachable` on iMac.

The example code `ch12_eh.cpp`, which includes `try` and `catch` exception handling, will generate these exception-related IRs on both iMac and Linux.

Ibdex/input/ch12_eh.cpp

```
class Ex1 {};
void throw_exception(int a, int b) {
    Ex1 ex1;

    if (a > b) {
        throw ex1;
    }
}

int test_try_catch() {
    try {
        throw_exception(2, 1);
    }
    catch(...) {
        return 1;
    }
    return 0;
}
```

```
JonathantekiiMac:input Jonathan$ clang -c ch12_eh.cpp -emit-llvm
-o ch12_eh.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch12_eh.bc -o -
```

./Ibdex/output/ch12_eh.ll

```
...
define dso_local i32 @_Z14test_try_catchv() #0 personality i8* bitcast (i32 (...))
* @_gxx_personality_v0 to i8*) {
entry:
...
invoke void @_Z15throw_exceptioni(i32 signext 2, i32 signext 1)
    to label %invoke.cont unwind label %lpad

invoke.cont:                                     ; preds = %entry
    br label %try.cont

lpad:                                              ; preds = %entry
    %0 = landingpad { i8*, i32 }
        catch i8* null
...
}
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch12_eh.bc -o -
.section .mdebug.abi32
.previous
.file "ch12_eh.bc"
```

(continues on next page)

(continued from previous page)

```
llc: /Users/Jonathan/llvm/test/llvm/lib/CodeGen/LiveVariables.cpp:133: void llvm::
LiveVariables::HandleVirtRegUse(unsigned int, llvm::MachineBasicBlock *, llvm
::MachineInstr *): Assertion `MRI->getVRegDef(reg) && "Register use before
def!"' failed.
```

A description of the C++ exception table formats can be found here².

For details about the LLVM IR used in exception handling, please refer to¹.

Chapter12_1 supports the LLVM IRs that correspond to the C++ **try** and **catch** keywords. It can compile `ch12_eh.bc` as follows:

Ibdex/chapters/Chapter12_1/Cpu0ISelLowering.h

```
/// If a physical register, this returns the register that receives the
/// exception address on entry to an EH pad.
Register
getExceptionPointerRegister(const Constant *PersonalityFn) const override {
    return Cpu0::A0;
}

/// If a physical register, this returns the register that receives the
/// exception typeid on entry to a landing pad.
Register
getExceptionSelectorRegister(const Constant *PersonalityFn) const override {
    return Cpu0::A1;
}
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch12_eh.bc -o -
```

./Ibdex/output/ch12_eh.cpu0.s

```
.type _Z14test_try_catchv,@function
.ent _Z14test_try_catchv          # @_Z14test_try_catchv
_Z14test_try_catchv:
...
$tmp0:
    addiu $4, $zero, 2
    addiu $5, $zero, 1
    jsub _Z15throw_exceptionii
    nop
$tmp1:
# %bb.1:                      # %invoke.cont
    jmp $BB1_4
$BB1_2:                         # %lpad
$tmp2:
    st $4, 16($fp)
    st $5, 12($fp)
# %bb.3:                      # %catch
```

(continues on next page)

² <http://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf>

¹ <http://llvm.org/docs/ExceptionHandling.html>

(continued from previous page)

```
ld $4, 16($fp)
jsub __cxa_begin_catch
nop
addiu $2, $zero, 1
st $2, 20($fp)
jsub __cxa_end_catch
nop
jmp $BB1_5
$BB1_4:                                # %try.cont
    addiu $2, $zero, 0
    st $2, 20($fp)
$BB1_5:                                # %return
    ld $2, 20($fp)
...

```

12.2 Thread variable

C++ support thread variable as the following file ch12_thread_var.cpp.

Ibdex/input/ch12_thread_var.cpp

```
__thread int a = 0;
thread_local int b = 0; // need option -std=c++11
int test_thread_var()
{
    a = 2;
    return a;
}

int test_thread_var_2()
{
    b = 3;
    return b;
}
```

While a global variable is a single instance shared by all threads in a process, a thread-local variable has a separate instance for each thread in the process. The same thread accesses the same instance of the thread-local variable, while different threads have their own instances with the same variable name³.

To support thread-local variables, symbols such as **tlsgd**, **tlsldm**, **dtp_hi**, **dtp_lo**, **gottp**, **tp_hi**, and **tp_lo** must be handled in both *evaluateRelocExpr()* of *Cpu0AsmParser.cpp* and *printImpl()* of *Cpu0MCExpr.cpp*.

Most of these symbols are used for relocation record handling, because the actual thread-local storage is created by the OS or language runtime that supports multi-threaded programming.

³ http://en.wikipedia.org/wiki/Thread-local_storage

Ibdex/chapters/Chapter12_1/MCTargetDesc/Cpu0AsmBackend.cpp

```

const MCFixupKindInfo &Cpu0AsmBackend::  

getFixupKindInfo(MCFixupKind Kind) const {  

    unsigned JSUBReloRec = 0;  

    if (HasLLD) {  

        JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel;  

    }  

    else {  

        JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel | MCFixupKindInfo::FKF_Constant;  

    }  

    const static MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds] = {  

        // This table *must* be in same the order of fixup_* kinds in  

        // Cpu0FixupKinds.h.  

        //  

        // name          offset  bits  flags  

        { "fixup_Cpu0_TLSGD",      0,     16,   0 },  

        { "fixup_Cpu0_GOTTP",      0,     16,   0 },  

        { "fixup_Cpu0_TP_HI",      0,     16,   0 },  

        { "fixup_Cpu0_TP_LO",      0,     16,   0 },  

        { "fixup_Cpu0_TLSLDM",      0,     16,   0 },  

        { "fixup_Cpu0_DTP_HI",      0,     16,   0 },  

        { "fixup_Cpu0_DTP_LO",      0,     16,   0 },  

        ...  

    };  

    ...
}

```

Ibdex/chapters/Chapter12_1/MCTargetDesc/Cpu0BaseInfo.h

```

namespace Cpu0II {  

    /// Target Operand Flag enum.  

    enum TOF {  

        //=====//  

        // Cpu0 Specific MachineOperand flags.  

        /// MO_TLSGD - Represents the offset into the global offset table at which  

        // the module ID and TSL block offset reside during execution (General  

        // Dynamic TLS).  

        MO_TLSGD,  

        /// MO_TLSLDM - Represents the offset into the global offset table at which  

        // the module ID and TSL block offset reside during execution (Local  

        // Dynamic TLS).  

        MO_TLSLDM,  

        MO_DTP_HI,  

        MO_DTP_LO,  

        /// MO_GOTTPREL - Represents the offset from the thread pointer (Initial
    }
}
```

(continues on next page)

(continued from previous page)

```
// Exec TLS).
MO_GOTTPREL,
/// MO_TPREL_HI/LO - Represents the hi and low part of the offset from
// the thread pointer (Local Exec TLS).
MO_TP_HI,
MO_TP_LO,
```

```
...
};

...
```

[Index/chapters/Chapter12_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp](#)

```
unsigned Cpu0ELFObjectWriter::getRelocType(MCContext &Ctx,
                                         const MCValue &Target,
                                         const MCFixup &Fixup,
                                         bool IsPCRel) const {
    // determine the type of the relocation
    unsigned Type = (unsigned)ELF::R_CPU0_NONE;
    unsigned Kind = (unsigned)Fixup.getKind();

    switch (Kind) {

case Cpu0::fixup_Cpu0_TLSDG:
    Type = ELF::R_CPU0_TLS_GD;
    break;
case Cpu0::fixup_Cpu0_GOTTPREL:
    Type = ELF::R_CPU0_TLS_GOTTPREL;
    break;

    ...
}
```

[Index/chapters/Chapter12_1/MCTargetDesc/Cpu0FixupKinds.h](#)

```
enum Fixups {

    // resulting in - R_CPU0_TLS_GD.
    fixup_Cpu0_TLSDG,

    // resulting in - R_CPU0_TLS_GOTTPREL.
    fixup_Cpu0_GOTTPREL,

    // resulting in - R_CPU0_TLS_TPREL_HI16.
    fixup_Cpu0_TP_HI,

    // resulting in - R_CPU0_TLS_TPREL_LO16.
    fixup_Cpu0_TP_LO,
```

(continues on next page)

(continued from previous page)

```
// resulting in = R_CPU0_TLS_LDM.  
fixup_Cpu0_TLSLDM,  
  
// resulting in = R_CPU0_TLS_DTP_HI16.  
fixup_Cpu0_DTP_HI,  
  
// resulting in = R_CPU0_TLS_DTP_LO16.  
fixup_Cpu0_DTP_LO,  
  
...  
};
```

Ibdex/chapters/Chapter12_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::  
getExprOpValue(const MCExpr *Expr, SmallVectorImpl<MCFixup> &Fixups,  
               const MCSubtargetInfo &STI) const {  
  
    case Cpu0MCExpr::CEK_TLSGD:  
        FixupKind = Cpu0::fixup_Cpu0_TLSGD;  
        break;  
    case Cpu0MCExpr::CEK_TLSLDM:  
        FixupKind = Cpu0::fixup_Cpu0_TLSLDM;  
        break;  
    case Cpu0MCExpr::CEK_DTP_HI:  
        FixupKind = Cpu0::fixup_Cpu0_DTP_HI;  
        break;  
    case Cpu0MCExpr::CEK_DTP_LO:  
        FixupKind = Cpu0::fixup_Cpu0_DTP_LO;  
        break;  
    case Cpu0MCExpr::CEK_GOTTPREL:  
        FixupKind = Cpu0::fixup_Cpu0_GOTTPREL;  
        break;  
    case Cpu0MCExpr::CEK_TP_HI:  
        FixupKind = Cpu0::fixup_Cpu0_TP_HI;  
        break;  
    case Cpu0MCExpr::CEK_TP_LO:  
        FixupKind = Cpu0::fixup_Cpu0_TP_LO;  
        break;  
  
    ...  
}
```

Ibdex/chapters/Chapter12_1/Cpu0InstrInfo.td

```
// TlsGd node is used to handle General Dynamic TLS  
def Cpu0TlsGd : SDNode<"Cpu0ISD::TlsGd", SDTIntUnaryOp>;  
  
// TpHi and TpLo nodes are used to handle Local Exec TLS
```

(continues on next page)

(continued from previous page)

```
def Cpu0TpHi : SDNode<"Cpu0ISD::TpHi", SDTIntUnaryOp>;
def Cpu0TpLo : SDNode<"Cpu0ISD::TpLo", SDTIntUnaryOp>;
```

```
let Predicates = [Ch12_1] in {
def : Pat<(Cpu0Hi tglobaltlsaddr:$in), (LUI tglobaltlsaddr:$in)>;
}
```

```
let Predicates = [Ch12_1] in {
def : Pat<(Cpu0Lo tglobaltlsaddr:$in), (ORi ZERO, tglobaltlsaddr:$in)>;
}
```

```
let Predicates = [Ch12_1] in {
def : Pat<(add CPUREgs:$hi, (Cpu0Lo tglobaltlsaddr:$lo)),
           (ORi CPUREgs:$hi, tglobaltlsaddr:$lo)>;
}
```

```
let Predicates = [Ch12_1] in {
def : WrapperPat<tglobaltlsaddr, ORi, CPUREgs>;
}
```

Ibdex/chapters/Chapter12_1/Cpu0SelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                         const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
    setOperationAction(ISD::GlobalTLSAddress, MVT::i32, Custom);
```

```
    ...
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    switch (Op.getOpcode())
    {
```

```
        case ISD::GlobalTLSAddress: return lowerGlobalTLSAddress(Op, DAG);
```

```
        ...
    }
    ...
}
```

```
SDValue Cpu0TargetLowering::
lowerGlobalTLSAddress(SDValue Op, SelectionDAG &DAG) const
{
    // If the relocation model is PIC, use the General Dynamic TLS Model or
```

(continues on next page)

(continued from previous page)

```
// Local Dynamic TLS model, otherwise use the Initial Exec or
// Local Exec TLS Model.

GlobalAddressSDNode *GA = cast<GlobalAddressSDNode>(Op);
if (DAG.getTarget().Options.EmulatedTLS)
    return LowerToTLSEmulatedModel(GA, DAG);

SDLoc DL(GA);
const GlobalValue *GV = GA->getGlobal();
EVT PtrVT = getPointerTy(DAG.getDataLayout());

TLSModel::Model model = getTargetMachine().getTLSModel(GV);

if (model == TLSModel::GeneralDynamic || model == TLSModel::LocalDynamic) {
    // General Dynamic and Local Dynamic TLS Model.
    unsigned Flag = (model == TLSModel::LocalDynamic) ? Cpu0II::MO_TLSLDM
                                                       : Cpu0II::MO_TLSGD;

    SDValue TGA = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0, Flag);
    SDValue Argument = DAG.getNode(Cpu0ISD::Wrapper, DL, PtrVT,
                                    getGlobalReg(DAG, PtrVT), TGA);
    unsigned PtrSize = PtrVT.getSizeInBits();
    IntegerType *PtrTy = Type::getIntNTy(*DAG.getContext(), PtrSize);

    SDValue TlsGetAddr = DAG.getExternalSymbol("__tls_get_addr", PtrVT);

    ArgListTy Args;
    ArgListEntry Entry;
    Entry.Node = Argument;
    Entry.Ty = PtrTy;
    Args.push_back(Entry);

    TargetLowering::CallLoweringInfo CLI(DAG);
    CLI.setDebugLoc(DL).setChain(DAG.getEntryNode())
        .setCallee(CallingConv::C, PtrTy, TlsGetAddr, std::move(Args));
    std::pair<SDValue, SDValue> CallResult = LowerCallTo(CLI);

    SDValue Ret = CallResult.first;

    if (model != TLSModel::LocalDynamic)
        return Ret;

    SDValue TGAHi = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                              Cpu0II::MO_DTP_HI);
    SDValue Hi = DAG getNode(Cpu0ISD::Hi, DL, PtrVT, TGAHi);
    SDValue TGALo = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                              Cpu0II::MO_DTP_LO);
    SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, PtrVT, TGALo);
    SDValue Add = DAG.getNode(ISD::ADD, DL, PtrVT, Hi, Ret);
    return DAG.getNode(ISD::ADD, DL, PtrVT, Add, Lo);
}
```

(continues on next page)

(continued from previous page)

```

SDValue Offset;
if (model == TLSModel::InitialExec) {
    // Initial Exec TLS Model
    SDValue TGA = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                              Cpu0II::MO_GOTTPREL);
    TGA = DAG.getNode(Cpu0ISD::Wrapper, DL, PtrVT, getGlobalReg(DAG, PtrVT),
                      TGA);
    Offset =
        DAG.getLoad(PtrVT, DL, DAG.getEntryNode(), TGA, MachinePointerInfo());
} else {
    // Local Exec TLS Model
    assert(model == TLSModel::LocalExec);
    SDValue TGAHi = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                                Cpu0II::MO_TP_HI);
    SDValue TGALo = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                                Cpu0II::MO_TP_LO);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, PtrVT, TGAHi);
    SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, PtrVT, TGALo);
    Offset = DAG.getNode(ISD::ADD, DL, PtrVT, Hi, Lo);
}
return Offset;
}

```

Ibdex/chapters/Chapter12_1/Cpu0ISelLowering.h

```
SDValue lowerGlobalTLSAddress(SDValue Op, SelectionDAG &DAG) const;
```

Ibdex/chapters/Chapter12_1/Cpu0MCInstLower.cpp

```

MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                               MachineOperandType MOTY,
                                               unsigned Offset) const {
    MCSymbolRefExpr::VariantKind Kind = MCSymbolRefExpr::VK_None;
    Cpu0MCE Expr::Cpu0ExprKind TargetKind = Cpu0MCE Expr::CEK_None;
    const MCSymbol *Symbol;

    switch (MO.getTargetFlags()) {

```

```

        case Cpu0II::MO_TLSGD:
            TargetKind = Cpu0MCE Expr::CEK_TLSGD;
            break;
        case Cpu0II::MO_TLSLDM:
            TargetKind = Cpu0MCE Expr::CEK_TLSLDM;
            break;
        case Cpu0II::MO_DTP_HI:
            TargetKind = Cpu0MCE Expr::CEK_DTP_HI;
            break;
        case Cpu0II::MO_DTP_LO:
            TargetKind = Cpu0MCE Expr::CEK_DTP_LO;
            break;
        case Cpu0II::MO_GOTTPREL:
    }
}
```

(continues on next page)

(continued from previous page)

```
TargetKind = Cpu0MCExpr::CEK_GOTTPREL;
break;
case Cpu0II::MO_TP_HI:
    TargetKind = Cpu0MCExpr::CEK_TP_HI;
    break;
case Cpu0II::MO_TP_LO:
    TargetKind = Cpu0MCExpr::CEK_TP_LO;
    break;
```

```
...
}
...
}
```

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch12_thread_var.cpp -emit-llvm -std=c++11 -o ch12_thread_var.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch12_thread_var.bc -o -
```

./ldex/output/ch12_thread_var.ll

```
...
@a = dso_local thread_local global i32 0, align 4
@b = dso_local thread_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone mustprogress
define dso_local i32 @_Z15test_thread_var() #0 {
entry:
    store i32 2, i32* @a, align 4
    %0 = load i32, i32* @a, align 4
    ret i32 %0
}

; Function Attrs: noinline nounwind optnone mustprogress
define dso_local i32 @_Z17test_thread_var_2v() #0 {
entry:
    store i32 3, i32* @b, align 4
    %0 = load i32, i32* @b, align 4
    ret i32 %0
}
...
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch12_thread_var.bc
-o ch12_thread_var.cpu0.pic.s
JonathantekiiMac:input Jonathan$ cat ch12_thread_var.cpu0.pic.s
```

./lbdex/output/ch12_thread_var.cpu0.pic.s

```
...
.ent _Z15test_thread_varv      # @_Z15test_thread_varv
_Z15test_thread_varv:
...
ori $4, $gp, %tlsldm(a)
ld $t9, %call16(__tls_get_addr)($gp)
jalr $t9
nop
ld $gp, 8($fp)
lui $3, %dtp_hi(a)
addu $2, $3, $2
ori $2, $2, %dtp_lo(a)
...
```

In PIC (Position-Independent Code) mode, the *thread* variable is accessed by calling the function *__tls_get_addr* with the address of the thread-local variable as an argument.

For C++11 *thread_local* variables, the compiler generates a call to the function *_ZTW1b*, which internally calls *__tls_get_addr* to retrieve the address of the *thread_local* variable.

In static mode, thread-local variables are accessed directly by loading their addresses using machine instructions. For example, variables *a* and *b* are accessed through direct address calculation instructions.

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm
ch12_thread_var.bc -o ch12_thread_var.cpu0.static.s
JonathantekiiMac:input Jonathan$ cat ch12_thread_var.cpu0.static.s
```

./lbdex/output/ch12_thread_var.cpu0.static.s

```
...
lui $2, %tp_hi(a)
ori $2, $2, %tp_lo(a)
...
lui $2, %tp_hi(b)
ori $2, $2, %tp_lo(b)
...
```

While MIPS uses the *rdhwr* instruction to access thread-local variables, Cpu0 accesses thread-local variables without introducing any new instructions.

Thread-local variables in Cpu0 are stored in a dedicated thread-local memory region, which is accessed through *%tp_hi* and *%tp_lo*. This memory section is protected by the kernel, meaning it can only be accessed in kernel mode.

As a result, user-mode programs cannot access this memory region, leaving no room for potential exploits or malicious programs to interfere with thread-local storage.

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=mips -relocation-model=static -filetype=asm
ch12_thread_var.bc -o -
...
lui $1, %tprel_hi(a)
ori $1, $1, %tprel_lo(a)
```

(continues on next page)

(continued from previous page)

```
.set push
.set mips32r2
rdhwr $3, $29
.set pop
addu $1, $3, $1
addiu $2, $zero, 2
sw $2, 0($1)
addiu $2, $zero, 2
...
```

In static mode, the thread variable is similar to global variable. In general, they are same in IRs, DAGs and machine code translation. List them in the following tables. You can check them with debug option enabled.

In static mode, the thread variable behaves similarly to a global variable. In general, they are the same in terms of LLVM IR, DAG, and machine code translation.

You can refer to the following tables for a detailed comparison.

To observe this in action, compile and check with debug options enabled.

Table 12.1: The DAGs of thread variable of static mode

stage	DAG
IR	load i32* @a, align 4;
Legalized selection DAG	(add Cpu0ISD::Hi Cpu0ISD::Lo);
Instruction Selection	ori \$2, \$zero, %tp_lo(a);
•	lui \$3, %tp_hi(a);
•	addu \$3, \$3, \$2;

Table 12.2: The DAGs of local_thread variable of static mode

stage	DAG
IR	ret i32* @b;
Legalized selection DAG	%0=(add Cpu0ISD::Hi Cpu0ISD::Lo);...
Instruction Selection	ori \$2, \$zero, %tp_lo(a);
•	lui \$3, %tp_hi(a);
•	addu \$3, \$3, \$2;

12.3 C++ Memory Order⁴⁵²

12.3.1 Source Code Compatibility

Memory Order:

- Memory Order is the rules that define how operations on shared memory appear to multiple threads — especially the ordering of reads/writes.
- These rules ensure correct execution in parallel programs.
- C++ Memory Order makes “source code compatible” across different platforms as shown below.

⁴ https://en.cppreference.com/w/cpp/atomic/memory_order

⁵ https://en.wikipedia.org/wiki/Memory_model_%28programming%29

⁶ <http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g>

```

volatile bool ready(false);

void producer() {
    // Populate data
    for (int i = 0; i < 10; ++i) {
        data.push_back(i * 10);
    }
    asm("__sync__"); // rely on sync or barrier in CPU instruction.
    // Release store: Ensures all writes to 'data' are visible before 'ready' is set.
    ready = true; // Release signal
}

void consumer() {
    // Acquire load: Ensures all writes that happened before the release store are
    // visible.
    while (!ready); // Wait for ready signal

    // Ensure that this print sees the correct value of `data`
    for (int i = 0; i < 10; ++i) {
        std::cout << data[i];
    }
}

```

Diagram Explanation:

- **CPU Core 1 (Thread 1 - Producer)**
 - Writes all elements to ‘data’(not immediately visible).
 - `asm("__sync__")` ensuring prior stores are visible before writing true to ready, “`ready = true`”.
 - * All memory access instructions (load/store) must be completed after `asm("__sync__")`.
- **Main Memory**
 - ‘`ready=true`’ propagates to main memory, making it visible to all cores.
- **CPU Core 2 (Thread 2 - Consumer)**
 - Waits until ‘`ready=true`’, ensuring visibility of all previous writes.
 - After acquiring ready, output all elements from ‘data’, which are now reliably published by the producer thread..

Above inline assembly `asm("__sync__")` has the following problem:

No standard atomic operations → source code incompatible.

- Developers had to use compiler-specific intrinsics (like GCC’s `__sync_*`, MSVC’s `Interlocked*`) or inline assembly, making source code incompatible.

The following C++11 example solve the problem of source code incompatible.

References/c++/mem-order-ex1.cpp (C++ code of memory order for producer-consumer)

```
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>
```

(continues on next page)

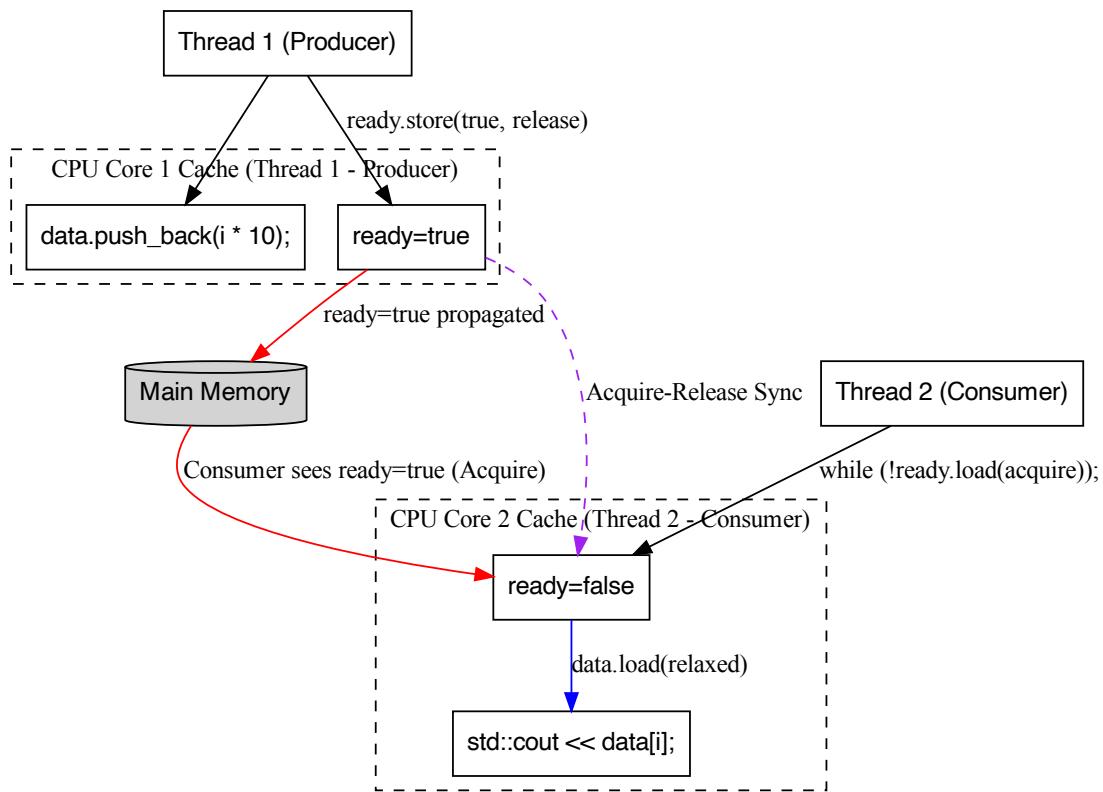


Fig. 12.1: Diagram for mem-order-ex1.cpp

(continued from previous page)

```
std::atomic<int> data(0);
std::atomic<bool> ready(false);

void producer() {
    // Populate data
    for (int i = 0; i < 10; ++i) {
        data.push_back(i * 10);
    }
    // Release store: Ensures all writes to 'data' are visible before 'ready' is set.
    ready.store(true, std::memory_order_release); // Release signal
}

void consumer() {
    // Acquire load: Ensures all writes that happened before the release store are visible.
    while (!ready.load(std::memory_order_acquire)); // Wait for ready signal

    // Ensure that this print sees the correct value of `data`
    for (int i = 0; i < 10; ++i) {
        std::cout << data[i];
    }
}

int main() {
    std::thread t1(producer);
    std::thread t2(consumer);

    t1.join();
    t2.join();

    return 0;
}
```

- `memory_order::release`

Same as `asm("__sync__")`, read and write operations appearing before the operation in the program must occur before it and are completed after `"ready.store(true, std::memory_order_release)"`.

- `memory_order::acquire`

Same as load volatile variable `"ready"`, read and write operations appearing after the operation in the program must occur after it (i.e., they cannot be re-ordered before the operation, `"while (!ready.load(std::memory_order_acquire))"`).

Summary:

C++ Memory Order makes “**source code compatible**” across different platforms.

12.3.2 The Problem Before C++11

Before **C++11**, multi-threaded programming relied on **mutexes**, **volatile variables**, and **platform-specific atomic operations** (such as `atomic_load(&a)` and `atomic_store(&a, 42)`), which often led to inefficiencies and undefined behavior.

For RISC CPUs, **only load/store instructions access memory**. Atomic instructions ensure memory consistency across multiple cores.

CPUs provide atomic operations such as **compare-and-swap**⁷ or ll/sc (load-linked/store-conditional), along with **BARRIER** or **SYNC** instructions to enforce memory ordering. However, **C++03 did not have a language feature to tell the compiler how to control memory order for load/store instructions**.

To address this, **C++11 introduced memory orderings via `std::atomic`**, giving programmers **fine-grained control** over synchronization and memory consistency.

1. No standard atomic operations → source code incompatible.

- Developers had to use compiler-specific intrinsics (like GCC's `__sync_*`, MSVC's `Interlocked*`) or inline assembly, making source code incompatible.

2. Unspecified Behavior in Multi-threading

- The C++98/03 standard had **no formal memory model**.
- Compilers **optimized code aggressively**, leading to race conditions.

3. Reliance on Volatile and Platform-specific Primitives

- volatile* **did not** prevent reordering by the compiler.
 - Though some compilers may choose to avoid reordering around volatile accesses.
- Programmers had to use **OS-specific APIs** (e.g., `'pthread_mutex'`).

4. Inefficient Synchronization Mechanisms

- Mutexes ensured correctness but **caused performance overhead**.
- Spin-locks wasted CPU cycles** due to busy-waiting.

12.3.3 C++11 Memory Model Solution

Note

Non-blocking algorithm: In computer science, an algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread.

If a suspended thread can temporarily release its mutex and reacquire it upon resuming, while always producing correct results, then the algorithm is non-blocking.

Wait-free: no starvation.

An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. In other words, wait-free algorithm has no starvation.

Lock-free: progressive, allow starvation.

Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress (for some sensible definition of progress).

⁷ <https://en.wikipedia.org/wiki/Compare-and-swap>

All wait-free algorithms are lock-free. In particular, if one thread is suspended, then a lock-free algorithm guarantees that the remaining threads can still make progress. Hence, if two threads can contend for the same mutex lock or spinlock, then the algorithm is not lock-free. (If we suspend one thread that holds the lock, then the second thread will block.)⁸.

Based on this definition, any algorithm that retains a mutex without allowing temporary release does not qualify as lock-free.

C++11 introduced a **well-defined memory model** and **atomic operations** with **memory orderings**, allowing programmers to control hardware-level optimizations.

Table 12.3: C++ memory order A

Feature	Description	Benefit
<code>std::atomic</code>	Provides lock-free atomic variables	Faster than mutexes
Memory Orderings (<code>std::memory_order</code>)	Controls instruction reordering	Fine-grained optimization
Sequential Consistency (<code>memory_order_seq_cst</code>)	Strongest ordering, default behavior	Prevents race conditions
Acquire-Release (<code>memory_order_acquire/release</code>)	Synchronization without mutexes	Efficient producer-consumer
Relaxed Ordering (<code>memory_order_relaxed</code>)	Allows reordering for performance	Best for atomic counters

Table 12.4: C++ memory order B

Memory Order	Description	Use Cases
<code>memory_order_relaxed</code>	No ordering guarantees; only atomicity.	Non-dependent atomic counters, statistics.
<code>memory_order_consume</code>	Data-dependent ordering (deprecated in practice).	Rarely used; intended for pointer chains.
<code>memory_order_acquire</code>	Ensures preceding reads/writes are visible.	Locks, consumer threads.
<code>memory_order_release</code>	Ensures following reads/writes are visible.	Locks, producer threads.
<code>memory_order_acq_rel</code>	Combines acquire + release.	Read-modify-write operations, synchronization.
<code>memory_order_seq_cst</code>	Strongest ordering; global sequential consistency.	Default behavior, safest but can be slow.

Explanation:

1. **Sequential Consistency** (``memory_order_seq_cst``)
 - Prevents reordering globally.
 - Ensures all threads observe operations in the same order.
 - Default behavior of `std::atomic`.
 - Provides **global order of operations**, preventing out-of-order execution.
 - The safest but can cause **performance overhead**.
2. **Acquire-Release** (``memory_order_acquire/release``)
 -

⁸ https://en.wikipedia.org/wiki/Non-blocking_algorithm

- Efficient alternative to mutexes.
- *acquire*: Ensures earlier loads are visible.
- *release*: Ensures later stores are visible.

3. Relaxed Ordering (`memory_order_relaxed`)

- Allows **maximum performance** without ordering constraints.
- Best for **counters and statistics** that don't need synchronization.
- Example: **Atomic counters** that don't require ordering.

4. `memory_order_acq_rel`

- Used in **atomic read-modify-write operations** like *fetch_add*.
- Ensures proper ordering in concurrent updates.

Summary:

- Use `memory_order_relaxed` for **performance** when ordering is unnecessary.
- Use `memory_order_acquire/release` for **synchronization** between threads.
- Use `memory_order_seq_cst` when you need **global ordering but at a performance cost**.

12.4 Cpu0 implementation for memory-order

In order to support atomic in C++ and java, llvm provides the atomic IRs and memory ordering here⁹¹⁰.

The chapter 19 of book DPC++¹¹ explains the memory ordering better and I add the related code fragment of libdex/input/atomics.ll to it for explanation as follows,

- `memory_order::relaxed`

Read and write operations can be re-ordered before or after the operation with no restrictions. There are no ordering guarantees.

```
define i8 @load_i8_unordered(i8* %mem) {
; CHECK-LABEL: load_i8_unordered
; CHECK: ll
; CHECK: sc
; CHECK-NOT: sync
%val = load atomic i8, i8* %mem unordered, align 1
ret i8 %val
}
```

No `sync` from CodeGen instructions above.

- `memory_order::acquire`

Read and write operations appearing after the operation in the program must occur after it (i.e., they cannot be re-ordered before the operation).

⁹ <http://llvm.org/docs/Atomics.html>

¹⁰ <http://llvm.org/docs/LangRef.html#ordering>

¹¹ Section “The `memory_order` Enumeration Class” which include figure 19-10 of book <https://link.springer.com/book/10.1007/978-1-4842-5574-2>

```
define i32 @load_i32_acquire(i32* %mem) {
; CHECK-LABEL: load_i32_acquire
; CHECK: ll
; CHECK: sc
%val = load atomic i32, i32* %mem acquire, align 4
; CHECK: sync
ret i32 %val
}
```

Sync guarantees “load atomic” complete before the next R/W (Read/Write). All writes in other threads that release the same atomic variable are visible in the current thread.

- `memory_order::release`

Read and write operations appearing before the operation in the program must occur before it (i.e., they cannot be re-ordered after the operation), and preceding write operations are guaranteed to be visible to other program instances which have been synchronized by a corresponding acquire operation (i.e., an atomic operation using the same variable and `memory_order::acquire` or a barrier function).

```
define void @store_i32_release(i32* %mem) {
; CHECK-LABEL: store_i32_release
; CHECK: sync
; CHECK: ll
; CHECK: sc
store atomic i32 42, i32* %mem release, align 4
ret void
}
```

Sync guarantees preceding R/W complete before “store atomic”. Mips’ ll and sc guarantee that “store atomic release” is visible to other processors.

- `memory_order::acq_rel`

The operation acts as both an acquire and a release. Read and write operations cannot be re-ordered around the operation, and preceding writes must be made visible as previously described for `memory_order::release`.

```
define i32 @cas_strong_i32_acqrel_acquire(i32* %mem) {
; CHECK-LABEL: cas_strong_i32_acqrel_acquire
; CHECK: ll
; CHECK: sc
%val = cmpxchg i32* %mem, i32 0, i32 1 acq_rel acquire
; CHECK: sync
%loaded = extractvalue { i32, i1} %val, 0
ret i32 %loaded
}
```

Sync guarantees preceding R/W complete before “`cmpxchg`”. Other processors’ preceding write operations are guaranteed to be visible to this “`cmpxchg` acquire” (Mips’s ll and sc guarantee it).

- `memory_order::seq_cst`

The operation acts as an acquire, release, or both depending on whether it is a read, write, or read-modify-write operation, respectively. All operations with this memory order are observed in a sequentially consistent order.

```
define i8 @cas_strong_i8_sc_sc(i8* %mem) {
; CHECK-LABEL: cas_strong_i8_sc_sc
```

(continues on next page)

(continued from previous page)

```

; CHECK: sync
; CHECK: ll
; CHECK: sc
%val = cmpxchg i8* %mem, i8 0, i8 1 seq_cst seq_cst
; CHECK: sync
%loaded = extractvalue { i8, i1 } %val, 0
ret i8 %loaded
}

```

First sync guarantees preceding R/W complete before “cmpxchg seq_cst” and visible to “cmpxchg seq_cst”. For seq_cst, a store performs a release operation. Which means “cmpxchg seq_cst” are visible to other threads/processors that acquire the same atomic variable as the memory_order_release definition. Mips’ll and sc guarantees this feature of “cmpxchg seq_cst”. Second Sync guarantees “cmpxchg seq_cst” complete before the next R/W.

There are several restrictions on which memory orders are supported by each operation. Fig. 12.2 (from book Figure 19-10) summarizes which combinations are valid.

Functions	Supported memory_order Values				
	relaxed	acquire	release	acq_rel	seq_cst
load	✓	✓	✗	✗	✓
store	✓	✗	✓	✗	✓
exchange					
compare_exchange_*	✓	✓	✓	✓	✓
fetch_*					
fence	✓	✓	✓	✓	✓

Figure 19-10. Supporting atomic operations with memory_order

Fig. 12.2: Supporting atomic operations with memory_order

Load operations do not write values to memory and are therefore incompatible with release semantics. Similarly, store operations do not read values from memory and are therefore incompatible with acquire semantics. The remaining read-modify-write atomic operations and fences are compatible with all memory orderings^{Page 613, 11}.

Feature	Standard C++	SYCL / DPC++
Atomic Objects	<code>std::atomic</code>	Not available.
Atomic References	<code>std::atomic_ref</code> (C++20 onwards)	<code>sycl::atomic_ref</code>
Memory Orders	<code>relaxed</code> <code>consume</code> <code>acquire</code> <code>release</code> <code>scq_rel</code> <code>seq_cst</code>	<code>relaxed</code> <code>acquire</code> <code>release</code> <code>scq_rel</code> <code>seq_cst</code>
Memory Scopes	Not available. Behavior of atomics and fences matches DPC++ <code>system</code> scope.	<code>work_item</code> <code>sub_group</code> <code>work_group</code> <code>device</code> <code>system</code>
Fences	<code>std::atomic_thread_fence</code>	<code>sycl::atomic_fence</code>
Barriers	<code>std::barrier</code> (C++20 onwards)	<code>nd_item::barrier</code> <code>sub_group::barrier</code>
Address Spaces	All memory is in a single (host) address space.	Host Device (Global) Device (Local) Device (Private) Shared (USM)

Figure 19-9. Comparing standard C++ and SYCL/DPC++ memory models

Fig. 12.3: Comparing standard C++ and SYCL/DPC++ memory models

Note**C++ memory_order_consume**

The C++ memory model additionally includes `memory_order::consume`, with similar behavior to `memory_order::acquire`. However, the C++17 standard discourages its use, noting that its definition is being revised. Its inclusion in dpC++ has therefore been postponed to a future version.

For a few years now, compilers have treated `consume` as a synonym for `acquire`¹².

The current expectation is that the replacement facility will rely on core memory model and atomics definitions very similar to what's currently there. Since `memory_order_consume` does have a profound impact on the memory model, removing this text would allow drastic simplification, but conversely would make it very difficult to add anything along the lines of `memory_order_consume` back in later, especially if the standard evolves in the meantime, as expected. Thus we are not proposing to remove the current wording¹³.

The following test files are extracted from `memory_checks()` in `clang/test/Sema/atomic-ops.c`. The `__c11_atomic_xxx` built-in functions used by Clang are defined in `clang/include/clang/Basic/Builtins.def`. Compiling these files with Clang produces the same results as shown in Fig. 12.2.

Note: Clang compiles `memory_order_consume` to the same result as `memory_order_acquire`.

lbdex/input/ch12_sema_atomic-ops.c

```
// clang -S ch12_sema_atomic-ops.c -emit-llvm -o -
// Uses /opt/homebrew/opt/llvm/bin/clang in macOS.

#include <stdatomic.h>

// From memory_checks() of Sema/atomic-ops.c
void memory_checks(_Atomic(int) *Ap, int *p, int val) {
    (void) __c11_atomic_load(Ap, memory_order_relaxed);
    (void) __c11_atomic_load(Ap, memory_order_acquire);
    (void) __c11_atomic_load(Ap, memory_order_consume);
    (void) __c11_atomic_load(Ap, memory_order_release); // expected-warning {{memory-
    ↪order argument to atomic operation is invalid}}
    (void) __c11_atomic_load(Ap, memory_order_acq_rel); // expected-warning {{memory-
    ↪order argument to atomic operation is invalid}}
    (void) __c11_atomic_load(Ap, memory_order_seq_cst);
    (void) __c11_atomic_load(Ap, val);
    (void) __c11_atomic_load(Ap, -1); // expected-warning {{memory order argument to
    ↪atomic operation is invalid}}
    (void) __c11_atomic_load(Ap, 42); // expected-warning {{memory order argument to
    ↪atomic operation is invalid}}

    (void) __c11_atomic_store(Ap, val, memory_order_relaxed);
    (void) __c11_atomic_store(Ap, val, memory_order_acquire); // expected-warning {
    ↪{{memory order argument to atomic operation is invalid}}
    (void) __c11_atomic_store(Ap, val, memory_order_consume); // expected-warning {
    ↪{{memory order argument to atomic operation is invalid}}
    (void) __c11_atomic_store(Ap, val, memory_order_release);
    (void) __c11_atomic_store(Ap, val, memory_order_acq_rel); // expected-warning {
```

(continues on next page)

¹² <https://stackoverflow.com/questions/65336409/what-does-memory-order-consume-really-do>

¹³ <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r1.html>

(continued from previous page)

```

→{memory order argument to atomic operation is invalid}
(void) __c11_atomic_store(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_exchange(Ap, val, memory_order_relaxed);
(void) __c11_atomic_exchange(Ap, val, memory_order_acquire);
(void) __c11_atomic_exchange(Ap, val, memory_order_consume);
(void) __c11_atomic_exchange(Ap, val, memory_order_release);
(void) __c11_atomic_exchange(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_exchange(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_relaxed, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_acquire, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_consume, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_release, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_acq_rel, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_seq_cst, memory_
→order_relaxed);

(void) __c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_relaxed, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_acquire, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_consume, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_release, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_acq_rel, memory_
→order_relaxed);
(void) __c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_seq_cst, memory_
→order_relaxed);

atomic_thread_fence(memory_order_relaxed);
atomic_thread_fence(memory_order_acquire);
atomic_thread_fence(memory_order_consume); // For a few years now, compilers have_
→treated consume as a synonym for acquire.
atomic_thread_fence(memory_order_release);
atomic_thread_fence(memory_order_acq_rel);
atomic_thread_fence(memory_order_seq_cst);
atomic_signal_fence(memory_order_seq_cst);
}

}

```

Index/input/ch12_sema_atomic-fetch.c

```
// clang -S ch12_sema_atomic-fetch.c -emit-llvm -o -
// Uses /opt/homebrew/opt/llvm/bin/clang in macOS.
```

(continues on next page)

(continued from previous page)

```
#include <stdatomic.h>

//#define WANT_COMPILE_FAIL

// From __c11_atomic_fetch_xxx of memory_checks() of Sema/atomic-ops.c
void memory_checks(_Atomic(int) *Ap, int *p, int val) {
    (void) __c11_atomic_fetch_add(Ap, 1, memory_order_relaxed);
    (void) __c11_atomic_fetch_add(Ap, 1, memory_order_acquire);
    (void) __c11_atomic_fetch_add(Ap, 1, memory_order_consume);
    (void) __c11_atomic_fetch_add(Ap, 1, memory_order_release);
    (void) __c11_atomic_fetch_add(Ap, 1, memory_order_acq_rel);
    (void) __c11_atomic_fetch_add(Ap, 1, memory_order_seq_cst);

#endif // fail to compile:
(void) __c11_atomic_fetch_add(
    (struct Incomplete * _Atomic *)0, // expected-error {{incomplete type 'struct
    ↪Incomplete'}}
    1, memory_order_seq_cst);
#endif

(void) __c11_atomic_init(Ap, val);

(void) __c11_atomic_fetch_sub(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_sub(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_sub(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_sub(Ap, val, memory_order_release);
(void) __c11_atomic_fetch_sub(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_sub(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_fetch_and(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_and(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_and(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_and(Ap, val, memory_order_release);
(void) __c11_atomic_fetch_and(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_and(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_fetch_or(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_or(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_or(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_or(Ap, val, memory_order_release);
(void) __c11_atomic_fetch_or(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_or(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_fetch_xor(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_xor(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_xor(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_xor(Ap, val, memory_order_release);
```

(continues on next page)

(continued from previous page)

```

(void) __c11_atomic_fetch_xor(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_xor(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_fetch_nand(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_nand(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_nand(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_nand(Ap, val, memory_order_release);
(void) __c11_atomic_fetch_nand(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_nand(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_fetch_min(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_min(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_min(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_min(Ap, val, memory_order_release);
(void) __c11_atomic_fetch_min(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_min(Ap, val, memory_order_seq_cst);

(void) __c11_atomic_fetch_max(Ap, val, memory_order_relaxed);
(void) __c11_atomic_fetch_max(Ap, val, memory_order_acquire);
(void) __c11_atomic_fetch_max(Ap, val, memory_order_consume);
(void) __c11_atomic_fetch_max(Ap, val, memory_order_release);
(void) __c11_atomic_fetch_max(Ap, val, memory_order_acq_rel);
(void) __c11_atomic_fetch_max(Ap, val, memory_order_seq_cst);
}

```

Table 12.5: Atomic related between clang's builtin and llvm ir

clang's builtin	llvm ir
<code>__c11_atomic_load</code>	<code>load atomic</code>
<code>__c11_atomic_store</code>	<code>store atomic</code>
<code>__c11_atomic_exchange_xxx</code>	<code>cmpxchg</code>
<code>atomic_thread_fence</code>	<code>fence</code>
<code>__c11_atomic_fetch_xxx</code>	<code>atomicrmw xxx</code>

C++ atomic functions are supported by calling implementation functions from the C++ standard library. These functions eventually call the `__c11_atomic_xxx` built-in functions for actual implementation.

Therefore, `__c11_atomic_xxx` functions, listed above, provide a lower-level and higher-performance interface for C++ programmers. An example is shown below:

Ibdex/input/ch12_c++_atomics.cpp

```

// ~/llvm/debug/build/bin/clang -S ch12_c++_atomics.cpp -emit-llvm -o -
// Uses /opt/homebrew/opt/llvm/bin/clang in macOS.

#include <atomic>

std::atomic<bool> winner (false);

int test_atomics() {
    int count = 0;

```

(continues on next page)

(continued from previous page)

```

bool res = winner.exchange(true);
if (res) count++;

return count;
}

```

To support LLVM atomic IR instructions, the following code is added to Chapter12_1.

Ibdex/chapters/Chapter12_1/Disassembler/Cpu0Disassembler.cpp

```

static DecodeStatus DecodeMem(MCInst &Inst,
                             unsigned Insn,
                             uint64_t Address,
                             const void *Decoder) {

    if(Inst.getOpcode() == Cpu0::SC) {
        Inst.addOperand(MCOperand::createReg(Reg));
    }

    ...
}

```

Ibdex/chapters/Chapter12_1/Cpu0InstrInfo.td

```

def SDT_Sync : SDTypeProfile<0, 1, [SDTCisVT<0, i32>]>;
def Cpu0Sync : SDNode<"Cpu0ISD::Sync", SDT_Sync, [SDNPHasChain]>;
def PtrRC : Operand<iPTR> {
    let MIOperandInfo = (ops ptr_rc);
    let DecoderMethod = "DecodeCPUREgsRegisterClass";
}

// Atomic instructions with 2 source operands (ATOMIC_SWAP & ATOMIC_LOAD_*).
class Atomic2Ops<PatFrag Op, RegisterClass DRC> :
    PseudoSE<(outs DRC:$dst), (ins PtrRC:$ptr, DRC:$incr),
    [(set DRC:$dst, (Op iPTR:$ptr, DRC:$incr))]>;

// Atomic Compare & Swap.
class AtomicCmpSwap<PatFrag Op, RegisterClass DRC> :
    PseudoSE<(outs DRC:$dst), (ins PtrRC:$ptr, DRC:$cmp, DRC:$swap),
    [(set DRC:$dst, (Op iPTR:$ptr, DRC:$cmp, DRC:$swap))]>;

class LLBase<bits<8> Opc, string opstring, RegisterClass RC, Operand Mem> :
    FMem<Opc, (outs RC:$ra), (ins Mem:$addr),
    !strconcat(opstring, "\t$ra, $addr"), [], IILoad> {
    let mayLoad = 1;
}

class SCBase<bits<8> Opc, string opstring, RegisterOperand RO, Operand Mem> :

```

(continues on next page)

(continued from previous page)

```
FMem<Opc, (outs RO:$dst), (ins RO:$ra, Mem:$addr),
    !strconcat(opstring, "\t$ra, $addr"), [], IIStore> {
let mayStore = 1;
let Constraints = "$ra = $dst";
}
```

```
let Predicates = [Ch12_1] in {
let usesCustomInserter = 1 in {
def ATOMIC_LOAD_ADD_I8      : Atomic2Ops<atomic_load_add_8, CPURegs>;
def ATOMIC_LOAD_ADD_I16     : Atomic2Ops<atomic_load_add_16, CPURegs>;
def ATOMIC_LOAD_ADD_I32     : Atomic2Ops<atomic_load_add_32, CPURegs>;
def ATOMIC_LOAD_SUB_I8      : Atomic2Ops<atomic_load_sub_8, CPURegs>;
def ATOMIC_LOAD_SUB_I16     : Atomic2Ops<atomic_load_sub_16, CPURegs>;
def ATOMIC_LOAD_SUB_I32     : Atomic2Ops<atomic_load_sub_32, CPURegs>;
def ATOMIC_LOAD_AND_I8      : Atomic2Ops<atomic_load_and_8, CPURegs>;
def ATOMIC_LOAD_AND_I16     : Atomic2Ops<atomic_load_and_16, CPURegs>;
def ATOMIC_LOAD_AND_I32     : Atomic2Ops<atomic_load_and_32, CPURegs>;
def ATOMIC_LOAD_OR_I8       : Atomic2Ops<atomic_load_or_8, CPURegs>;
def ATOMIC_LOAD_OR_I16     : Atomic2Ops<atomic_load_or_16, CPURegs>;
def ATOMIC_LOAD_OR_I32     : Atomic2Ops<atomic_load_or_32, CPURegs>;
def ATOMIC_LOAD_XOR_I8      : Atomic2Ops<atomic_load_xor_8, CPURegs>;
def ATOMIC_LOAD_XOR_I16     : Atomic2Ops<atomic_load_xor_16, CPURegs>;
def ATOMIC_LOAD_XOR_I32     : Atomic2Ops<atomic_load_xor_32, CPURegs>;
def ATOMIC_LOAD_NAND_I8      : Atomic2Ops<atomic_load_nand_8, CPURegs>;
def ATOMIC_LOAD_NAND_I16     : Atomic2Ops<atomic_load_nand_16, CPURegs>;
def ATOMIC_LOAD_NAND_I32     : Atomic2Ops<atomic_load_nand_32, CPURegs>;

def ATOMIC_SWAP_I8          : Atomic2Ops<atomic_swap_8, CPURegs>;
def ATOMIC_SWAP_I16         : Atomic2Ops<atomic_swap_16, CPURegs>;
def ATOMIC_SWAP_I32         : Atomic2Ops<atomic_swap_32, CPURegs>;

def ATOMIC_CMP_SWAP_I8      : AtomicCmpSwap<atomic_cmp_swap_8, CPURegs>;
def ATOMIC_CMP_SWAP_I16     : AtomicCmpSwap<atomic_cmp_swap_16, CPURegs>;
def ATOMIC_CMP_SWAP_I32     : AtomicCmpSwap<atomic_cmp_swap_32, CPURegs>;
}
}
```

```
let Predicates = [Ch12_1] in {
let hasSideEffects = 1 in
def SYNC : Cpu0Inst<(outs), (ins i32imm:$stype), "sync $stype",
    [(Cpu0Sync imm:$stype)], NoItinerary, FrmOther>
{
bits<5> stype;
let Opcode = 0x60;
let Inst{25-11} = 0;
let Inst{10-6} = stype;
let Inst{5-0} = 0;
}
}
```

```
/// Load-linked, Store-conditional
```

(continues on next page)

(continued from previous page)

```
def LL      : LLBase<0x61, "ll", CPURegs, mem>;
def SC      : SCBase<0x62, "sc", RegisterOperand<CPURegs>, mem>;
```

```
def : Cpu0InstAlias<"sync",
           (SYNC 0), 1>;
```

Ibdex/chapters/Chapter12_1/Cpu0ISelLowering.h

```
MachineBasicBlock *
EmitInstrWithCustomInserter(MachineInstr &MI,
                           MachineBasicBlock *MBB) const override;

SDValue lowerATOMIC_FENCE(SDValue Op, SelectionDAG& DAG) const;

bool shouldInsertFencesForAtomic(const Instruction *I) const override {
    return true;
}

/// Emit a sign-extension using shl/sra appropriately.
MachineBasicBlock *emitSignExtendToI32InReg(MachineInstr &MI,
                                              MachineBasicBlock *BB,
                                              unsigned Size, unsigned DstReg,
                                              unsigned SrcReg) const;
MachineBasicBlock *emitAtomicBinary(MachineInstr &MI, MachineBasicBlock *BB,
                                   unsigned Size, unsigned BinOpcode, bool Nand = false) const;
MachineBasicBlock *emitAtomicBinaryPartword(MachineInstr &MI,
                                            MachineBasicBlock *BB, unsigned Size, unsigned BinOpcode,
                                            bool Nand = false) const;
MachineBasicBlock *emitAtomicCmpSwap(MachineInstr &MI,
                                     MachineBasicBlock *BB, unsigned Size) const;
MachineBasicBlock *emitAtomicCmpSwapPartword(MachineInstr &MI,
                                             MachineBasicBlock *BB, unsigned Size) const;
```

Ibdex/chapters/Chapter12_1/Cpu0ISelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
```

```
case Cpu0ISD::Sync:           return "Cpu0ISD::Sync";
```

```
...
```

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
: TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
setOperationAction(ISD::ATOMIC_LOAD,          MVT::i32,      Expand);
setOperationAction(ISD::ATOMIC_LOAD,          MVT::i64,      Expand);
```

(continues on next page)

(continued from previous page)

```
setOperationAction(llvm::ISD::ATOMIC_STORE,           MVT::i32,      Expand);  
setOperationAction(llvm::ISD::ATOMIC_STORE,           MVT::i64,      Expand);
```

```
SDValue Cpu0TargetLowering::
```

```
LowerOperation(SDValue Op, SelectionDAG &DAG) const
```

```
{
```

```
    switch (Op.getOpcode())
```

```
{
```

```
    case llvm::ISD::ATOMIC_FENCE:      return lowerATOMIC_FENCE(Op, DAG);
```

```
    ...
```

```
}
```

```
MachineBasicBlock *
```

```
Cpu0TargetLowering::EmitInstrWithCustomInserter(MachineInstr &MI,  
                                               MachineBasicBlock *BB) const {
```

```
    switch (MI.getOpcode()) {
```

```
    default:
```

```
        llvm_unreachable("Unexpected instr type to insert");
```

```
    case Cpu0::ATOMIC_LOAD_ADD_I8:
```

```
        return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::ADDu);
```

```
    case Cpu0::ATOMIC_LOAD_ADD_I16:
```

```
        return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::ADDu);
```

```
    case Cpu0::ATOMIC_LOAD_ADD_I32:
```

```
        return emitAtomicBinary(MI, BB, 4, Cpu0::ADDu);
```

```
    case Cpu0::ATOMIC_LOAD_AND_I8:
```

```
        return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::AND);
```

```
    case Cpu0::ATOMIC_LOAD_AND_I16:
```

```
        return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::AND);
```

```
    case Cpu0::ATOMIC_LOAD_AND_I32:
```

```
        return emitAtomicBinary(MI, BB, 4, Cpu0::AND);
```

```
    case Cpu0::ATOMIC_LOAD_OR_I8:
```

```
        return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::OR);
```

```
    case Cpu0::ATOMIC_LOAD_OR_I16:
```

```
        return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::OR);
```

```
    case Cpu0::ATOMIC_LOAD_OR_I32:
```

```
        return emitAtomicBinary(MI, BB, 4, Cpu0::OR);
```

```
    case Cpu0::ATOMIC_LOAD_XOR_I8:
```

```
        return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::XOR);
```

```
    case Cpu0::ATOMIC_LOAD_XOR_I16:
```

```
        return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::XOR);
```

```
    case Cpu0::ATOMIC_LOAD_XOR_I32:
```

```
        return emitAtomicBinary(MI, BB, 4, Cpu0::XOR);
```

```
    case Cpu0::ATOMIC_LOAD_NAND_I8:
```

```
        return emitAtomicBinaryPartword(MI, BB, 1, 0, true);
```

```
    case Cpu0::ATOMIC_LOAD_NAND_I16:
```

(continues on next page)

(continued from previous page)

```

        return emitAtomicBinaryPartword(MI, BB, 2, 0, true);
    case Cpu0::ATOMIC_LOAD_NAND_I32:
        return emitAtomicBinary(MI, BB, 4, 0, true);

    case Cpu0::ATOMIC_LOAD_SUB_I8:
        return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::SUBu);
    case Cpu0::ATOMIC_LOAD_SUB_I16:
        return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::SUBu);
    case Cpu0::ATOMIC_LOAD_SUB_I32:
        return emitAtomicBinary(MI, BB, 4, Cpu0::SUBu);

    case Cpu0::ATOMIC_SWAP_I8:
        return emitAtomicBinaryPartword(MI, BB, 1, 0);
    case Cpu0::ATOMIC_SWAP_I16:
        return emitAtomicBinaryPartword(MI, BB, 2, 0);
    case Cpu0::ATOMIC_SWAP_I32:
        return emitAtomicBinary(MI, BB, 4, 0);

    case Cpu0::ATOMIC_CMP_SWAP_I8:
        return emitAtomicCmpSwapPartword(MI, BB, 1);
    case Cpu0::ATOMIC_CMP_SWAP_I16:
        return emitAtomicCmpSwapPartword(MI, BB, 2);
    case Cpu0::ATOMIC_CMP_SWAP_I32:
        return emitAtomicCmpSwap(MI, BB, 4);
    }

}

// This function also handles Cpu0::ATOMIC_SWAP_I32 (when BinOpcode == 0), and
// Cpu0::ATOMIC_LOAD_NAND_I32 (when Nand == true)
MachineBasicBlock *Cpu0TargetLowering::emitAtomicBinary(
    MachineInstr &MI, MachineBasicBlock *BB, unsigned Size, unsigned BinOpcode,
    bool Nand) const {
    assert((Size == 4) && "Unsupported size for EmitAtomicBinary.");

    MachineFunction *MF = BB->getParent();
    MachineRegisterInfo &RegInfo = MF->getRegInfo();
    const TargetRegisterClass *RC = getRegClassFor(MVT::getIntegerVT(Size * 8));
    const TargetInstrInfo *TII = Subtarget.getInstrInfo();
    DebugLoc DL = MI.getDebugLoc();
    unsigned LL, SC, AND, XOR, ZERO, BEQ;

    LL = Cpu0::LL;
    SC = Cpu0::SC;
    AND = Cpu0::AND;
    XOR = Cpu0::XOR;
    ZERO = Cpu0::ZERO;
    BEQ = Cpu0::BEQ;

    unsigned OldVal = MI.getOperand(0).getReg();
    unsigned Ptr = MI.getOperand(1).getReg();
    unsigned Incr = MI.getOperand(2).getReg();

```

(continues on next page)

(continued from previous page)

```

unsigned StoreVal = RegInfo.createVirtualRegister(RC);
unsigned AndRes = RegInfo.createVirtualRegister(RC);
unsigned AndRes2 = RegInfo.createVirtualRegister(RC);
unsigned Success = RegInfo.createVirtualRegister(RC);

// insert new blocks after the current block
const BasicBlock *LLVM_BB = BB->getBasicBlock();
MachineBasicBlock *loopMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineFunction::iterator It = ++BB->getIterator();
MF->insert(It, loopMBB);
MF->insert(It, exitMBB);

// Transfer the remainder of BB and its successor edges to exitMBB.
exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
exitMBB->transferSuccessorsAndUpdatePHIs(BB);

// thisMBB:
// ...
// fallthrough --> loopMBB
BB->addSuccessor(loopMBB);
loopMBB->addSuccessor(loopMBB);
loopMBB->addSuccessor(exitMBB);

// loopMBB:
//   ll oldval, 0(ptr)
//   <binop> storeval, oldval, incr
//   sc success, storeval, 0(ptr)
//   beq success, $0, loopMBB
BB = loopMBB;
BuildMI(BB, DL, TII->get(LL), OldVal).addReg(Ptr).addImm(0);
if (Nand) {
    // and andres, oldval, incr
    // xor storeval, $0, andres
    // xor storeval2, $0, storeval
    BuildMI(BB, DL, TII->get(AND), AndRes).addReg(OldVal).addReg(Incr);
    BuildMI(BB, DL, TII->get(XOR), StoreVal).addReg(ZERO).addReg(AndRes);
    BuildMI(BB, DL, TII->get(XOR), AndRes2).addReg(ZERO).addReg(AndRes);
} else if (BinOpcode) {
    // <binop> storeval, oldval, incr
    BuildMI(BB, DL, TII->get(BinOpcode), StoreVal).addReg(OldVal).addReg(Incr);
} else {
    StoreVal = Incr;
}
BuildMI(BB, DL, TII->get(SC), Success).addReg(StoreVal).addReg(Ptr).addImm(0);
BuildMI(BB, DL, TII->get(BEQ)).addReg(Success).addReg(ZERO).addMBB(loopMBB);

MI.eraseFromParent(); // The instruction is gone now.

return exitMBB;
}

```

(continues on next page)

(continued from previous page)

```
MachineBasicBlock *Cpu0TargetLowering::emitSignExtendToI32InReg(
    MachineInstr &MI, MachineBasicBlock *BB, unsigned Size, unsigned DstReg,
    unsigned SrcReg) const {
    const TargetInstrInfo *TII = Subtarget.getInstrInfo();
    DebugLoc DL = MI.getDebugLoc();

    MachineFunction *MF = BB->getParent();
    MachineRegisterInfo &RegInfo = MF->getRegInfo();
    const TargetRegisterClass *RC = getRegClassFor(MVT::i32);
    unsigned ScrReg = RegInfo.createVirtualRegister(RC);

    assert(Size < 32);
    int64_t ShiftImm = 32 - (Size * 8);

    BuildMI(BB, DL, TII->get(Cpu0::SHL), ScrReg).addReg(SrcReg).addImm(ShiftImm);
    BuildMI(BB, DL, TII->get(Cpu0::SRA), DstReg).addReg(ScrReg).addImm(ShiftImm);

    return BB;
}

MachineBasicBlock *Cpu0TargetLowering::emitAtomicBinaryPartword(
    MachineInstr &MI, MachineBasicBlock *BB, unsigned Size, unsigned BinOpcode,
    bool Nand) const {
    assert((Size == 1 || Size == 2) &&
           "Unsupported size for EmitAtomicBinaryPartial.");

    MachineFunction *MF = BB->getParent();
    MachineRegisterInfo &RegInfo = MF->getRegInfo();
    const TargetRegisterClass *RC = getRegClassFor(MVT::i32);
    const TargetInstrInfo *TII = Subtarget.getInstrInfo();
    DebugLoc DL = MI.getDebugLoc();

    unsigned Dest = MI.getOperand(0).getReg();
    unsigned Ptr = MI.getOperand(1).getReg();
    unsigned Incr = MI.getOperand(2).getReg();

    unsigned AlignedAddr = RegInfo.createVirtualRegister(RC);
    unsigned ShiftAmt = RegInfo.createVirtualRegister(RC);
    unsigned Mask = RegInfo.createVirtualRegister(RC);
    unsigned Mask2 = RegInfo.createVirtualRegister(RC);
    unsigned Mask3 = RegInfo.createVirtualRegister(RC);
    unsigned NewVal = RegInfo.createVirtualRegister(RC);
    unsigned OldVal = RegInfo.createVirtualRegister(RC);
    unsigned Incr2 = RegInfo.createVirtualRegister(RC);
    unsigned MaskLSB2 = RegInfo.createVirtualRegister(RC);
    unsigned PtrLSB2 = RegInfo.createVirtualRegister(RC);
    unsigned MaskUpper = RegInfo.createVirtualRegister(RC);
    unsigned AndRes = RegInfo.createVirtualRegister(RC);
    unsigned BinOpRes = RegInfo.createVirtualRegister(RC);
    unsigned BinOpRes2 = RegInfo.createVirtualRegister(RC);
    unsigned MaskedOldVal0 = RegInfo.createVirtualRegister(RC);
```

(continues on next page)

(continued from previous page)

```
unsigned StoreVal = RegInfo.createVirtualRegister(RC);
unsigned MaskedOldVal1 = RegInfo.createVirtualRegister(RC);
unsigned SrlRes = RegInfo.createVirtualRegister(RC);
unsigned Success = RegInfo.createVirtualRegister(RC);

// insert new blocks after the current block
const BasicBlock *LLVM_BB = BB->getBasicBlock();
MachineBasicBlock *loopMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *sinkMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineFunction::iterator It = ++BB->getIterator();

MF->insert(It, loopMBB);
MF->insert(It, sinkMBB);
MF->insert(It, exitMBB);

// Transfer the remainder of BB and its successor edges to exitMBB.
exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
exitMBB->transferSuccessorsAndUpdatePHIs(BB);

BB->addSuccessor(loopMBB);
loopMBB->addSuccessor(loopMBB);
loopMBB->addSuccessor(sinkMBB);
sinkMBB->addSuccessor(exitMBB);

// thisMBB:
//    addiu    masklsb2,$0,-4          # 0xfffffffffc
//    and     alignedaddr,ptr,masklsb2
//    andi    ptrlsb2,ptr,3
//    sll     shiftamt,ptrlsb2,3
//    ori     maskupper,$0,255         # 0xff
//    sll     mask,maskupper,shiftamt
//    xor     mask2,$0,mask
//    xor     mask3,$0,mask2
//    sll     incr2,incr,shiftamt

int64_t MaskImm = (Size == 1) ? 255 : 65535;
BuildMI(BB, DL, TII->get(Cpu0::ADDiu), MaskLSB2)
    .addReg(Cpu0::ZERO).addImm(-4);
BuildMI(BB, DL, TII->get(Cpu0::AND), AlignedAddr)
    .addReg(Ptr).addReg(MaskLSB2);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), PtrLSB2).addReg(Ptr).addImm(3);
if (Subtarget.isLittle()) {
    BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(PtrLSB2).addImm(3);
} else {
    unsigned Off = RegInfo.createVirtualRegister(RC);
    BuildMI(BB, DL, TII->get(Cpu0::XORi), Off)
        .addReg(PtrLSB2).addImm((Size == 1) ? 3 : 2);
    BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(Off).addImm(3);
}
BuildMI(BB, DL, TII->get(Cpu0::ORi), MaskUpper)
```

(continues on next page)

(continued from previous page)

```

    .addReg(Cpu0::ZERO).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), Mask)
    .addReg(MaskUpper).addReg(ShiftAmt);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask2).addReg(Cpu0::ZERO).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask3).addReg(Cpu0::ZERO).addReg(Mask2);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), Incr2).addReg(Incr).addReg(ShiftAmt);

// atomic.load.binop
// loopMBB:
//    ll      oldval,0(alignedaddr)
//    binop  binopres,oldval,incr2
//    and    newval,binopres,mask
//    and    maskedoldval0,oldval,mask3
//    or     storeval,maskedoldval0,newval
//    sc     success,storeval,0(alignedaddr)
//    beq   success,$0,loopMBB

// atomic.swap
// loopMBB:
//    ll      oldval,0(alignedaddr)
//    and    newval,incr2,mask
//    and    maskedoldval0,oldval,mask3
//    or     storeval,maskedoldval0,newval
//    sc     success,storeval,0(alignedaddr)
//    beq   success,$0,loopMBB

BB = loopMBB;
unsigned LL = Cpu0::LL;
BuildMI(BB, DL, TII->get(LL), OldVal).addReg(AlignedAddr).addImm(0);
if (Nand) {
    // and andres, oldval, incr2
    // xor binopres, $0, andres
    // xor binopres2, $0, binopres
    // and newval, binopres, mask
    BuildMI(BB, DL, TII->get(Cpu0::AND), AndRes).addReg(OldVal).addReg(Incr2);
    BuildMI(BB, DL, TII->get(Cpu0::XOR), BinOpRes)
        .addReg(Cpu0::ZERO).addReg(AndRes);
    BuildMI(BB, DL, TII->get(Cpu0::XOR), BinOpRes2)
        .addReg(Cpu0::ZERO).addReg(BinOpRes);
    BuildMI(BB, DL, TII->get(Cpu0::AND), NewVal).addReg(BinOpRes).addReg(Mask);
} else if (BinOpcode) {
    // <binop> binopres, oldval, incr2
    // and newval, binopres, mask
    BuildMI(BB, DL, TII->get(BinOpcode), BinOpRes).addReg(OldVal).addReg(Incr2);
    BuildMI(BB, DL, TII->get(Cpu0::AND), NewVal).addReg(BinOpRes).addReg(Mask);
} else { // atomic.swap
    // and newval, incr2, mask
    BuildMI(BB, DL, TII->get(Cpu0::AND), NewVal).addReg(Incr2).addReg(Mask);
}

BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal0)
    .addReg(OldVal).addReg(Mask2);

```

(continues on next page)

(continued from previous page)

```

BuildMI(BB, DL, TII->get(Cpu0::OR), StoreVal)
    .addReg(MaskedOldVal0).addReg(NewVal);
unsigned SC = Cpu0::SC;
BuildMI(BB, DL, TII->get(SC), Success)
    .addReg(StoreVal).addReg(AlignedAddr).addImm(0);
BuildMI(BB, DL, TII->get(Cpu0::BEQ))
    .addReg(Success).addReg(Cpu0::ZERO).addMBB(loopMBB);

// sinkMBB:
//     and      maskedoldval1,oldval,mask
//     srl      srlres,maskedoldval1,shiftamt
//     sign_extend dest,srlres
BB = sinkMBB;

BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal1)
    .addReg(OldVal).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::SHRV), SrlRes)
    .addReg(MaskedOldVal1).addReg(ShiftAmt);
BB = emitSignExtendToI32InReg(MI, BB, Size, Dest, SrlRes);

MI.eraseFromParent(); // The instruction is gone now.

return exitMBB;
}

MachineBasicBlock * Cpu0TargetLowering::emitAtomicCmpSwap(MachineInstr &MI,
                                                          MachineBasicBlock *BB,
                                                          unsigned Size) const {
assert((Size == 4) && "Unsupported size for EmitAtomicCmpSwap.");

MachineFunction *MF = BB->getParent();
MachineRegisterInfo &RegInfo = MF->getRegInfo();
const TargetRegisterClass *RC = getRegClassFor(MVT::getIntegerVT(Size * 8));
const TargetInstrInfo *TII = Subtarget.getInstrInfo();
DebugLoc DL = MI.getDebugLoc();
unsigned LL, SC, ZERO, BNE, BEQ;

LL = Cpu0::LL;
SC = Cpu0::SC;
ZERO = Cpu0::ZERO;
BNE = Cpu0::BNE;
BEQ = Cpu0::BEQ;

unsigned Dest      = MI.getOperand(0).getReg();
unsigned Ptr       = MI.getOperand(1).getReg();
unsigned OldVal   = MI.getOperand(2).getReg();
unsigned NewVal   = MI.getOperand(3).getReg();

unsigned Success = RegInfo.createVirtualRegister(RC);

// insert new blocks after the current block
const BasicBlock *LLVM_BB = BB->getBasicBlock();

```

(continues on next page)

(continued from previous page)

```

MachineBasicBlock *loop1MBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *loop2MBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineFunction::iterator It = ++BB->getIterator();

MF->insert(It, loop1MBB);
MF->insert(It, loop2MBB);
MF->insert(It, exitMBB);

// Transfer the remainder of BB and its successor edges to exitMBB.
exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
exitMBB->transferSuccessorsAndUpdatePHIs(BB);

// thisMBB:
// ...
// fallthrough --> loop1MBB
BB->addSuccessor(loop1MBB);
loop1MBB->addSuccessor(exitMBB);
loop1MBB->addSuccessor(loop2MBB);
loop2MBB->addSuccessor(loop1MBB);
loop2MBB->addSuccessor(exitMBB);

// loop1MBB:
//   ll dest, 0(ptr)
//   bne dest, oldval, exitMBB
BB = loop1MBB;
BuildMI(BB, DL, TII->get(LL), Dest).addReg(Ptr).addImm(0);
BuildMI(BB, DL, TII->get(BNE))
    .addReg(Dest).addReg(OldVal).addMBB(exitMBB);

// loop2MBB:
//   sc success, newval, 0(ptr)
//   beq success, $0, loop1MBB
BB = loop2MBB;
BuildMI(BB, DL, TII->get(SC), Success)
    .addReg(NewVal).addReg(Ptr).addImm(0);
BuildMI(BB, DL, TII->get(BEQ))
    .addReg(Success).addReg(ZERO).addMBB(loop1MBB);

MI.eraseFromParent(); // The instruction is gone now.

return exitMBB;
}

MachineBasicBlock *
Cpu0TargetLowering::emitAtomicCmpSwapPartword(MachineInstr &MI,
                                              MachineBasicBlock *BB,
                                              unsigned Size) const {
    assert((Size == 1 || Size == 2) &&
           "Unsupported size for EmitAtomicCmpSwapPartial.");
}

```

(continues on next page)

(continued from previous page)

```
MachineFunction *MF = BB->getParent();
MachineRegisterInfo &RegInfo = MF->getRegInfo();
const TargetRegisterClass *RC = getRegClassFor(MVT::i32);
const TargetInstrInfo *TII = Subtarget.getInstrInfo();
DebugLoc DL = MI.getDebugLoc();

unsigned Dest      = MI.getOperand(0).getReg();
unsigned Ptr       = MI.getOperand(1).getReg();
unsigned CmpVal   = MI.getOperand(2).getReg();
unsigned NewVal   = MI.getOperand(3).getReg();

unsigned AlignedAddr = RegInfo.createVirtualRegister(RC);
unsigned ShiftAmt = RegInfo.createVirtualRegister(RC);
unsigned Mask = RegInfo.createVirtualRegister(RC);
unsigned Mask2 = RegInfo.createVirtualRegister(RC);
unsigned Mask3 = RegInfo.createVirtualRegister(RC);
unsigned ShiftedCmpVal = RegInfo.createVirtualRegister(RC);
unsigned OldVal = RegInfo.createVirtualRegister(RC);
unsigned MaskedOldVal0 = RegInfo.createVirtualRegister(RC);
unsigned ShiftedNewVal = RegInfo.createVirtualRegister(RC);
unsigned MaskLSB2 = RegInfo.createVirtualRegister(RC);
unsigned PtrLSB2 = RegInfo.createVirtualRegister(RC);
unsigned MaskUpper = RegInfo.createVirtualRegister(RC);
unsigned MaskedCmpVal = RegInfo.createVirtualRegister(RC);
unsigned MaskedNewVal = RegInfo.createVirtualRegister(RC);
unsigned MaskedOldVal1 = RegInfo.createVirtualRegister(RC);
unsigned StoreVal = RegInfo.createVirtualRegister(RC);
unsigned SrlRes = RegInfo.createVirtualRegister(RC);
unsigned Success = RegInfo.createVirtualRegister(RC);

// insert new blocks after the current block
const BasicBlock *LLVM_BB = BB->getBasicBlock();
MachineBasicBlock *loop1MBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *loop2MBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *sinkMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineFunction::iterator It = ++BB->getIterator();

MF->insert(It, loop1MBB);
MF->insert(It, loop2MBB);
MF->insert(It, sinkMBB);
MF->insert(It, exitMBB);

// Transfer the remainder of BB and its successor edges to exitMBB.
exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
exitMBB->transferSuccessorsAndUpdatePHIs(BB);

BB->addSuccessor(loop1MBB);
loop1MBB->addSuccessor(sinkMBB);
loop1MBB->addSuccessor(loop2MBB);
loop2MBB->addSuccessor(loop1MBB);
```

(continues on next page)

```

loop2MBB->addSuccessor(sinkMBB);
sinkMBB->addSuccessor(exitMBB);

// FIXME: computation of newval2 can be moved to loop2MBB.
// thisMBB:
//    addiu    masklsb2,$0,-4          # 0xfffffffffc
//    and     alignedaddr,ptr,masklsb2
//    andi    ptrlsb2,ptr,3
//    shl     shiftamt,ptrlsb2,3
//    ori     maskupper,$0,255        # 0xff
//    shl     mask,maskupper,shiftamt
//    xor     mask2,$0,mask
//    xor     mask3,$0,mask2
//    andi    maskedcmpval,cmpval,255
//    shl     shiftedcmpval,maskedcmpval,shiftamt
//    andi    maskednewval,newval,255
//    shl     shiftednewval,maskednewval,shiftamt
int64_t MaskImm = (Size == 1) ? 255 : 65535;
BuildMI(BB, DL, TII->get(Cpu0::ADDiu), MaskLSB2)
    .addReg(Cpu0::ZERO).addImm(-4);
BuildMI(BB, DL, TII->get(Cpu0::AND), AlignedAddr)
    .addReg(Ptr).addReg(MaskLSB2);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), PtrLSB2).addReg(Ptr).addImm(3);
if (Subtarget.isLittle()) {
    BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(PtrLSB2).addImm(3);
} else {
    unsigned Off = RegInfo.createVirtualRegister(RC);
    BuildMI(BB, DL, TII->get(Cpu0::XORi), Off)
        .addReg(PtrLSB2).addImm((Size == 1) ? 3 : 2);
    BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(Off).addImm(3);
}
BuildMI(BB, DL, TII->get(Cpu0::ORi), MaskUpper)
    .addReg(Cpu0::ZERO).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), Mask)
    .addReg(MaskUpper).addReg(ShiftAmt);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask2).addReg(Cpu0::ZERO).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask3).addReg(Cpu0::ZERO).addReg(Mask2);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), MaskedCmpVal)
    .addReg(CmpVal).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), ShiftedCmpVal)
    .addReg(MaskedCmpVal).addReg(ShiftAmt);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), MaskedNewVal)
    .addReg(NewVal).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), ShiftedNewVal)
    .addReg(MaskedNewVal).addReg(ShiftAmt);

// loop1MBB:
//    ll      oldval,0(alginedaddr)
//    and    maskedoldval0,oldval,mask
//    bne    maskedoldval0,shiftedcmpval,sinkMBB
BB = loop1MBB;
unsigned LL = Cpu0::LL;
BuildMI(BB, DL, TII->get(LL), OldVal).addReg(AlignedAddr).addImm(0);
```

(continues on next page)

(continued from previous page)

```

BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal0)
    .addReg(OldVal).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::BNE))
    .addReg(MaskedOldVal0).addReg(ShiftedCmpVal).addMBB(sinkMBB);

// loop2MBB:
//     and      maskedoldval1,oldval,mask3
//     or       storeval,maskedoldval1,shiftednewval
//     sc       success,storeval,0(alignedaddr)
//     beq      success,$0,loop1MBB
BB = loop2MBB;
BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal1)
    .addReg(OldVal).addReg(Mask3);
BuildMI(BB, DL, TII->get(Cpu0::OR), StoreVal)
    .addReg(MaskedOldVal1).addReg(ShiftedNewVal);
unsigned SC = Cpu0::SC;
BuildMI(BB, DL, TII->get(SC), Success)
    .addReg(StoreVal).addReg(AignedAddr).addImm(0);
BuildMI(BB, DL, TII->get(Cpu0::BEQ))
    .addReg(Success).addReg(Cpu0::ZERO).addMBB(loop1MBB);

// sinkMBB:
//     srl      srlres,maskedoldval0,shiftamt
//     sign_extend dest,srlres
BB = sinkMBB;

BuildMI(BB, DL, TII->get(Cpu0::SHRV), SrlRes)
    .addReg(MaskedOldVal0).addReg(ShiftAmt);
BB = emitSignExtendToI32InReg(MI, BB, Size, Dest, SrlRes);

MI.eraseFromParent(); // The instruction is gone now.

return exitMBB;
}

```

```

SDValue Cpu0TargetLowering::lowerATOMIC_FENCE(SDValue Op,
                                              SelectionDAG &DAG) const {
    // FIXME: Need pseudo-fence for 'singlethread' fences
    // FIXME: Set SType for weaker fences where supported/appropriate.
    unsigned SType = 0;
    SDLoc DL(Op);
    return DAG.getNode(Cpu0ISD::Sync, DL, MVT::Other, Op.getOperand(0),
                        DAG.getConstant(SType, DL, MVT::i32));
}

```

Ibdex/chapters/Chapter12_1/Cpu0RegisterInfo.h

```

/// Code Generation virtual methods...
const TargetRegisterClass *getPointerRegClass(const MachineFunction &MF,
                                              unsigned Kind) const override;

```

Ibdex/chapters/Chapter12_1/Cpu0RegisterInfo.cpp

```
const TargetRegisterClass *
Cpu0RegisterInfo::getPointerRegClass(const MachineFunction &MF,
                                      unsigned Kind) const {
    return &Cpu0::CPURegsRegClass;
}
```

Ibdex/chapters/Chapter12_1/Cpu0SEISelLowering.cpp

```
Cpu0SETargetLowering::Cpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                             const Cpu0Subtarget &STI)
    : Cpu0TargetLowering(TM, STI) {

    setOperationAction(ISD::ATOMIC_FENCE, MVT::Other, Custom);

    ...
}
```

Ibdex/chapters/Chapter12_1/Cpu0TargetMachine.cpp

```
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {

    void addIRPasses() override;

    ...

};

void Cpu0PassConfig::addIRPasses() {
    TargetPassConfig::addIRPasses();
    addPass(createAtomicExpandPass());
}
```

Since the *SC* instruction uses *RegisterOperand* type in *Cpu0InstrInfo.td* and *SC* uses the *FMem* node whose *DecoderMethod* is *DecodeMem*, the *DecodeMem()* function in *Cpu0Disassembler.cpp* needs to be modified accordingly.

The atomic node defined in *let usesCustomInserter = 1 in* within *Cpu0InstrInfo.td* tells LLVM to call *EmitInstrWithCustomInserter()* in *Cpu0ISelLowering.cpp* after the Instruction Selection stage, specifically in the *Cpu0TargetLowering::EmitInstrWithCustomInserter()* function invoked during the *ExpandISelPseudos::runOnMachineFunction()* phase.

For example, the declaration *def ATOMIC_LOAD_ADD_I8 : Atomic2Ops<atomic_load_add_8, CPURegs>;* will trigger a call to *EmitInstrWithCustomInserter()* with the machine instruction opcode *ATOMIC_LOAD_ADD_I8* when the IR *load atomic i8** is encountered.

The call to *setInsertFencesForAtomic(true);* in *Cpu0ISelLowering.cpp* will trigger the *addIRPasses()* function in *Cpu0TargetMachine.cpp*, which in turn invokes *createAtomicExpandPass()* to create the LLVM IR *ATOMIC_FENCE*.

Later, *lowerATOMIC_FENCE()* in *Cpu0ISelLowering.cpp* will emit a *Cpu0ISD::Sync* when it sees an *ATOMIC_FENCE* IR, because of the statement *setOperationAction(ISD::ATOMIC_FENCE, MVT::Other, Custom);* in *Cpu0SEISelLowering.cpp*.

Finally, the pattern defined in *Cpu0InstrInfo.td* will translate the DAG node into the actual *sync* instruction via *def SYNC* and its alias *SYNC 0*.

This part of the Cpu0 backend code is similar to Mips, except that Cpu0 does not include the *nor* instruction.

Below is a table listing the atomic IRs, their corresponding DAG nodes, and machine opcodes.

Table 12.6: The atomic related IRs, their corresponding DAGs and Opcode
of Cpu0ISelLowering.cpp

IR	DAG	Opcode
load atomic	AtomicLoad	ATOMIC_CMP_SWAP_XXX
store atomic	AtomicStore	ATOMIC_SWAP_XXX
atomicrmw add	AtomicLoadAdd	ATOMIC_LOAD_ADD_XXX
atomicrmw sub	AtomicLoadSub	ATOMIC_LOAD_SUB_XXX
atomicrmw xor	AtomicLoadXor	ATOMIC_LOAD_XOR_XXX
atomicrmw and	AtomicLoadAnd	ATOMIC_LOAD_AND_XXX
atomicrmw nand	AtomicLoadNand	ATOMIC_LOAD_NAND_XXX
atomicrmw or	AtomicLoadOr	ATOMIC_LOAD_OR_XXX
cpxchg	AtomicCmpSwapWithSuccess	ATOMIC_CMP_SWAP_XXX
atomicrmw xchg	AtomicLoadSwap	ATOMIC_SWAP_XXX

The input files *atomics.ll* and *atomics-fences.ll* include tests for LLVM atomic IRs.

The C++ source files *ch12_atomics.cpp* and *ch12_atomics-fences.cpp* are used to generate the corresponding LLVM atomic IRs. To compile these files, use the following *clang++* options:

```
clang++ -pthread -std=c++11
```

VERIFY BACKEND ON VERILOG SIMULATOR

- *Create Verilog Simulator of Cpu0*
- *Verify backend*
- *Other LLVM-Based Tools for Cpu0 Processor*

Until now, we have developed an LLVM backend capable of compiling C or assembly code, as illustrated in the white part of Fig. 13.1. If the program does not contain global variables, the ELF object file can be dumped to a hex file using the following command:

```
llvm-objdump -d
```

This functionality was completed in Chapter *ELF Support*.

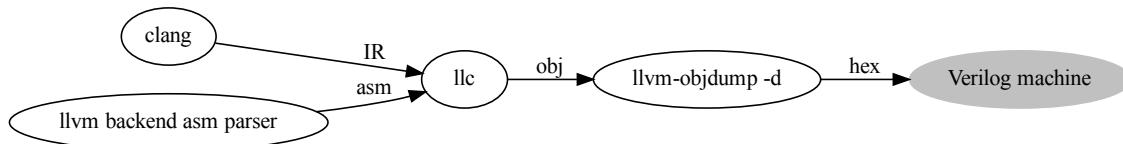


Fig. 13.1: Cpu0 backend without linker

This chapter implements the Cpu0 instructions using the Verilog language, as represented by the gray part in the figure above.

With this Verilog-based machine, we can execute the hex program generated by the LLVM backend on the Cpu0 Verilog simulator running on a PC. This allows us to observe and verify the execution results of Cpu0 instructions directly on the hardware model.

13.1 Create Verilog Simulator of Cpu0

Verilog is an IEEE-standard language widely used in IC design. There are many books and free online resources available for learning Verilog¹²³⁴⁵.

Verilog is also known as Verilog HDL (Hardware Description Language), not to be confused with **VHDL**, which serves the same purpose but is a competing language⁶.

An example implementation, `lbdex/verilog/cpu0.v`, contains the Cpu0 processor design written in Verilog. As described in Appendix A, we have installed the Icarus Verilog tool on both iMac and Linux systems. The `cpu0.v` design is relatively simple, with only a few hundred lines of code in total.

Although this implementation does not include pipelining, it simulates delay slots (via the `SIMULATE_DELAY_SLOT` section of the code) to accurately estimate pipeline machine cycles.

Verilog has a C-like syntax, and since this book focuses on compiler implementation, we present the `cpu0.v` code and the build commands below **without an in-depth explanation**. We expect that readers with some patience and curiosity will be able to understand the Verilog code.

Cpu0 supports **memory-mapped I/O**, one of the two primary I/O models in computer architecture (the other being **instruction-based I/O**). Cpu0 maps the output port to memory address `0x80000`. When executing the instruction:

```
st $ra, cx($rb)
```

where `cx($rb)` equals `0x80000`, the Cpu0 processor outputs the content to that I/O port, as demonstrated below.

```
ST : begin
  ...
  if (R[b]+c16 == `IOADDR) begin
    outw(R[a]);
  end
end
```

Ibdex/verilog/cpu0.v

```
// https://www.francisz.cn/download/IEEE_Standard_1800-2012%20SystemVerilog.pdf

// configurable value below
`define SIMULATE_DELAY_SLOT
// cpu032I memory limit, jsub:24-bit
`define MEMSIZE      'h1000000
`define MEMEMPTY    8'hFF
`define NULL        8'h00
`define IOADDR      'hff000000 // IO mapping address
`define TIMEOUT     #30000000000

// Operand width
`define INT32 2'b11      // 32 bits
`define INT24 2'b10      // 24 bits
`define INT16 2'b01      // 16 bits
`define BYTE  2'b00      // 8  bits
```

(continues on next page)

¹ <http://ccckmit.wikidot.com/ve:main>

² <http://www.ece.umd.edu/courses/enee359a/>

³ http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf

⁴ http://d1.amobbs.com/bbs_upload782111/files_33/ourdev_585395BQ8J9A.pdf

⁵ <http://en.wikipedia.org/wiki/Verilog>

⁶ <http://en.wikipedia.org/wiki/VHDL>

(continued from previous page)

```

`define EXE 3'b000
`define RESET 3'b001
`define ABORT 3'b010
`define IRQ 3'b011
`define ERROR 3'b100

// Reference web: http://ccckmit.wikidot.com/ocs:cpu0
module cpu0(input clock, reset, input [2:0] itype, output reg [2:0] tick,
            output reg [31:0] ir, pc, mar, mdr, inout [31:0] dbus,
            output reg m_en, m_rw, output reg [1:0] m_size,
            input cfg);
    reg signed [31:0] R [0:15];
    reg signed [31:0] C0R [0:1]; // co-processor 0 register
    // High and Low part of 64 bit result
    reg [7:0] op;
    reg [3:0] a, b, c;
    reg [4:0] c5;
    reg signed [31:0] c12, c16, c24, Ra, Rb, Rc, pc0; // pc0: instruction pc
    reg [31:0] uc16, URa, URb, URc, HI, LO, CF, tmp;
    reg [63:0] cycles;

    // register name
    `define SP    R[13]    // Stack Pointer
    `define LR    R[14]    // Link Register
    `define SW    R[15]    // Status Word

    // C0 register name
    `define PC    C0R[0]   // Program Counter
    `define EPC   C0R[1]   // exception PC value

    // SW Flage
    `define I2    `SW[16]   // Hardware Interrupt 1, IO1 interrupt, status,
                    // 1: in interrupt
    `define I1    `SW[15]   // Hardware Interrupt 0, timer interrupt, status,
                    // 1: in interrupt
    `define IO    `SW[14]   // Software interrupt, status, 1: in interrupt
    `define I    `SW[13]   // Interrupt, 1: in interrupt
    `define I2E   `SW[12]   // Hardware Interrupt 1, IO1 interrupt, Enable
    `define I1E   `SW[11]   // Hardware Interrupt 0, timer interrupt, Enable
    `define IOE   `SW[10]   // Software Interrupt Enable
    `define IE    `SW[9]    // Interrupt Enable
    `define M    `SW[8:6]  // Mode bits, itype
    `define D    `SW[5]    // Debug Trace
    `define V    `SW[3]    // Overflow
    `define C    `SW[2]    // Carry
    `define Z    `SW[1]    // Zero
    `define N    `SW[0]    // Negative flag

    `define LE    CF[0]   // Endian bit, Big Endian:0, Little Endian:1
    // Instruction Opcode
parameter [7:0] NOP=8'h00, LD=8'h01, ST=8'h02, LB=8'h03, LBu=8'h04, SB=8'h05,

```

(continues on next page)

(continued from previous page)

```

LH=8'h06,LHu=8'h07,SH=8'h08,ADDiu=8'h09,MOVZ=8'h0A,MOVN=8'h0B,ANDi=8'h0C,
ORi=8'h0D,XORi=8'h0E,LUi=8'h0F,
ADDu=8'h11,SUBu=8'h12,ADD=8'h13,SUB=8'h14,CLZ=8'h15,CLO=8'h16,MUL=8'h17,
AND=8'h18,OR=8'h19,XOR=8'h1A,NOR=8'h1B,
ROL=8'h1C,ROR=8'h1D,SHL=8'h1E,SHR=8'h1F,
SRA=8'h20,SRAV=8'h21,SHLV=8'h22,SHRV=8'h23,ROLV=8'h24,RORV=8'h25,
`ifdef CPU0II
    SLTi=8'h26,SLTi=8'h27, SLT=8'h28,SLTu=8'h29,
`endif
    CMP=8'h2A,
    CMPu=8'h2B,
    JEQ=8'h30,JNE=8'h31,JLT=8'h32,JGT=8'h33,JLE=8'h34,JGE=8'h35,
    JMP=8'h36,
`ifdef CPU0II
    BEQ=8'h37,BNE=8'h38,
`endif
    JALR=8'h39,BAL=8'h3A,JSUB=8'h3B,RET=8'h3C,
    MULT=8'h41,MULTu=8'h42,DIV=8'h43,DIVu=8'h44,
    MFHI=8'h46,MFLO=8'h47,MTHI=8'h48,MTLO=8'h49,
    MFC0=8'h50,MTC0=8'h51,COMOV=8'h52;

    reg [0:0] inExe = 0;
    reg [2:0] state, next_state;
    reg [2:0] st_taskInt, ns_taskInt;
    parameter Reset=3'h0, Fetch=3'h1, Decode=3'h2, Execute=3'h3, MemAccess=3'h4,
              WriteBack=3'h5;
    integer i;
`ifdef SIMULATE_DELAY_SLOT
    reg [0:0] nextInstIsDelaySlot;
    reg [0:0] isDelaySlot;
    reg signed [31:0] delaySlotNextPC;
`endif

//transform data from the memory to little-endian form
task changeEndian(input [31:0] value, output [31:0] changeEndian); begin
    changeEndian = {value[7:0], value[15:8], value[23:16], value[31:24]};
end endtask

// Read Memory Word
task memReadStart(input [31:0] addr, input [1:0] size); begin
    mar = addr;      // read(m[addr])
    m_rw = 1;        // Access Mode: read
    m_en = 1;        // Enable read
    m_size = size;
end endtask

// Read Memory Finish, get data
task memReadEnd(output [31:0] data); begin
    mdr = dbus; // get momory, dbus = m[addr]
    data = mdr; // return to data
    m_en = 0; // read complete
end endtask

```

(continues on next page)

(continued from previous page)

```
// Write memory -- addr: address to write, data: date to write
task memWriteStart(input [31:0] addr, input [31:0] data, input [1:0] size);
begin
    mar = addr;      // write(m[addr], data)
    mdr = data;
    m_rw = 0;        // access mode: write
    m_en = 1;        // Enable write
    m_size = size;
end endtask

task memWriteEnd; begin // Write Memory Finish
    m_en = 0; // write complete
end endtask

task regSet(input [3:0] i, input [31:0] data); begin
    if (i != 0) R[i] = data;
end endtask

task C0regSet(input [3:0] i, input [31:0] data); begin
    if (i < 2) C0R[i] = data;
end endtask

task PCSet(input [31:0] data); begin
`ifndef SIMULATE_DELAY_SLOT
    nextInstIsDelaySlot = 1;
    delaySlotNextPC = data;
`else
    `PC = data;
`endif
end endtask

task retValSet(input [3:0] i, input [31:0] data); begin
    if (i != 0)
`ifndef SIMULATE_DELAY_SLOT
        R[i] = data + 4;
`else
        R[i] = data;
`endif
end endtask

task regHILoset(input [31:0] data1, input [31:0] data2); begin
    HI = data1;
    LO = data2;
end endtask

// output a word to Output port (equal to display the word to terminal)
task outw(input [31:0] data); begin
    if (`LE) begin // Little Endian
        changeEndian(data, data);
    end
    if (data[7:0] != 8'h00) begin
```

(continues on next page)

(continued from previous page)

```

$write("%c", data[7:0]);
if (data[15:8] != 8'h00)
    $write("%c", data[15:8]);
if (data[23:16] != 8'h00)
    $write("%c", data[23:16]);
if (data[31:24] != 8'h00)
    $write("%c", data[31:24]);
end
end endtask

// output a character (a byte)
task outc(input [7:0] data); begin
    $write("%c", data);
end endtask

task taskInterrupt(input [2:0] iMode); begin
if (inExe == 0) begin
    case (iMode)
        `RESET: begin
            `PC = 0; tick = 0; R[0] = 0; `SW = 0; `LR = -1;
            `IE = 0; `IOE = 1; `I1E = 1; `I2E = 1;
            `I = 0; `I0 = 0; `I1 = 0; `I2 = 0; inExe = 1;
            `LE = cfg;
            cycles = 0;
        end
        `ABORT: begin `PC = 4; end
        `IRQ: begin `PC = 8; `IE = 0; inExe = 1; end
        `ERROR: begin `PC = 12; end
    endcase
end
$display("taskInterrupt(%3b)", iMode);
end endtask

task taskExecute; begin
    tick = tick+1;
    case (state)
        Fetch: begin // Tick 1 : instruction fetch, throw PC to address bus,
                    // memory.read(m[PC])
            memReadStart(`PC, `INT32);
            pc0 = `PC;
        `ifdef SIMULATE_DELAY_SLOT
            if (nextInstIsDelaySlot == 1) begin
                isDelaySlot = 1;
                nextInstIsDelaySlot = 0;
                `PC = delaySlotNextPC;
            end
            else begin
                if (isDelaySlot == 1) isDelaySlot = 0;
                `PC = `PC+4;
            end
        `else
            `PC = `PC+4;
    end
end

```

(continues on next page)

(continued from previous page)

```

`endif
    next_state = Decode;
end
Decode: begin // Tick 2 : instruction decode, ir = m[PC]
    memReadEnd(ir); // IR = dbus = m[PC]
    {op,a,b,c} = ir[31:12];
    c24 = $signed(ir[23:0]);
    c16 = $signed(ir[15:0]);
    uc16 = ir[15:0];
    c12 = $signed(ir[11:0]);
    c5 = ir[4:0];
    Ra = R[a];
    Rb = R[b];
    Rc = R[c];
    URa = R[a];
    URb = R[b];
    URC = R[c];
    next_state = Execute;
end
Execute: begin // Tick 3 : instruction execution
    case (op)
        NOP:   ;
        // load and store instructions
        LD:    memReadStart(Rb+c16, `INT32);           // LD Ra, [Rb+Cx]; Ra<=[Rb+Cx]
        ST:    memWriteStart(Rb+c16, Ra, `INT32);      // ST Ra, [Rb+Cx]; Ra>[Rb+Cx]
        // LB Ra, [Rb+Cx]; Ra<=(byte) [Rb+Cx]
        LB:    memReadStart(Rb+c16, `BYTE);
        // LBu Ra, [Rb+Cx]; Ra<=(byte) [Rb+Cx]
        LBu:   memReadStart(Rb+c16, `BYTE);
        // SB Ra, [Rb+Cx]; Ra>=(byte) [Rb+Cx]
        SB:    memWriteStart(Rb+c16, Ra, `BYTE);
        LH:    memReadStart(Rb+c16, `INT16);           // LH Ra, [Rb+Cx]; Ra<=(2bytes) [Rb+Cx]
        LHu:   memReadStart(Rb+c16, `INT16);          // LHu Ra, [Rb+Cx]; Ra<=(2bytes) [Rb+Cx]
        // SH Ra, [Rb+Cx]; Ra>=(2bytes) [Rb+Cx]
        SH:    memWriteStart(Rb+c16, Ra, `INT16);
        // Conditional move
        MOVZ:  if (Rc==0) regSet(a, Rb);             // move if Rc equal to 0
        MOVN:  if (Rc!=0) regSet(a, Rb);             // move if Rc not equal to 0
        // Mathematic
        ADDiu: regSet(a, Rb+c16);                  // ADDiu Ra, Rb+Cx; Ra<=Rb+Cx
        CMP:   begin
            if (Rb < Rc) `N=1; else `N=0;
            // `N=(Rb-Rc<0); // why not work for bash make.sh cpu032I el Makefile.
        ↵builtins?
            `Z=(Rb-Rc==0);
        end // CMP Rb, Rc; SW=(Rb >= Rc)
        CMPu:  begin
            if (URb < URC) `N=1; else `N=0;
            `Z=(URb-URC==0);
        end // CMPu URb, URC; SW=(URb >= URC)
        ADDu:  regSet(a, Rb+Rc);                   // ADDu Ra,Rb,Rc; Ra<=Rb+Rc
        ADD:   begin regSet(a, Rb+Rc); if (a < Rb) `V = 1; else `V = 0;
    end

```

(continues on next page)

(continued from previous page)

```

if (`V) begin `I0 = 1; `I = 1; end
end
                                // ADD Ra,Rb,Rc; Ra<=Rb+Rc
SUBu: regSet(a, Rb-Rc);           // SUBu Ra,Rb,Rc; Ra<=Rb-Rc
SUB:   begin regSet(a, Rb-Rc); if (Rb < 0 && Rc > 0 && a >= 0)
      `V = 1; else `V =0;
      if (`V) begin `I0 = 1; `I = 1; end
end      // SUB Ra,Rb,Rc; Ra<=Rb-Rc
CLZ:   begin
      for (i=0; (i<32)&&((Rb&32'h80000000)==32'h00000000); i=i+1) begin
          Rb=Rb<<1;
      end
      regSet(a, i);
end
CLO:   begin
      for (i=0; (i<32)&&((Rb&32'h80000000)==32'h80000000); i=i+1) begin
          Rb=Rb<<1;
      end
      regSet(a, i);
end
MUL:   regSet(a, Rb*Rc);           // MUL Ra,Rb,Rc;      Ra<=Rb*Rc
DIVu:  regHILOSet(URa%URb, URa/URb); // DIVu URa,URb; HI<=URa%URb;
                                // LO<=URa/URb
                                // without exception overflow
DIV:   begin regHILOSet(Ra%Rb, Ra/Rb);
      if ((Ra < 0 && Rb < 0) || (Ra == 0)) `V = 1;
      else `V =0; end // DIV Ra,Rb; HI<=Ra%Rb; LO<=Ra/Rb; With overflow
AND:   regSet(a, Rb&Rc);           // AND Ra,Rb,Rc; Ra<=(Rb and Rc)
ANDi:  regSet(a, Rb&uc16);        // ANDi Ra,Rb,c16; Ra<=(Rb and c16)
OR:    regSet(a, Rb|Rc);           // OR Ra,Rb,Rc; Ra<=(Rb or Rc)
ORi:   regSet(a, Rb|uc16);         // ORi Ra,Rb,c16; Ra<=(Rb or c16)
XOR:   regSet(a, Rb^Rc);           // XOR Ra,Rb,Rc; Ra<=(Rb xor Rc)
NOR:   regSet(a, ~(Rb|Rc));        // NOR Ra,Rb,Rc; Ra<=(Rb nor Rc)
XORi:  regSet(a, Rb^uc16);         // XORi Ra,Rb,c16; Ra<=(Rb xor c16)
LUI:   regSet(a, uc16<<16);
SHL:   regSet(a, Rb<<c5);        // Shift Left; SHL Ra,Rb,Cx; Ra<=(Rb << Cx)
SRA:   regSet(a, (Rb>>c5));     // Shift Right with signed bit fill;
                                // https://stackoverflow.com/questions/39911655/how-to-synthesize-hardware-
→for-sra-instruction
    SHR:   regSet(a, Rb>>c5);     // Shift Right with 0 fill;
                                // SHR Ra,Rb,Cx; Ra<=(Rb >> Cx)
    SHLV:  regSet(a, Rb<<Rc);     // Shift Left; SHLV Ra,Rb,Rc; Ra<=(Rb << Rc)
    SRAV:  regSet(a, (Rb>>Rc));   // Shift Right with signed bit fill;
    SHRV:  regSet(a, Rb>>Rc);     // Shift Right with 0 fill;
                                // SHRV Ra,Rb,Rc; Ra<=(Rb >> Rc)
    ROL:   regSet(a, (Rb<<c5) | (Rb>>(32-c5))); // Rotate Left;
    ROR:   regSet(a, (Rb>>c5) | (Rb<<(32-c5))); // Rotate Right;
    ROLV:  begin // Can set Rc to -32<=Rc<=32 more efficiently.
              while (Rc < -32) Rc=Rc+32;
              while (Rc > 32) Rc=Rc-32;
              regSet(a, (Rb<<Rc) | (Rb>>(32-Rc))); // Rotate Left;
end

```

(continues on next page)

(continued from previous page)

```

RORV: begin
    while (Rc < -32) Rc=Rc+32;
    while (Rc > 32) Rc=Rc-32;
    regSet(a, (Rb>>Rc) | (Rb<<(32-Rc))); // Rotate Right;
end
MFLO: regSet(a, LO); // MFLO Ra; Ra<=LO
MFHI: regSet(a, HI); // MFHI Ra; Ra<=HI
MTLO: LO = Ra; // MTLO Ra; LO<=Ra
MTHI: HI = Ra; // MTHI Ra; HI<=Ra
MULT: {HI, LO}=Ra*Rb; // MULT Ra,Rb; HI<=((Ra*Rb)>>32);
// LO<=((Ra*Rb) and 0x00000000ffffffff);
// with exception overflow
MULTu: {HI, LO}=URa*URb; // MULT URa,URb; HI<=((URa*URb)>>32);
// LO<=((URa*URb) and 0x00000000ffffffff);
// without exception overflow
MFC0: regSet(a, C0R[b]); // MFC0 a, b; Ra<=C0R[Rb]
MTC0: C0regSet(a, Rb); // MTC0 a, b; C0R[a]<=Rb
COMOV: C0regSet(a, C0R[b]); // COMOV a, b; C0R[a]<=C0R[b]
`ifdef CPU0II
    // set
    SLT: if (Rb < Rc) R[a]=1; else R[a]=0;
    SLTu: if (URb < URc) R[a]=1; else R[a]=0;
    SLTi: if (Rb < c16) R[a]=1; else R[a]=0;
    SLTi: if (URb < uc16) R[a]=1; else R[a]=0;
    // Branch Instructions
    BEQ: if (Ra==Rb) PCSet(`PC+c16);
    BNE: if (Ra!=Rb) PCSet(`PC+c16);
`endif
    // Jump Instructions
    JEQ: if (`Z) PCSet(`PC+c24); // JEQ Cx; if SW(=) PC PC+Cx
    JNE: if (!`Z) PCSet(`PC+c24); // JNE Cx; if SW(!=) PC PC+Cx
    JLT: if (`N) PCSet(`PC+c24); // JLT Cx; if SW(<) PC PC+Cx
    JGT: if (!`N&&!`Z) PCSet(`PC+c24); // JGT Cx; if SW(>) PC PC+Cx
    JLE: if (`N || `Z) PCSet(`PC+c24); // JLE Cx; if SW(<=) PC PC+Cx
    JGE: if (!`N || `Z) PCSet(`PC+c24); // JGE Cx; if SW(>=) PC PC+Cx
    JMP: `PC = `PC+c24; // JMP Cx; PC <= PC+Cx
    JALR: begin retValSet(a, `PC); PCSet(Rb); end // JALR Ra,Rb; Ra<=PC; PC<=Rb
    BAL: begin `LR = `PC; `PC = `PC+c24; end // BAL Cx; LR<=PC; PC<=PC+Cx
    JSUB: begin retValSet(14, `PC); PCSet(`PC+c24); end // JSUB Cx; LR<=PC; PC
`=<=PC+Cx
    RET: begin PCSet(Ra); end // RET; PC <= Ra
    default :
        $display("%4dns %8x : OP code %8x not support", $stime, pc0, op);
    endcase
    if (`IE && `I && (`IOE && `IO || `I1E && `I1 || `I2E && `I2)) begin
        `EPC = `PC;
        next_state = Fetch;
        inExe = 0;
    end else
        next_state = MemAccess;
    end
    MemAccess: begin

```

(continues on next page)

(continued from previous page)

```

case (op)
ST, SB, SH  :
    memWriteEnd();                      // write memory complete
    endcase
    next_state = WriteBack;
end
WriteBack: begin // Read/Write finish, close memory
case (op)
LB, LBu  :
    memReadEnd(R[a]);                //read memory complete
LH, LHu  :
    memReadEnd(R[a]);
LD  : begin
    memReadEnd(R[a]);
    if (`D)
        $display("%4dns %8x : %8x m[%-04x+%-04x]=%8x SW=%8x", $stime, pc0,
                  ir, R[b], c16, R[a], `SW);
    end
    endcase
case (op)
LB  : begin
    if (R[a] > 8'h7f) R[a]=R[a] | 32'hffff80;
end
LH  : begin
    if (R[a] > 16'h7fff) R[a]=R[a] | 32'hffff8000;
end
endcase
case (op)
MULT, MULTu, DIV, DIVu, MTHI, MTLO :
    if (`D)
        $display("%4dns %8x : %8x HI=%8x LO=%8x SW=%8x", $stime, pc0, ir, HI,
                  LO, `SW);
ST : begin
    if (`D)
        $display("%4dns %8x : %8x m[%-04x+%-04x]=%8x SW=%8x", $stime, pc0,
                  ir, R[b], c16, R[a], `SW);
    if (R[b]+c16 == `IOADDR) begin
        outw(R[a]);
    end
end
SB : begin
    if (`D)
        $display("%4dns %8x : %8x m[%-04x+%-04x]=%c SW=%8x, R[a]=%8x",
                  $stime, pc0, ir, R[b], c16, R[a][7:0], `SW, R[a]);
    if (R[b]+c16 == `IOADDR) begin
        if (`LE)
            outc(R[a][7:0]);
        else
            outc(R[a][7:0]);
    end
end
MFC0, MTC0 :

```

(continues on next page)

(continued from previous page)

```

if (`D)
    $display("%4dns %8x : %8x R[%02d]=-8x C0R[%02d]=-8x SW=%8x",
             $stime, pc0, ir, a, R[a], a, C0R[a], `SW);
COMOV :
if (`D)
    $display("%4dns %8x : %8x C0R[%02d]=-8x C0R[%02d]=-8x SW=%8x",
             $stime, pc0, ir, a, C0R[a], b, C0R[b], `SW);
default :
if (`D) // Display the written register content
    $display("%4dns %8x : %8x R[%02d]=-8x SW=%8x", $stime, pc0, ir,
             a, R[a], `SW);
endcase
if (`PC < 0) begin
    $display("total cpu cycles = %-d", cycles);
    $display("RET to PC < 0, finished!");
    $finish;
end
next_state = Fetch;
end
endcase
end endtask

always @(posedge clock) begin
if (inExe == 0 && (state == Fetch) && (`IE && `I) && (`IOE && `IO)) begin
// software int
`M = `IRQ;
taskInterrupt(`IRQ);
m_en = 0;
state = Fetch;
end else if (inExe == 0 && (state == Fetch) && (`IE && `I) &&
            ((`I1E && `I1) || (`I2E && `I2))) begin
`M = `IRQ;
taskInterrupt(`IRQ);
m_en = 0;
state = Fetch;
end else if (inExe == 0 && itype == `RESET) begin
// Condition itype == `RESET must after the other `IE condition
taskInterrupt(`RESET);
`M = `RESET;
state = Fetch;
end else begin
`ifdef TRACE
`D = 1; // Trace register content at beginning
`endif
taskExecute();
state = next_state;
end
pc = `PC;
cycles = cycles + 1;
end
endmodule

```

(continues on next page)

```

module memory0(input clock, reset, en, rw, input [1:0] m_size,
               input [31:0] abus, dbus_in, output [31:0] dbus_out,
               output cfg);
    reg [31:0] mconfig [0:0];
    reg [7:0] m [0:`MEMSIZE-1];
    reg [31:0] data;

    integer i;

`define LE mconfig[0][0:0] // Endian bit, Big Endian:0, Little Endian:1

initial begin
// erase memory
    for (i=0; i < `MEMSIZE; i=i+1) begin
        m[i] = `MEMEMPTY;
    end
// load config from file to memory
    $readmemh("cpu0.config", mconfig);
// load program from file to memory
    $readmemh("cpu0.hex", m);
// display memory contents
`ifdef TRACE
    for (i=0; i < `MEMSIZE && (m[i] != `MEMEMPTY || m[i+1] != `MEMEMPTY || m[i+2] != `MEMEMPTY || m[i+3] != `MEMEMPTY); i=i+4) begin
        $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
    end
`endif
end

always @(clock or abus or en or rw or dbus_in)
begin
    if (abus >= 0 && abus <= `MEMSIZE-4) begin
        if (en == 1 && rw == 0) begin // r_w==0:write
            data = dbus_in;
            if (`LE) begin // Little Endian
                case (m_size)
`BYTE: {m[abus]} = dbus_in[7:0];
`INT16: {m[abus], m[abus+1]} = {dbus_in[7:0], dbus_in[15:8]};
`INT24: {m[abus], m[abus+1], m[abus+2]} =
{dbus_in[7:0], dbus_in[15:8], dbus_in[23:16]};
`INT32: {m[abus], m[abus+1], m[abus+2], m[abus+3]} =
{dbus_in[7:0], dbus_in[15:8], dbus_in[23:16], dbus_in[31:24]};
                endcase
            end else begin // Big Endian
                case (m_size)
`BYTE: {m[abus]} = dbus_in[7:0];
`INT16: {m[abus], m[abus+1]} = dbus_in[15:0];
`INT24: {m[abus], m[abus+1], m[abus+2]} = dbus_in[23:0];
`INT32: {m[abus], m[abus+1], m[abus+2], m[abus+3]} = dbus_in;
                endcase
            end
        end else if (en == 1 && rw == 1) begin // r_w==1:read
            if (`LE) begin // Little Endian

```

(continues on next page)

(continued from previous page)

```

case (m_size)
`BYTE: data = {8'h00,      8'h00,      8'h00,      m[abus]};
`INT16: data = {8'h00,      8'h00,      m[abus+1],  m[abus]};
`INT24: data = {8'h00,      m[abus+2],  m[abus+1],  m[abus]};
`INT32: data = {m[abus+3], m[abus+2],  m[abus+1],  m[abus]};
endcase
end else begin // Big Endian
case (m_size)
`BYTE: data = {8'h00 , 8'h00,      8'h00,      m[abus] };
`INT16: data = {8'h00 , 8'h00,      m[abus],    m[abus+1]};
`INT24: data = {8'h00 , m[abus],   m[abus+1],  m[abus+2]};
`INT32: data = {m[abus], m[abus+1], m[abus+2], m[abus+3]};
endcase
end
end else
data = 32'hZZZZZZZZ;
end else
data = 32'hZZZZZZZZ;
end
assign dbus_out = data;
assign cfg = mconfig[0][0:0];
endmodule

module main;
reg clock;
reg [2:0] itype;
wire [2:0] tick;
wire [31:0] pc, ir, mar, mdr, dbus;
wire m_en, m_rw;
wire [1:0] m_size;
wire cfg;

cpu0 cpu(.clock(clock), .itype(itype), .pc(pc), .tick(tick), .ir(ir),
.mar(mar), .mdr(mdr), .dbus(dbus), .m_en(m_en), .m_rw(m_rw), .m_size(m_size),
.cfg(cfg));

memory0 mem(.clock(clock), .reset(reset), .en(m_en), .rw(m_rw),
.m_size(m_size), .abus(mar), .dbus_in(mdr), .dbus_out(dbus), .cfg(cfg));

initial
begin
clock = 0;
itype = `RESET;
`TIMEOUT $finish;
end

always #10 clock=clock+1;

endmodule

```

Ibdex/verilog/Makefile

```
#TRACE=-D TRACE
all:
    iverilog ${TRACE} -o cpu0Is cpu0.v
    iverilog ${TRACE} -D CPU0II -o cpu0IIs cpu0.v

.PHONY: clean
clean:
    rm -rf cpu0.hex cpu0Is cpu0IIs
    rm -f *~ cpu0.config
```

Since the Cpu0 Verilog machine supports both big-endian and little-endian modes, the memory and CPU modules communicate this configuration through a dedicated wire.

The endian information is stored in the ROM of the memory module. Upon system startup, the memory module reads this configuration and sends the endian setting to the CPU via the connected wire.

This mechanism is implemented according to the following code snippet:

Ibdex/verilog/cpu0.v

```
assign cfg = mconfig[0][0:0];
...
wire cfg;

cpu0 cpu(.clock(clock), .itype(itype), .pc(pc), .tick(tick), .ir(ir),
.mar(mar), .mdr(mdr), .dbus(dbus), .m_en(m_en), .m_rw(m_rw), .m_size(m_size),
.cfg(cfg));

memory0 mem(.clock(clock), .reset(reset), .en(m_en), .rw(m_rw),
.m_size(m_size), .abus(mar), .dbus_in(mdr), .dbus_out(dbus), .cfg(cfg));
```

13.2 Verify backend

Now let's compile `ch_run_backend.cpp` as shown below. Since code size grows from low to high addresses and the stack grows from high to low addresses, the `$sp` register is set to `0x7ffffc`. This is because `cpu0.v` is assumed to use `0x80000` bytes of memory.

Ibdex/input/start.h

```
#ifndef _START_H_
#define _START_H_

#include "config.h"

#define SET_SW \
asm("andi $sw, $zero, 0"); \
asm("ori $sw, $sw, 0x1e00"); // enable all interrupts

#define initRegs() \
```

(continues on next page)

(continued from previous page)

```

asm("addiu $1, $zero, 0"); \
asm("addiu $2, $zero, 0"); \
asm("addiu $3, $zero, 0"); \
asm("addiu $4, $zero, 0"); \
asm("addiu $5, $zero, 0"); \
asm("addiu $t9, $zero, 0"); \
asm("addiu $7, $zero, 0"); \
asm("addiu $8, $zero, 0"); \
asm("addiu $9, $zero, 0"); \
asm("addiu $10, $zero, 0"); \
SET_SW; \
asm("addiu $fp, $zero, 0");

#endif

```

Ibdex/input/boot.cpp

```

#include "start.h"

// boot:
asm("boot:");
// _start:
asm("jmp 12"); // RESET: jmp RESET_START;
asm("jmp 4"); // ERROR: jmp ERR_HANDLE;
asm("jmp 4"); // IRQ: jmp IRQ_HANDLE;
asm("jmp -4"); // ERR_HANDLE: jmp ERR_HANDLE; (loop forever)

// RESET_START:
initRegs();
asm("addiu $gp, $ZERO, 0");
asm("addiu $lr, $ZERO, -1");

INIT_SP;
asm("mfco $3, $pc");
asm("addiu $3, $3, 0x8"); // Assume main() entry point is at the next next
                           // instruction.
asm("jr $3");
asm("nop");

```

Ibdex/input/print.h

```

#ifndef _PRINT_H_
#define _PRINT_H_

#include "start.h"

void print_char(const char c);
void dump_mem(unsigned char *str, int n);
void print_string(const char *str);
void print_integer(int x);

```

(continues on next page)

(continued from previous page)

```
#endif
```

Ibdex/input/print.cpp

```
#include "print.h"
#include "itoa.cpp"

// For memory IO
void print_char(const char c)
{
    char *p = (char*)IOADDR;
    *p = c;

    return;
}

void print_string(const char *str)
{
    const char *p;

    for (p = str; *p != '\0'; p++)
        print_char(*p);
    print_char(*p);
    print_char('\n');

    return;
}

// For memory IO
void print_integer(int x)
{
    char str[INT_DIGITS + 2];
    itoa(str, x);
    print_string(str);

    return;
}
```

Ibdex/input/ch_nolld.h

```
#include "debug.h"
#include "boot.cpp"

#include "print.h"

int test_nolld();
```

lbdex/input/ch_nlld.cpp

```
#define TEST_ROXV
#define RUN_ON_VERILOG

#include "print.cpp"

#include "ch4_1_math.cpp"
#include "ch4_1_rotate.cpp"
#include "ch4_1_mult2.cpp"
#include "ch4_1_mod.cpp"
#include "ch4_1_div.cpp"
#include "ch4_2_logic.cpp"
#include "ch7_1_localpointer.cpp"
#include "ch7_1_char_short.cpp"
#include "ch7_1_bool.cpp"
#include "ch7_1_longlong.cpp"
#include "ch7_1_vector.cpp"
#include "ch8_1_ctrl.cpp"
#include "ch8_2_deluselessjmp.cpp"
#include "ch8_2_select.cpp"
#include "ch9_1_longlong.cpp"
#include "ch9_3_vararg.cpp"
#include "ch9_3_stacksave.cpp"
#include "ch9_3_bswap.cpp"
#include "ch9_3_alloc.cpp"
#include "ch11_2.cpp"

// Test build only for the following files on build-run_backend.sh since it
// needs lld linker support.
// Test in build-slink.sh
#include "ch6_1.cpp"
#include "ch9_1_struct.cpp"
#include "ch9_1_constructor.cpp"
#include "ch9_3_template.cpp"
#include "ch12_inherit.cpp"

void test_asm_build()
{
    #include "ch11_1.cpp"
#ifdef CPU032II
    #include "ch11_1_2.cpp"
#endif
}

int test_rotate()
{
    int a = test_rotate_left1(); // rolv 4, 30 = 1
    int b = test_rotate_left(); // rol 8, 30 = 2
    int c = test_rotate_right(); // rorv 1, 30 = 4

    return (a+b+c);
}
```

(continues on next page)

(continued from previous page)

```
int test_nolld()
{
    bool pass = true;
    int a = 0;

    a = test_math();
    print_integer(a); // a = 68
    if (a != 68) pass = false;
    a = test_rotate();
    print_integer(a); // a = 7
    if (a != 7) pass = false;
    a = test_mult();
    print_integer(a); // a = 0
    if (a != 0) pass = false;
    a = test_mod();
    print_integer(a); // a = 0
    if (a != 0) pass = false;
    a = test_div();
    print_integer(a); // a = 253
    if (a != 253) pass = false;
    a = test_local_pointer();
    print_integer(a); // a = 3
    if (a != 3) pass = false;
    a = (int)test_load_bool();
    print_integer(a); // a = 1
    if (a != 1) pass = false;
    a = test_andorxornotcomplement();
    print_integer(a); // a = 13
    if (a != 13) pass = false;
    a = test_setxx();
    print_integer(a); // a = 3
    if (a != 3) pass = false;
    a = test_signed_char();
    print_integer(a); // a = -126
    if (a != -126) pass = false;
    a = test_unsigned_char();
    print_integer(a); // a = 130
    if (a != 130) pass = false;
    a = test_signed_short();
    print_integer(a); // a = -32766
    if (a != -32766) pass = false;
    a = test_unsigned_short();
    print_integer(a); // a = 32770
    if (a != 32770) pass = false;
    long long b = test_longlong();
    print_integer((int)(b >> 32)); // 393307
    if ((int)(b >> 32) != 393307) pass = false;
    print_integer((int)b); // 16777218
    if ((int)(b) != 16777218) pass = false;
    a = test_cmplt_short();
    print_integer(a); // a = -3
```

(continues on next page)

(continued from previous page)

```

if (a != -3) pass = false;
a = test_cmplt_long();
print_integer(a); // a = -4
if (a != -4) pass = false;
a = test_control1();
print_integer(a); // a = 51
if (a != 51) pass = false;
a = test_DelUselessJMP();
print_integer(a); // a = 2
if (a != 2) pass = false;
a = test_movx_1();
print_integer(a); // a = 3
if (a != 3) pass = false;
a = test_movx_2();
print_integer(a); // a = 1
if (a != 1) pass = false;
print_integer(2147483647); // test mod % (mult) from itoa.cpp
print_integer(-2147483648); // test mod % (multu) from itoa.cpp
a = test_sum_longlong();
print_integer(a); // a = 9
if (a != 9) pass = false;
a = test_va_arg();
print_integer(a); // a = 12
if (a != 12) pass = false;
a = test_stacksaverestore(100);
print_integer(a); // a = 5
if (a != 5) pass = false;
a = test_bswap();
print_integer(a); // a = 0
if (a != 0) pass = false;
a = test_alloc();
print_integer(a); // a = 31
if (a != 31) pass = false;
a = test_inlineasm();
print_integer(a); // a = 49
if (a != 49) pass = false;

return pass;
}

```

lbdex/input/ch_run_backend.cpp

```

#include "ch_nolld.h"

int main()
{
    bool pass = true;
    pass = test_nolld();

    return pass;
}

```

(continues on next page)

(continued from previous page)

```
}
```

```
#include "ch_nolld.cpp"
```

Index/input/functions.sh

```
prologue() {
    if [ $ARG_NUM == 0 ]; then
        echo "usage: bash $sh_name cpu_type endian"
        echo "  cpu_type: cpu032I or cpu032II"
        echo "  endian: eb (big endian, default) or el (little endian)"
        echo "for example:"
        echo "  bash build-slinker.sh cpu032I be"
        exit 1;
    fi
    if [ $CPU != cpu032I ] && [ $CPU != cpu032II ]; then
        echo "1st argument is cpu032I or cpu032II"
        exit 1
    fi

    OS=`uname -s`
    echo "OS =" ${OS}

    TOOLDIR=~/llvm/test/build/bin
    CLANG=~/llvm/test/build/bin/clang

    CPU=$CPU
    echo "CPU =" "${CPU}"

    if [ "$ENDIAN" != "" ] && [ $ENDIAN != el ] && [ $ENDIAN != eb ]; then
        echo "2nd argument is eb (big endian, default) or el (little endian)"
        exit 1
    fi
    if [ $ENDIAN == eb ]; then
        ENDIAN=
    fi
    echo "ENDIAN =" "${ENDIAN}"

    bash clean.sh
}

isLittleEndian() {
    echo "ENDIAN = "$ENDIAN"
    if [ "$ENDIAN" == "LittleEndian" ] ; then
        LE="true"
    elif [ "$ENDIAN" == "BigEndian" ] ; then
        LE="false"
    else
        echo "!ENDIAN unknown"
        exit 1
    fi
}
```

(continues on next page)

(continued from previous page)

```

}

elf2hex() {
    ${TOOLDIR}/llvm-objdump -elf2hex -le=$LE a.out > ../verilog/cpu0.hex
    if [ $LE == "true" ] ; then
        echo "1 /* 0: big endian, 1: little endian */ > ../verilog/cpu0.config
    else
        echo "0 /* 0: big endian, 1: little endian */ > ../verilog/cpu0.config
    fi
    cat ../verilog/cpu0.config
}

epilogue() {
    endian=`${TOOLDIR}/llvm-readobj -h a.out|grep "DataEncoding"|awk '{print $2}'`  

    isLittleEndian;
    elf2hex;
}

```

Ibdex/input/build-run_backend.sh

```

#!/usr/bin/env bash

# for example:
# bash build-run_backend.sh cpu032I el
# bash build-run_backend.sh cpu032II eb

source functions.sh

sh_name=build-run_backend.sh
ARG_NUM=$#
CPU=$1
ENDIAN=$2

DEFFLAGS=""
if [ "$CPU" == cpu032II ] ; then
    DEFFLAGS="${DEFFLAGS}" -DCPU032II"
fi
echo ${DEFFLAGS}

prologue;

# ch8_2_select_global_pic.cpp just for compile build test only, without running
# on verilog.
$CLANG ${DEFFLAGS} -target mips-unknown-linux-gnu -c ch8_2_select_global_pic.cpp \
-emit-llvm -o ch8_2_select_global_pic.bc
${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=pic \
-filetype=obj ch8_2_select_global_pic.bc -o ch8_2_select_global_pic.cpu0.o

$CLANG ${DEFFLAGS} -target mips-unknown-linux-gnu -c ch_run_backend.cpp \
-emit-llvm -o ch_run_backend.bc
echo "${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=static \

```

(continues on next page)

(continued from previous page)

```
-filetype=obj -enable-cpu0-tail-calls ch_run_backend.bc -o ch_run_backend.cpu0.o"  
${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=static \  
-filetype=obj -enable-cpu0-tail-calls ch_run_backend.bc -o ch_run_backend.cpu0.o  
# print must at the same line, otherwise it will spilt into 2 lines  
${TOOLDIR}/llvm-objdump --section=.text -d ch_run_backend.cpu0.o | tail -n +8| awk \  
'{print /* " $1 " */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" " $9" " $10  
"\t*/"\}' \  
> ../verilog/cpu0.hex  
  
ENDIAN=`${TOOLDIR}/llvm-readobj -h ch_run_backend.cpu0.o|grep "DataEncoding"|awk '  
`{print $2}`'  
isLittleEndian;  
  
if [ $LE == "true" ] ; then  
    echo "1 /* 0: big ENDIAN, 1: little ENDIAN */" > ../verilog/cpu0.config  
else  
    echo "0 /* 0: big ENDIAN, 1: little ENDIAN */" > ../verilog/cpu0.config  
fi  
cat ../verilog/cpu0.config
```

To run program without linker implementation at this point, the `boot.cpp` must be set at the beginning of code, and the `main()` of `ch_run_backend.cpp` comes immediately after it.

Let's run `Chapter11_2/` with `llvm-objdump -d` for input file `ch_run_backend.cpp` to generate the hex file via `build-run_bacekend.sh`, then feed the hex file to `cpu0`'s Verilog simulator to get the output result as below.

Remind that `ch_run_backend.cpp` has to be compiled with the option `clang -target mips-unknown-linux-gnu` since the example code `ch9_3_vararg.cpp`, which uses vararg, needs to be compiled with this option. Other example codes have no differences between this option and the default option.

```
JonathantekiiMac:input Jonathan$ pwd  
/Users/Jonathan/llvm/test/lbdex/input  
JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032I eb  
JonathantekiiMac:input Jonathan$ cd ..//verilog cd ..//verilog  
JonathantekiiMac:input Jonathan$ pwd  
/Users/Jonathan/llvm/test/lbdex/verilog  
JonathantekiiMac:verilog Jonathan$ make  
JonathantekiiMac:verilog Jonathan$ ./cpu0Is  
WARNING: cpu0Is.v:386: $readmemh(cpu0.hex): Not enough words in the file for the  
taskInterrupt(001)  
68  
7  
0  
0  
253  
3  
1  
13  
3  
-126  
130  
-32766  
32770
```

(continues on next page)

(continued from previous page)

```
393307
16777218
3
4
51
2
3
1
2147483647
-2147483648
15
5
0
31
49
total cpu cycles = 50645
RET to PC < 0, finished!
```

```
JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032II eb
JonathantekiiMac:input Jonathan$ cd ../verilog
JonathantekiiMac:verilog Jonathan$ ./cpu0IIs
...
total cpu cycles = 48335
RET to PC < 0, finished!
```

The “total CPU cycles” are calculated in this Verilog simulator to allow performance review of both the backend compiler and the CPU.

Only CPU cycles are counted in this implementation, as I/O cycle times are unknown.

As explained in Chapter “Control Flow Statements”, `cpu032II`, which uses instructions `slt` and `beq`, performs better than `cmp` and `jeq` in `cpu032I`.

The instruction `jmp` has no delay slot, making it preferable for use in dynamic linker implementations.

You can trace memory binary code and changes to destination registers at every instruction execution by unmarking `TRACE` in the Makefile, as shown below:

Ibdex/verilog/Makefile

```
TRACE=-D TRACE
```

```
JonathantekiiMac:raw Jonathan$ ./cpu0IIs
WARNING: cpu0.v:386: $readmemh(cpu0.hex): Not enough words in the file for the
requested range [0:28671].
00000000: 2600000c
00000004: 26000004
00000008: 26000004
0000000c: 26fffffc
00000010: 09100000
00000014: 09200000
...
taskInterrupt(001)
1530ns 00000054 : 02ed002c m[28620+44] == -1           SW=00000000
```

(continues on next page)

(continued from previous page)

```
1610ns 00000058 : 02bd0028 m[28620+40] =0 SW=00000000
...
RET to PC < 0, finished!
```

As shown in the result above, `cpu0.v` dumps the memory content after reading the input file `cpu0.hex`. Next, it runs instructions from address 0 and prints each destination register value in the fourth column.

The first column is the timestamp in nanoseconds. The second column is the instruction address. The third column is the instruction content.

Most of the example codes discussed in previous chapters are verified by printing variables using `print_integer()`.

Since the `cpu0.v` machine is written in Verilog, it is assumed to be capable of running on a real FPGA device (though I have not tested this myself). The actual output hardware interface or port depends on the specific output device, such as RS232, speaker, LED, etc. You must implement the I/O interface or port and wire your I/O device accordingly when programming an FPGA.

By running the compiled code on the Verilog simulator, the compiled result from the Cpu0 backend and the total CPU cycles can be verified and measured.

Currently, this Cpu0 Verilog implementation does not support pipeline architecture. However, based on the instruction set, it can be extended to a pipelined model.

The cycle time of the pipelined Cpu0 model is expected to be more than 1/5 of the “total CPU cycles” shown above, due to dependencies between instructions.

Although the Verilog simulator is slow for running full system programs and does not count cycles for cache and I/O operations, it provides a simple and effective way to validate CPU design ideas in the early development stages using small program patterns.

Creating a full system simulator is complex. While the Wiki website⁷ provides tools for building simulators, doing so requires significant effort.

To generate `cpu032I` code with little-endian format, you can run the following command. The script `build-run_backend.sh` writes the endian configuration to `../verilog/cpu0.config` as shown below.

```
JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032I el
```

`./verilog/cpu0.config`

```
1 /* 0: big endian, 1: little endian */
```

The following files test more features.

Ibdex/input/ch_nolld2.h

```
#include "debug.h"
#include "boot.cpp"

#include "print.h"

int test_nolld2();
```

⁷ https://en.wikipedia.org/wiki/Computer_architecture_simulator

Ibdex/input/ch_nolld2.cpp

```
#include "print.cpp"

#include "ch9_3_alloc.cpp"

int test_nolld2()
{
    bool pass = true;
    int a = 0;

    a = test_alloc();
    print_integer(a); // a = 31
    if (a != 31) pass = false;
    return pass;
}
```

Ibdex/input/ch_run_backend2.cpp

```
#include "ch_nolld2.h"

int main()
{
    bool pass = true;
    pass = test_nolld2();

    return pass;
}

#include "ch_nolld2.cpp"
```

Ibdex/input/build-run_backend2.sh

```
#!/usr/bin/env bash

source functions.sh

sh_name=build-run_backend.sh
ARG_NUM=$#
CPU=$1
ENDIAN=$2

DEFFLAGS=""
if [ "$arg1" == cpu032II ] ; then
    DEFFLAGS=${DEFFLAGS} -DCPU032II"
fi
echo ${DEFFLAGS}

prologue;
```

(continues on next page)

(continued from previous page)

```
 ${CLANG} ${DEFFLAGS} -c ch_run_backend2.cpp \
 -emit-llvm -o ch_run_backend2.bc
 ${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=static \
 -filetype=obj ch_run_backend2.bc -o ch_run_backend2.cpu0.o
 ${TOOLDIR}/llvm-objdump -d ch_run_backend2.cpu0.o | tail -n +8 | awk \
 '{print /* $1 */ $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" \
 " $9" " $10 "\t*/"}' > ../verilog/cpu0.hex

ENDIAN=`${TOOLDIR}/llvm-readobj -h ch_run_backend2.cpu0.o | grep "DataEncoding" | awk \
'{print $2}'`  

isLittleEndian;

if [ $LE == "true" ] ; then
    echo "1 /* 0: big endian, 1: little endian */" > ../verilog/cpu0.config
else
    echo "0 /* 0: big endian, 1: little endian */" > ../verilog/cpu0.config
fi
cat ../verilog/cpu0.config
```

```
JonathanekiiMac:input Jonathan$ bash build-run_backend.sh cpu032II el
...
JonathanekiiMac:input Jonathan$ cd ../verilog
JonathanekiiMac:verilog Jonathan$ ./cpu0II.s
...
31
...
```

13.3 Other LLVM-Based Tools for Cpu0 Processor

You can find the Cpu0 ELF linker implementation based on `lld`, which is the official LLVM linker project, as well as `elf2hex`, which is modified from the `llvm-objdump` driver, at the following website:

```
http://jonathan2251.github.io/lbt/index.html
```

CHAPTER
FOURTEEN

THE CONCEPT OF GPU COMPILER

- *Concept in Graphics and Systems*
 - *3D Modeling*
 - * *Animation*
 - * *Node-Editor (shaders generator)*
 - *Node-Editor*
 - *Code Generation from Node-Editor*
 - * *3D Modeling Tools*
 - *Graphics HW and SW Stack*
 - * *HW Block Diagram*
 - * *SW Stack and Data Flow*
 - * *Pixels Displaying*
 - * *The Role and Purpose of Shaders*
 - *Basic geometry in computer graphics*
 - * *Color*
 - * *Transformation*
 - * *Projection*
 - * *Cross Product*
- *OpenGL*
 - *Example of OpenGL program*
 - *3D Rendering*
 - * *Animation Parameters*
 - * *3D Rendering Pipeline*
 - * *Mobile GPU 3D Rendering*
 - *TBDR — Tile-Based Deferred Rendering*
 - *TBDR Rendering*
 - * *Mesh-Shader Pipeline*

- * *Animation Example*
- *GLSL (GL Shader Language)*
 - * *Background*
 - * *Examples*
 - * *Goals*
 - * *GLSL vs. C: Feature Overview*
 - * *GLSL Qualifiers by Shader Stage*
- *OpenGL Shader Compiler*
- *GPU Architecture*
 - *GPU Hardware Units*
 - *SM (SIMT)*
 - * *SM Hardware*
 - * *SM Scheduling*
 - * *SIMT and SPMD Pipelines*
 - *Processor Units and Memory Hierarchy in NVIDIA GPU*
 - *Memory Subsystem*
 - * *Address Coalescing and Gather-scatter*
 - * *VRAM dGPU*
 - * *RegLess-style architectures*
 - *Specialized Units*
 - * *Geometry Units*
 - * *Rasterization Units*
 - * *Texture Mapping Units (TMUs)*
 - * *Render Output Units (ROPs)*
 - *System Features —Buffers*
- *Software Structure*
 - *General purpose GPU*
 - * *Mapping data in GPU*
 - * *Work between CPU and GPU in Cuda*
 - *OpenCL, Vulkan and spir-v*
 - * *Summary*
 - *Unified IR Conversion Flows*
 - * *Graphics and OpenCL Compilation*
 - * *Graphics Compilation Flow (Microsoft DirectX & OpenGL)*
 - *References*

- * *ML and GPU Compilation*
 - *NVIDIA IR Conversion Flow*
 - *AMD IR Conversion Flow*
 - *ARM IR Conversion Flow*
 - *Imagination Technologies IR Conversion Flow*
 - *Comparison Summary*
 - *References*
- *Accelerate ML/DL on OpenCL/SYCL*
- *Open Sources*

Basically, a CPU is a SISD (Single Instruction Single Data) architecture in each core. The multimedia instructions in CPUs are smaller-scale forms of SIMD (Single Instruction Multiple Data), while GPUs are large-scale SIMD processors, capable of coloring millions of image pixels in just a few milliseconds.

Since 2D and 3D graphic processing offers great potential for parallel data processing, GPU hardware typically includes tens of thousands of functional units per chip, as seen in products by NVIDIA and other manufacturers.

This chapter provides an overview of how 3D animation is created and executed on a CPU+GPU system. Following that, it introduces GPU compilers and hardware features relevant to graphics applications. Finally, it explains how GPUs have taken on more computational tasks traditionally handled by CPUs, through the GPGPU (General-Purpose computing on Graphics Processing Units) concept and the emergence of related standards.

Website: Basic Theory of 3D Graphics with OpenGL¹.

14.1 Concept in Graphics and Systems

14.1.1 3D Modeling

By creating 3D models with triangles or quads on a surface, the model is formed using a polygon mesh². This mesh consists of all the vertices shown in the first image as Fig. 14.1.

After applying smooth shading³, the vertices and edge lines are covered with color (or the edges are visually removed—edges never actually have black outlines). As a result, the model appears much smoother³.

Furthermore, after texturing (texture mapping), the model looks even more realistic⁴.

Animation

To understand how animation works for a 3D model, please refer to the video here⁵. According to the video on skeleton animation, joints are positioned at different poses and assigned timing (keyframes), as illustrated in Fig. 14.2.

In this series of videos, you will see 3D modeling tools generating Java code instead of C/C++ code calling OpenGL API and shaders. This is because Java can call OpenGL API through a wrapper library⁶.

Animation flow

¹ https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

² <https://www.quora.com/Which-one-is-better-for-3D-modeling-Quads-or-Tris>

³ <https://en.wikipedia.org/wiki/Shading>

⁴ https://en.wikipedia.org/wiki/Texture_mapping

⁵ <https://www.youtube.com/watch?v=f3Cr8Yx3GGA>

⁶ https://en.wikipedia.org/wiki/Java_OpenGL

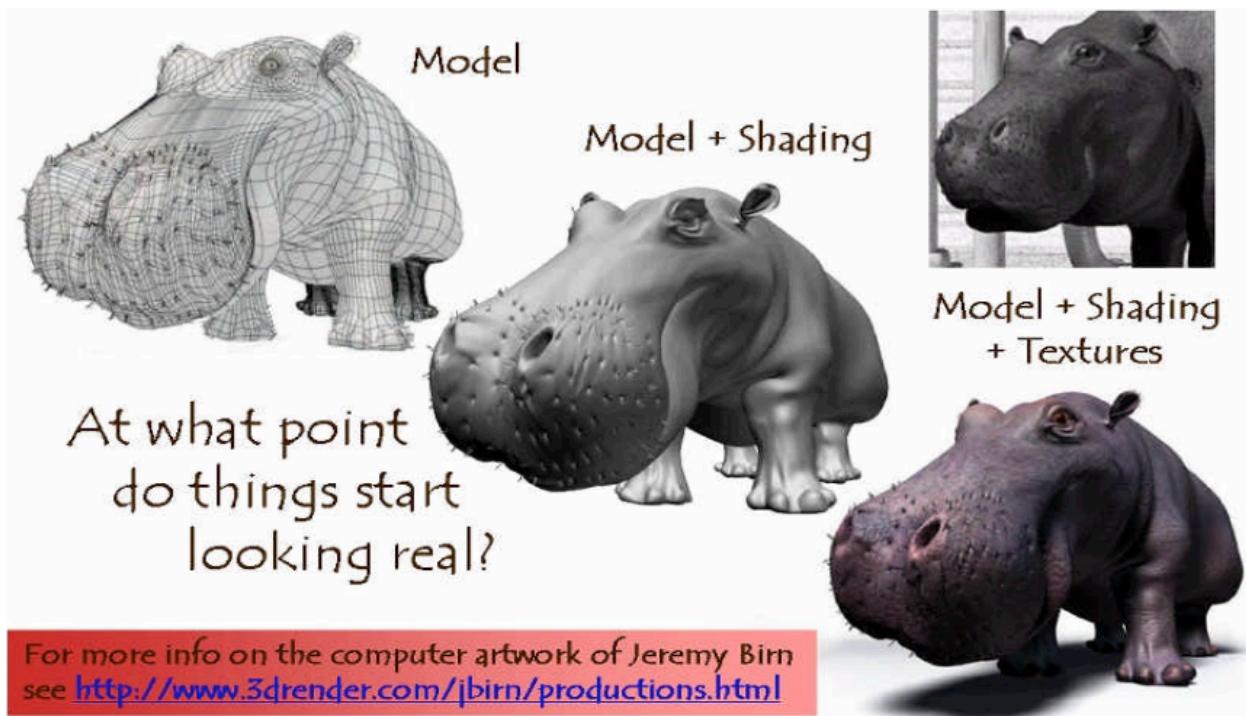


Fig. 14.1: Creating 3D model and texturing

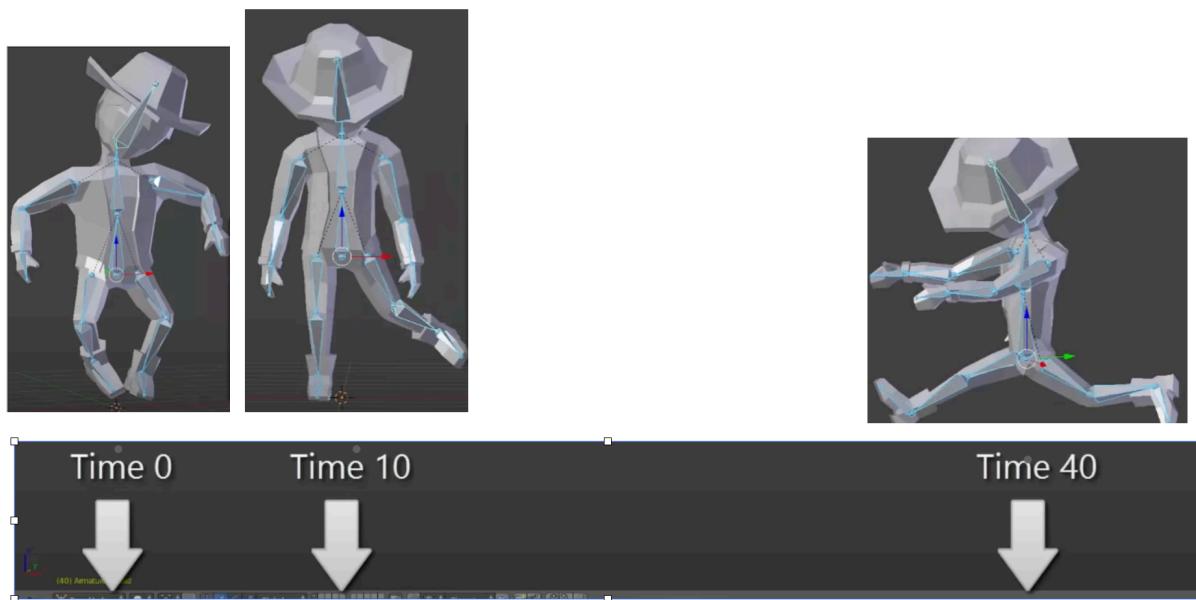
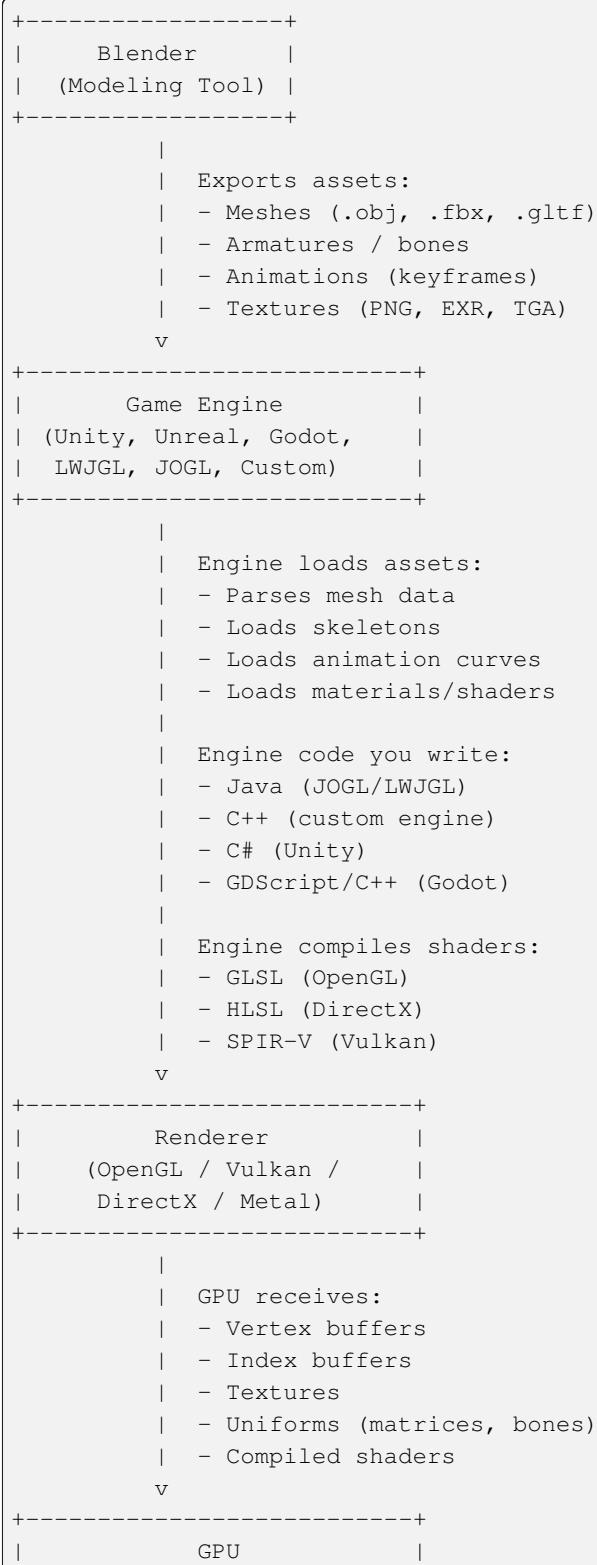


Fig. 14.2: Set time point at keyframes

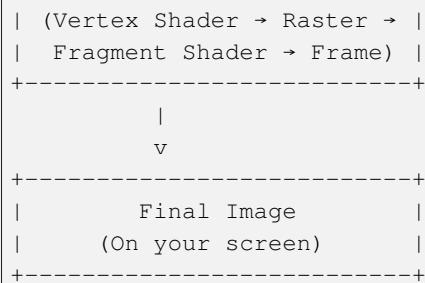
Every modern 3D animation tool comes with its own **built-in render engine**, and often more than one. In 3D game design, **game engines (Unity, Unreal, Godot)** use real-time engines for real-time animation.

Pipeline: Blender → Engine → OpenGL



(continues on next page)

(continued from previous page)



Note

3D modeling tools do store animation and movement data

—but they do NOT store any rendering or API-specific code.

Game engines do store animation data

—but programmers still write the logic that plays, blends, and controls those animations.

Animation Data vs. Movement Speed in Games

List the animation types in a table for inclusion in this book.

Table 14.1: Animation Types

Animation Type	What Moves	Description	GPU Requirement
Transform Animation	Object transform	The entire mesh moves as a rigid body using position, rotation, and scale. No vertex-level deformation occurs.	Optional (fixed-function or shaders)
Skinning	Vertex positions	Vertices are blended by bone matrices to deform the mesh (arms bending, legs walking). Requires per-vertex matrix blending.	Requires shaders
Morph Target Animation	Vertex positions	Vertices blend between multiple stored shapes (facial expressions, muscle bulges). Uses morph weights to interpolate.	Requires shaders
Procedural Deformation	Vertex positions	Vertices are modified by mathematical functions (wind, waves, noise, squash-and-stretch). Driven by time or simulation parameters.	Requires shaders

Example: Walking Animation: Skinning + Transform Animation

When a character walks in a game, the animation is produced by two different systems working together:

- 1. Skinning (Bone Animation)** Skinning is responsible for deforming the mesh. It drives the motion of limbs such as legs, arms, spine, and feet. Without skinning, the character would move as a rigid statue with no bending or articulation.
- 2. Transform Animation (Rigid-Body Movement)** Transform animation moves the entire character through the world. This includes translation, rotation, and root motion. Without transform animation, the character would walk in place without actually moving forward.

Both systems are required to create a complete walking animation:

- **Skinning** provides the internal limb motion.
- **Transform animation** provides the external world-space movement.

Together, they produce the final effect of a character walking naturally through the environment.

The following explains what animation data 3D modeling tools store, what game engines store, and what programmers must implement manually. It also clarifies the relationship between animation curves, movement speed, and gameplay logic.

1. What 3D Modeling Tools Actually Store

3D modeling and animation tools such as Blender, Maya, and 3ds Max store **animation data**, not gameplay logic.

They **do store**:

- Keyframes (frame 0, frame 10, frame 24, etc.)
- Bone transforms at each keyframe
- Interpolation curves (Bezier, linear, quaternion)
- Animation duration (e.g., 1.2 seconds)
- Frame rate (e.g., 24 fps)
- Skeleton hierarchy
- Skin weights (vertex-to-bone influences)
- Optional root bone motion (displacement over time)

Modeling tools produce **data**, not rendering code and not gameplay rules in OpenGL/DirectX code.

Example:

```
Bone "Arm" rotation at frame 0 = (0°, 0°, 0°)
Bone "Arm" rotation at frame 10 = (45°, 0°, 0°)
```

2. What Game Engines Actually Store

Game engines such as Unity, Unreal Engine, Godot, or custom engines store and manage animation data, but still do not define gameplay movement speed.

They **do store**:

- Animation clips
- State machines (Idle → Walk → Run)
- Blend trees
- Transition rules
- Animation events
- Curves for rotation, scaling, and root motion

Again: data, not OpenGL/DirectX code.

Example:

```
If speed < 0.1 → Idle
If speed > 0.1 → Walk
If speed > 4.0 → Run
```

This is engine logic, not GPU code.

Game engines interpret animation data but rely on programmer logic to control how characters move in the world.

3. What Programmers Must Implement

Programmers write the **logic** that uses animation data to move objects.

Examples:

In Unity (C#)

```
animator.SetFloat("speed", playerVelocity);
...
float speed = 3.5f;
transform.position += direction * speed * Time.deltaTime;
```

In a custom engine (C++/OpenGL)

```
shader.setMatrix("boneMatrices[0]", boneMatrix);
...
float velocity = 3.5f;
position += velocity * deltaTime;
```

In JOGL/LWJGL (Java)

```
glUniformMatrix4fv(boneLocation, false, boneMatrixBuffer);
...
float velocity = 3.5f;
position += velocity * deltaTime;
```

Programmers write:

Programmers implement:

- Movement speed
 - E.g. Set the value for speed or velocity as the code above.
- Acceleration and deceleration
- Physics integration
- AI movement
- Player input
- Animation blending logic
- Uploading bone matrices to the GPU
- GLSL shader code for skinning

Animation data is *used* by code, not replaced by it.

4. Root Motion vs. Movement Speed

Some animations include **root motion**, where the root bone moves forward during a walk cycle. Modeling tools export this as bone displacement over time, but they still do **not** define speed.

Example:

If the root bone moves 1 meter in 0.5 seconds, the engine can compute:

```
speed = 1m / 0.5s = 2 m/s
```

However:

- Blender does not store “2 m/s”
- The engine derives speed from displacement
- Programmers decide whether to use root motion or in-place animation

5. Summary Table

Concept	Stored in Blender?	Stored in Engine?	Controlled by Programmer?
Keyframes	Yes	Yes	No
Bone transforms	Yes	Yes	No
Animation length	Yes	Yes	No
Movement speed	No	Yes (derived)	Yes
Physics movement	No	Yes	Yes
AI movement	No	Yes	Yes

6. Final Clarification

- **3D modeling tools store animation timing, not gameplay speed.**
- **Game engines store animation clips, not movement speed.**
- **Programmers control movement speed, physics, and gameplay behavior.**
- **No tool generates JOGL/OpenGL/Vulkan/DirectX code.**
- **All rendering API calls are written by engine developers or by you in a custom engine.**

Example for accelerating playing

Animation Speed vs Engine Rendering (5x Speed)

The following table shows how animation playback, movement speed, and GPU rendering interact when the gameplay speed is multiplied by five. The animation remains 24 fps internally, but its playback time advances five times faster. The GPU continues to render at 60 fps and samples the animation at the current time.

Property	Original Value	After 5x Speed
Animation FPS (baked)	24 fps	24 fps (unchanged)
Animation Playback Speed	1x	5x
Steps per Second	6 steps/sec	30 steps/sec
Movement Speed	6 m/sec	30 m/sec
GPU Rendering FPS	60 fps	60 fps
Engine Playing Frames (What GPU Displays)	Samples animation at 60 fps	Samples animation at 60 fps (skips/interpolates intermediate animation frames)

Summary:

- The animation does **not** become 120 fps; it is simply played 5x faster.
- The runner appears to take **30 steps per second** and move **30 meters per second**.
- The GPU still renders **60 frames per second**.
- The engine **samples** the animation at each rendered frame, so it effectively displays every fifth animation sample, using interpolation for smoothness. For this case, it may display **1 out of 2 animation frames** from 3D modeling.

Node-Editor (shaders generator)

- 3D animation tools (Blender, Maya, Houdini) use render engines and node editors for materials, lighting, and effects.
- Game engines (Unity, Unreal, Godot) use real-time engines and node editors for shaders, VFX, and sometimes logic.

A node editor defines the **entire material** that is applied to the surface of a 3D object. The shader generated from the node graph runs on **every pixel** (fragment) of the object's surface. In this sense, the node editor controls the **whole surface**, not only a specific region.

However, the node graph can include **masks**, **textures**, **vertex colors**, or **procedural patterns** that allow the artist to specify which *parts* of the surface receive a particular effect. These masks do not limit the shader to only part of the surface; instead, they instruct the shader how to behave differently across different regions.

Node-Editor

Example

Let's say you want:

- rust only on the edges
- metal everywhere else

In the node editor:

1. Load a rust texture
2. Load a metal texture
3. Use a mask (curvature or hand-painted)
4. Mix them using a Mix node

The shader still runs on the whole surface, but the mask tells it:

- “Use rust here”
- “Use metal here”

In summary:

- The node editor defines the **full material** for the **entire surface**.
- Artists can use masks or textures to **target specific areas** within that surface.
- The shader still executes globally, but its **output varies** based on the mask inputs.

Thus, a node editor controls the whole surface, while masks determine how different parts of that surface are shaded.

For 3D video game engines, the only case where mask data is inside the model file is vertex colors. Everything else lives in textures or material/shader assets.

Procedural Rust on Edges Using Shader Nodes

To demonstrate how to create a **rust-on-edges** material using Blender's shader node editor as shown in Fig. 14.3. The goal is to reproduce the effect commonly used on metal containers: clean metal on flat surfaces and rust accumulation along exposed edges.

The technique relies on three core ideas:

1. Detecting edges using the *Bevel* or *Pointiness* attribute.

⁷ <https://odysee.com/@jsabbott:d/how-to-make-procedural-edge-wear-in:2>

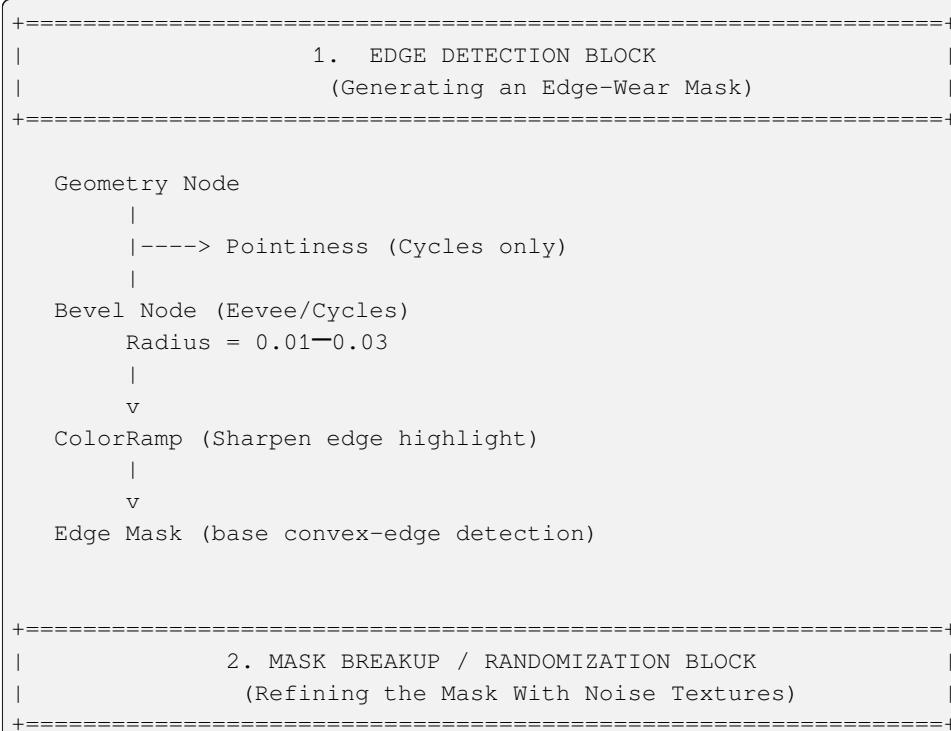


Fig. 14.3: An example to Rust on Edges Using Shader Nodes^{[Page 672, 7](#)}.

2. Creating a mask that isolates only the worn edges.
3. Blending a rust material with a metal material using that mask.

This workflow is fully procedural and does not require painting or external textures.

Procedural Edge Wear Node Graph (ASCII Diagram) to create Fig. 14.3 in video^{[Page 672, 7](#)} includes 1. edge detection, 2. mask breakup, 3. material creation, and 4. final blending as follows:



(continues on next page)

(continued from previous page)

```
Noise Texture (Scale 5—15)
|
v
ColorRamp (optional shaping)
|
v
Multiply Node <----- Edge Mask
|
v
ColorRamp (final threshold control)
|
v
Final Edge Wear Mask

material creation
+=====+
|           3. MATERIAL BLOCK          |
|           (Base Material Structure) |
+=====+

METAL MATERIAL:
Principled BSDF
Metallic = 1.0
Roughness = 0.2—0.4

RUST MATERIAL:
Principled BSDF
Base Color = orange/brown
Roughness = 0.7—1.0
Optional Noise → color variation
Optional Bump → rust height

+=====+
|           4. BLENDING BLOCK          |
|           (Blending Metal and Rust Materials) |
+=====+

Metal BSDF -----
|
v
Mix Shader ----> Material Output
^
|
|
Rust BSDF -----
|
|
Final Edge Wear Mask (Fac)
```

Code Generation from Node-Editor

Node-based shader editors are visual tools used in modern game engines and DCC (Digital Content Creation) software. They allow users to build shaders by connecting nodes instead of writing GLSL, HLSL, or Metal code manually. These editors **do generate shader code automatically**.

1. Node-Based Editors Do Generate Shader Code

Node-based shader editors such as:

- Unity Shader Graph
- Unreal Engine Material Editor
- Godot Visual Shader Editor
- Blender Shader Nodes (for Eevee/Cycles)

all **compile the visual node graph into real shader code**.

Depending on the engine, the generated code may be:

- GLSL (OpenGL / Vulkan)
- HLSL (DirectX)
- MSL (Metal)
- SPIR-V (Vulkan intermediate format)

The generated code is usually not shown to the user, but it is compiled and sent to the GPU at runtime.

2. How Users Generate Shader Code with Nodes

The workflow for generating shader code through a node editor typically looks like this:

1. The user opens the shader editor.
2. The user creates nodes representing: - math operations (add, multiply, dot product) - texture sampling - lighting functions - color adjustments - UV transformations
3. The user connects nodes visually to define the shader logic.
4. The engine converts the node graph into an internal shader representation.
5. The engine compiles this representation into platform-specific shader code (GLSL, HLSL, MSL, or SPIR-V).
6. The compiled shader is sent to the GPU and used for rendering.

The user never writes the shader code directly; the editor generates it automatically.

3. Who Is the “User” of Node-Based Editors?

The typical users of node-based shader editors are:

Graphics Designers / Technical Artists

- Primary users.
- They create visual effects, materials, and surface shaders.
- They usually do not write GLSL or HLSL manually.
- Node editors allow them to work visually without programming.

Software Programmers / Graphics Programmers

- Secondary users.
- They may create custom nodes or extend the shader system.

- They write low-level shader code when needed.
- They integrate the generated shaders into the rendering pipeline.

In most workflows:

- **Graphics designers** build the shader visually.
- **The engine** generates the shader code.
- **Programmers** handle advanced logic, optimization, or custom nodes.

4. Summary

- Node-based shader editors **do generate shader code** automatically.
- Users generate shaders by connecting visual nodes rather than writing GLSL/HLSL manually.
- The primary “user” is the **graphics designer or technical artist**.
- Programmers support the system by writing custom nodes or low-level shaders when needed.

The shaders introduction is illustrated in the next section *OpenGL*.

3D Modeling Tools

Every CAD software manufacturer, such as AutoDesk and Blender, has their own proprietary format. To solve interoperability problems, neutral or open source formats were created as intermediate formats to convert between proprietary formats.

Naturally, these neutral formats have become very popular. Two famous examples are STL (with a *.STL* extension) and COLLADA (with a *.DAE* extension). Below is a list showing 3D file formats along with their types.

Table 14.2: 3D file formats⁸

3D file format	Type
STL	Neutral
OBJ	ASCII variant is neutral, binary variant is proprietary
FBX	Proprietary
COLLADA	Neutral
3DS	Proprietary
IGES	Neutral
STEP	Neutral
VRML/X3D	Neutral

The four key features a 3D file can store include the model’s geometry, the model’s surface texture, scene details, and animation of the model⁸.

Specifically, they can store details about four key features of a 3D model, though it’s worth bearing in mind that you may not always take advantage of all four features in all projects, and not all file formats support all four features!

3D printer applications do not support animation. CAD and CAM such as designing airplane does not need feature of scene details.

DAE (Collada) appeared in the video animation above. Collada files belong to a neutral format used heavily in the video game and film industries. It’s managed by the non-profit technology consortium, the Khronos Group.

The file extension for the Collada format is *.dae*. The Collada format stores data using the XML mark-up language.

⁸ <https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/>

The original intention behind the Collada format was to become a standard among 3D file formats. Indeed, in 2013, it was adopted by ISO as a publicly available specification, ISO/PAS 17506. As a result, many 3D modeling programs support the Collada format.

That said, the consensus is that the Collada format hasn't kept up with the times. It was once used heavily as an interchange format for Autodesk Max/Maya in film production, but the industry has now shifted more towards OBJ, FBX, and Alembic^{Page 676, 8}.

14.1.2 Graphics HW and SW Stack

This section provides a more detailed illustration of animation across the software and hardware stacks on both CPU and GPU, and explains how data flows between the CPU, the GPU, and each layer of the software stack.

In the previous section *3D Modeling*, described what information 3D models store and how this information is used to perform animation.

In the incoming section *SW Stack and Data Flow* will describe **how each frame is generated** to display the **movement animation or skinning effects** using the small animation parameters stored in 3D model and sent from CPU.

The the incoming section *Role and Purpose of Shaders* will explain different visual effects can be achieved by **switching shaders** to shappingly different materials across frames.

Reference:

- https://en.wikipedia.org/wiki/Free_and_open-source_graphics_device_driver

HW Block Diagram

The block diagram of the Graphic Processing Unit (GPU) is shown in Fig. 14.4.

The roles of the CPU and GPU in graphic animation are illustrated in Fig. 14.5.

- GPU can't directly read user input from, say, keyboard, mouse, gamepad, or play audio, or load files from a hard drive, or anything like that. In this situation, cannot let GPU handle the animation work¹¹.
- A graphics driver consists of an implementation of the OpenGL state machine and a compilation stack to compile the shaders into the GPU's machine language. This compilation, as well as pretty much anything else, is executed on the CPU, then the compiled shaders are sent to the GPU and are executed by it. (SDL = Simple DirectMedia Layer)¹².

The GPU driver write command and data from CPU to GPU's system memory through PCIe. These commands are called Command Stream Fronted (CSF) in the memory of GPU. A chipset of GPU includes tens of SIMD processors (cores). In order to speedup the GPU driver's processing, the CSF is designed to a simpler form. As result, GPU chipset include MCU (Micro Chip Unit) and specific HW to transfer the CSF into individual data structure for each SIMD processor to execute as Fig. 14.6. The firmware version of MCU is updated by MCU itself usually.

SW Stack and Data Flow

The driver runs on the CPU side as shown in Fig. 14.7. The OpenGL API eventually calls the driver's functions, and the driver executes these functions by issuing commands to the GPU hardware and/or sending data to the GPU.

Even so, the GPU's rendering work, which uses data such as 3D vertices and colors sent from the CPU and stored in GPU or shared memory, consumes more computing power than the CPU.

✓ As section *Animation* and Fig. 14.7. Higher-level libraries and frameworks on top of OpenGL provide animation frameworks and tools to generate OpenGL APIs and shaders from 3D models.

⁹ https://en.wikipedia.org/wiki/Graphics_processing_unit

¹⁰ <https://en.wikipedia.org/wiki/Vulkan>

¹¹ <https://stackoverflow.com/questions/47426655/cpu-and-gpu-in-3d-game-whos-doing-what>

¹² <[https://en.wikipedia.org/wiki/Mesa_\(computer_graphics\)](https://en.wikipedia.org/wiki/Mesa_(computer_graphics))>

¹³ <https://developer.arm.com/documentation/102813/0107/GPU-activity>

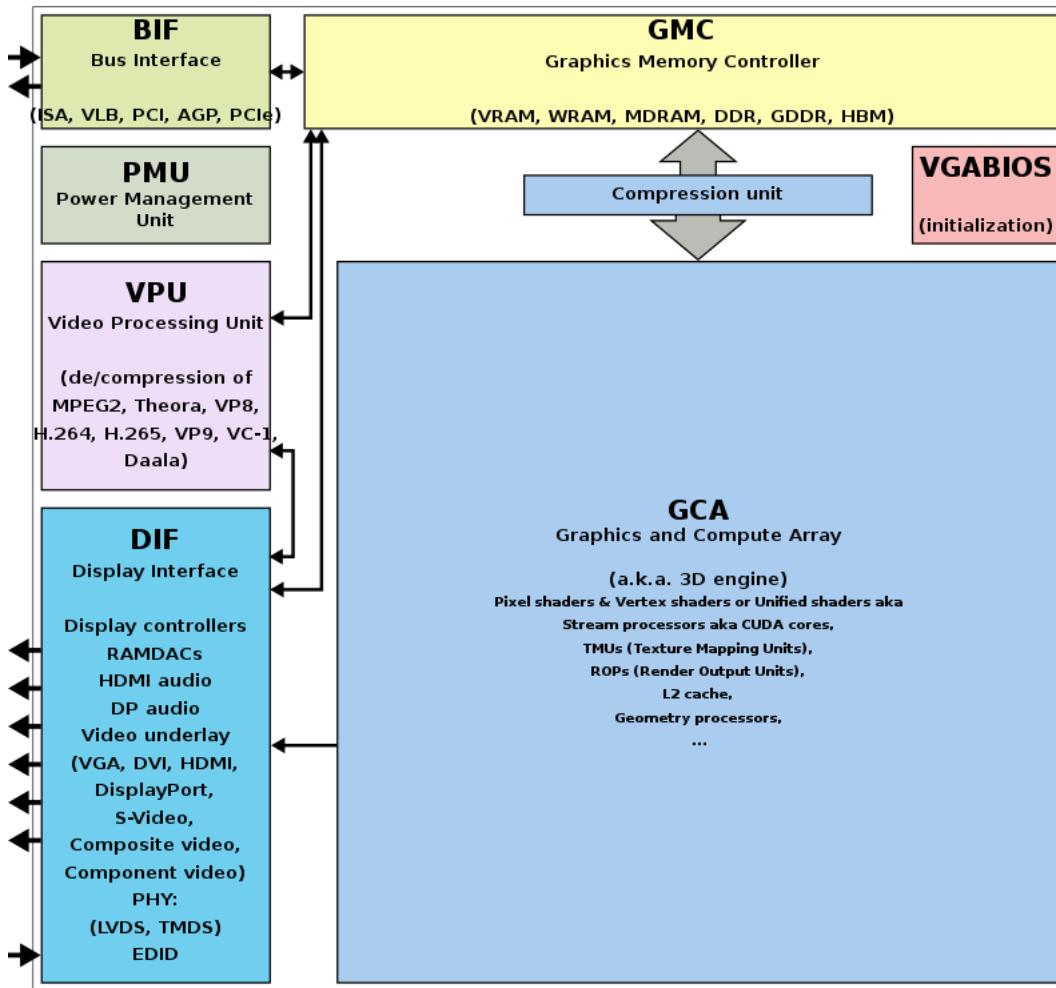


Fig. 14.4: Components of a GPU: GPU has accelerated video decoding and encoding^{Page 677, 9}

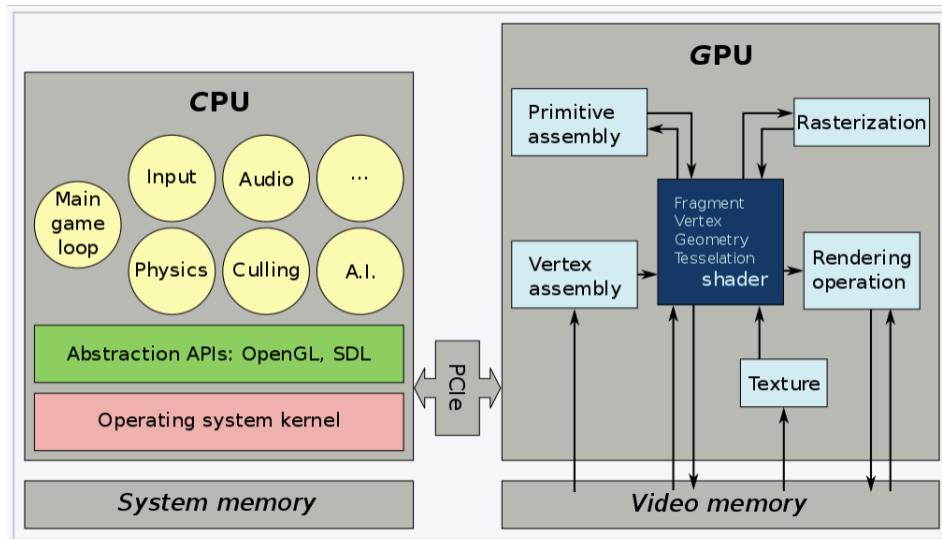


Fig. 14.5: OpenGL and Vulkan are both rendering APIs. In both cases, the GPU executes shaders, while the CPU executes everything else^{Page 677, 10}.

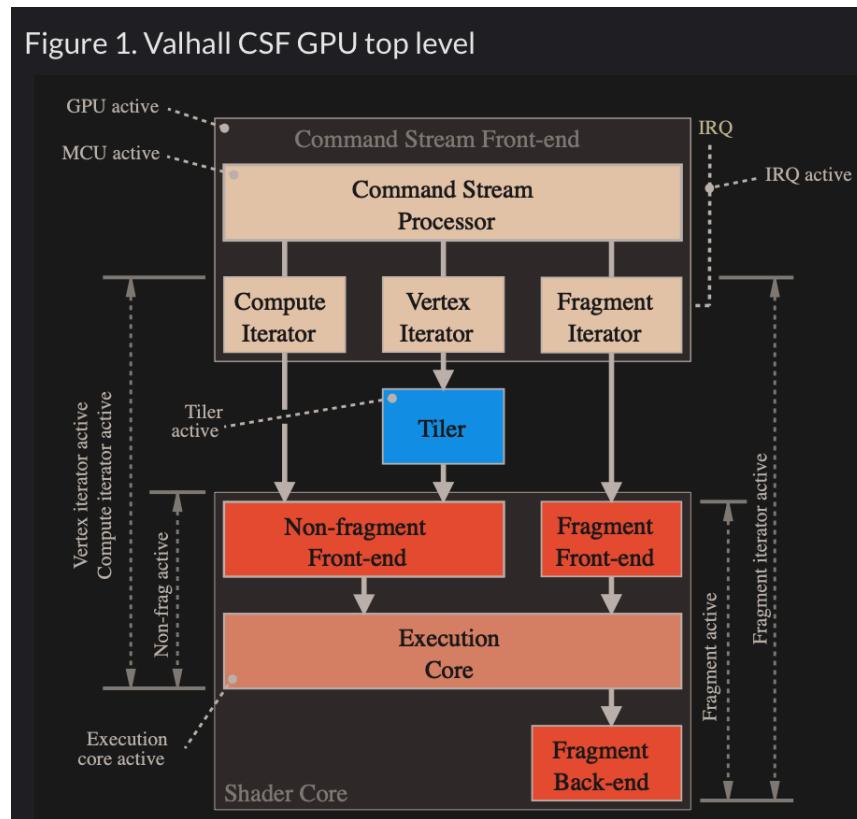


Fig. 14.6: MCU and specific HW circuits to speedup the processing of CSF (Command Stream Frontend)^{Page 677, 13}.

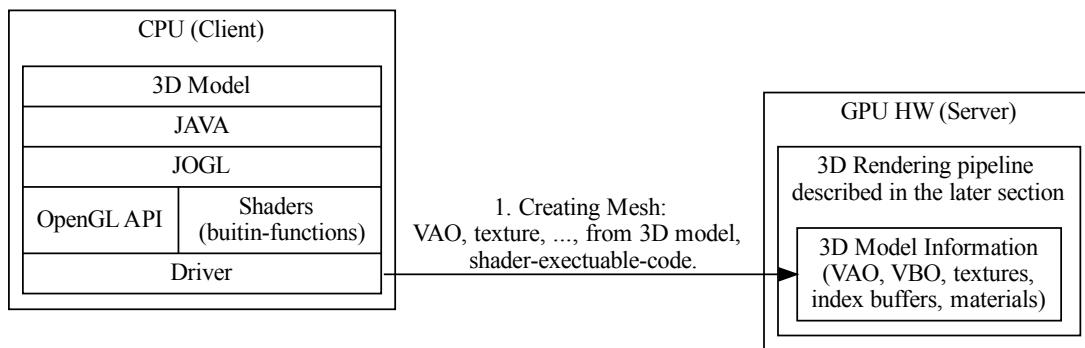


Fig. 14.7: Graphic SW Stack and data flow in initializing graphic model

After the user creates a skeleton and textures for each model and sets keyframe times using a 3D modeling tool, the tool can either generate Java code that calls JOGL (Java OpenGL)^{Page 665, 6} or generate OpenCL APIs directly.

As section *Node-Editor (shaders generator)*, shaders are primarily created by Graphics Designers / Technical Artists and secondly created by Software Programmers / Graphics Programmers.

Shaders may call built-in functions written in Compute Shaders, SPIR-V, or LLVM-IR. LLVM *libclc* is a project for OpenCL built-in functions, which can also be used in OpenGL¹⁴. Like CPU built-ins, new GPU ISAs or architectures must implement their own built-ins or port them from open source projects like *libclc*.

The 3D model on the CPU performs these animations in movement and others by computing each frame from the stored keyframes, as illustrated in animation section *Animation*.

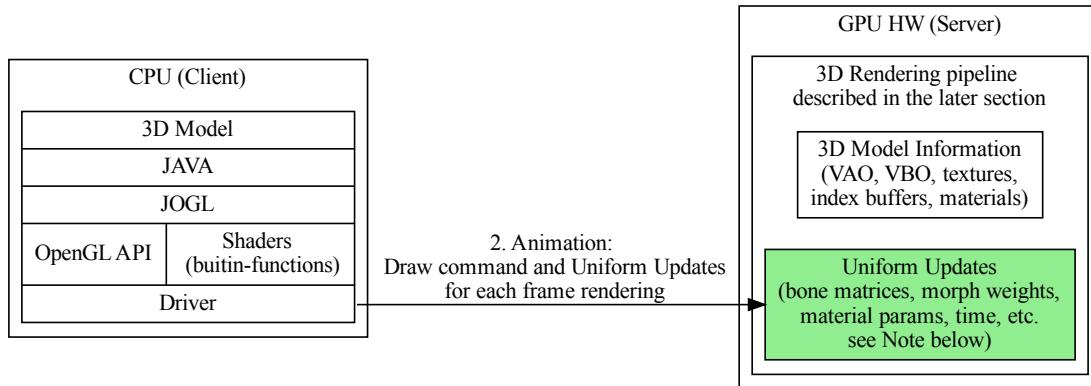


Fig. 14.8: Graphic SW Stack and data flow in rendering

The per-frame data is not the full set of vertices, but rather a small set of animation parameters named **Uniform Updates** as appeared in Fig. 14.8, which are described later.

Note

Bone matrices determine the positions of triangles within a 3D model during animation. This bone transformation data is much **smaller** than the complete mesh of the 3D model. We will provide an example and explain this in more detail in the *Animation Example* section. Because this transformation data is small and constant across all shader pipeline stages, it is stored in the GPU's global memory and can be cached in the **uniform/constant cache** for performance, as illustrated in Fig. 14.48 of *Processor Units and Memory Hierarchy in NVIDIA GPU 81* section.

The CPU updates only these small animation parameters and issues draw command to the GPU server side.

Next, the 3D Rendering-pipeline is illustrated in Fig. 14.9.

The shape of object data are stored in the form of VAOs (Vertex Array Objects) in OpenGL. This will be explained in a later *section OpenGL*. Additionally, OpenGL provides VBOs (Vertex Buffer Objects), which allow vertex array data to be stored in high-performance graphics memory on the server side GPU and enable efficient data transfer¹⁵¹⁶.

¹⁴ <https://libclc.llvm.org>

¹⁵ http://www.songho.ca/opengl/gl_vbo.html

¹⁶ If your models will be rigid, meaning you will not change each vertex individually, and you will render many frames with the same model, you will achieve the best performance not by storing the models in your class, but in vertex buffer objects (VBOs) <https://gamedev.stackexchange.com/questions/19560/what-is-the-best-way-to-store-meshes-or-3d-models-in-a-class>

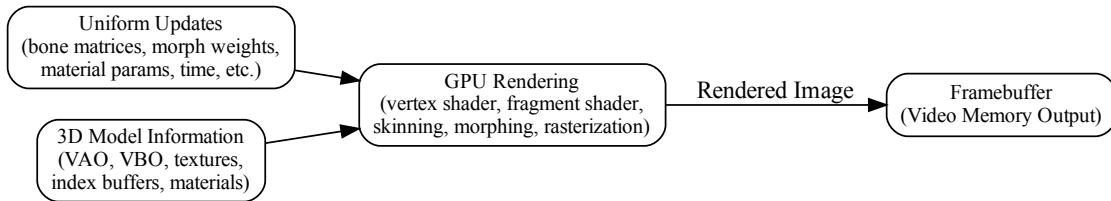


Fig. 14.9: The input and output for GPU rendering

After GPU receives the **Uniform Updates** from CPU, it performs the computationally intensive per-vertex work within the rendering pipeline to generate the **final pixel values** for **each frame** displayed on screen. These final pixel values are collectively referred to the **Rendered Image**.

- ✓ “**Rendered Image**” = the final per-frame output written into the framebuffer. The **Uniform Updates** will be described in detail later.
- ✓ CPU only updates small animation parameters named **Uniform Updates** as appeared in Fig. 14.8; GPU computes the heavy per-vertex work.

As mentioned in the previous section on *animation movement*, 3D modeling tools store Keyframes, bone transforms at each keyframe and related data, and perform animation based on this information.

The CPU updates only the **bone** transformation data ..., rather than updating the entire vertex or mesh data for each animation frame. These updates are very small—on the order of kilobytes rather than megabytes. For each rendered frame, the CPU sends these small updates to the GPU, and the **GPU takes over the animation work from the CPU**. This type of movement animation is called **skinning**, and is illustrated as follows:

Skinning

Skinning is a vertex deformation technique used to animate a mesh by attaching its vertices to a hierarchical skeleton (bones). Each vertex stores one or more bone indices and corresponding weights that describe how strongly each bone influences that vertex.

During animation, the application updates the bone transformation matrices. The vertex shader then computes the final vertex position by blending the transformed positions according to the stored weights. This allows the mesh to bend, twist, and deform smoothly as the skeleton moves.

Skinning does not create new geometry or smooth the surface topology. It only transforms the existing vertices of the mesh. Examples include bending an arm, flapping a wing, or deforming a flexible tube as its bones rotate.

CPU only update high-level animation state, such as:

- Current animation time
- **Bone matrices (small)**
- **Morph weights**
- Material parameters
- Particle emitter settings
- Global uniforms (camera, lights, etc.)

These are tiny updates —kilobytes, not megabytes.

$$finalPosition = \sum_{i=0}^{N-1} \mathbf{weight}_i (\mathbf{boneMatrix}_i \cdot originalPosition)$$

Example: Bending an Arm

Imagine a character's arm mesh. Each vertex in the elbow area has weights like:

- 70% influenced by upper-arm bone
- 30% influenced by lower-arm bone

When the elbow bends:

- Upper-arm bone rotates
- Lower-arm bone rotates
- GPU blends the influence
- The elbow area deforms smoothly

This is skinning.

✓ After the GPU animation, the color pixels are written to framebuffer (video memory). The display device (monitor, LCD, OLED, etc.) fetches these pixels and displays them on the screen. The interface between framebuffer and display device is explained in the next section [Pixels Displaying](#).

Pixels Displaying

The interface between frame buffer and displaying device is shown as [Fig. 14.10](#).

GPU and screen (monitor, LCD, OLED, etc.) use **VSync**, **NVIDIA G-SYNC** or **AMD FreeSync** to prevent **screen tearing**, as described below:

PC with Frame Buffer to LCD Display

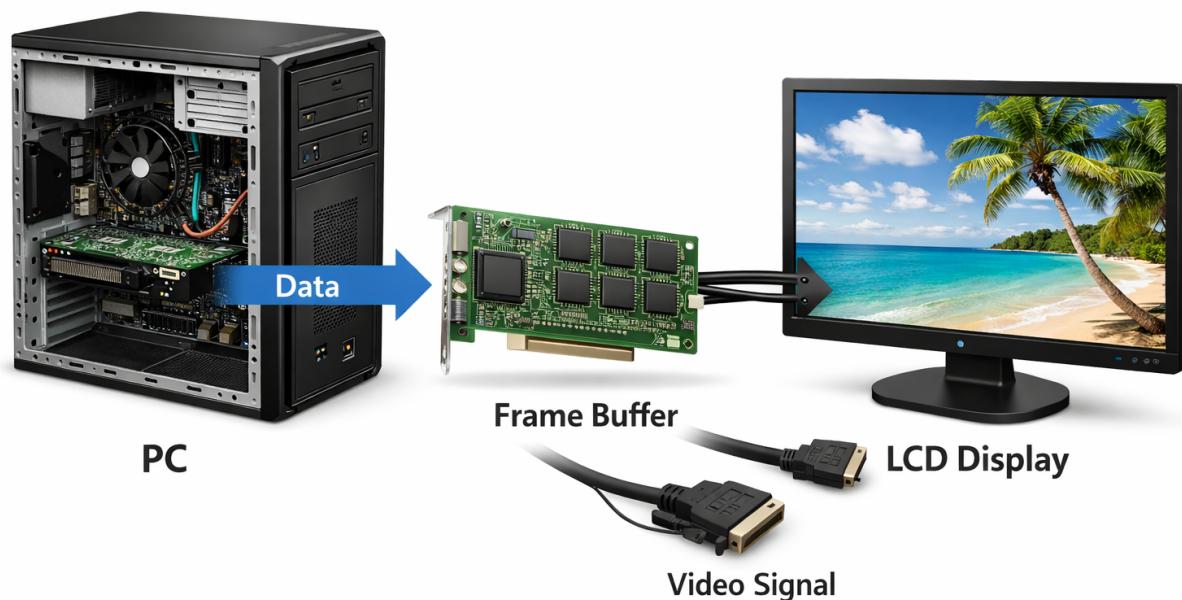
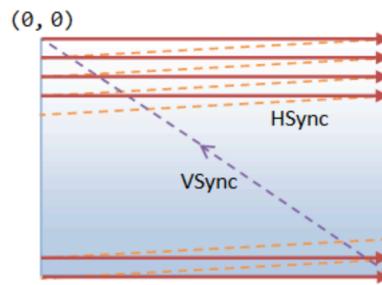


Fig. 14.10: PC with Frame Buffer to LCD Display

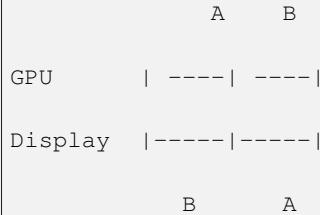


Raster Scan: The display updates its contents row-by-row from top-to-bottom, left-to-right by reading the color value of the pixels from the frame buffer.

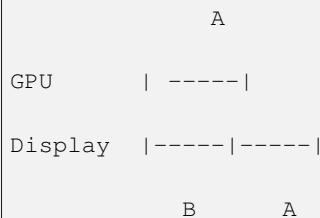
Fig. 14.11: VSync

VSync

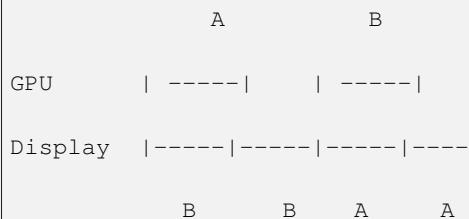
No tearing occurs when the GPU and display operate at the same refresh rate, since the GPU refreshes faster than the display as shown below.



Tearing occurs when the GPU has exact refresh cycles but VSync takes one more cycle than the display as shown below.



To avoid tearing, the GPU runs at half the refresh rate of the display, as shown below.



- Double Buffering

While the display is reading from the frame buffer to display the current frame, we might be updating its contents for the next frame (not necessarily in raster-scan manner). This would result in the so-called tearing, in which the screen shows parts of the old frame and parts of the new frame. This could be resolved by using so-called double buffering. Instead of using a single frame buffer, modern GPU uses two of them: a front buffer and a back buffer. The display reads from the front buffer, while we can write the next frame to the back buffer. When we finish, we signal to GPU to swap the front and back buffer (known as buffer swap or page flip).

- VSync

Double buffering alone does not solve the entire problem, as the buffer swap might occur at an inappropriate time, for example, while the display is in the middle of displaying the old frame. This is resolved via the so-called vertical synchronization (or VSync) at the end of the raster-scan. When we signal to the GPU to do a buffer swap, the GPU will wait till the next VSync to perform the actual swap, after the entire current frame is displayed.

As above text diagram. The most important point is: When the VSync buffer-swap is enabled, you cannot refresh the display faster than the refresh rate of the display!!! If GPU is capable of producing higher frame rates than the display's refresh rate, then GPU can use fast rate without tearing. If GPU has same or less frame rates than display's and you application refreshes at a fixed rate, the resultant refresh rate is likely to be an integral factor of the display's refresh rate, i.e., 1/2, 1/3, 1/4, etc. Otherwise it will cause tearing^{[Page 665, 1](#)}.

- NVIDIA G-SYNC and AMD FreeSync

If your monitor and graphics card both in your customer computer support NVIDIA G-SYNC, you're in luck. With this technology, a special chip in the display communicates with the graphics card. This lets the monitor vary the refresh rate to match the frame rate of the NVIDIA GTX graphics card, up to the maximum refresh rate of the display. This means that the frames are displayed as soon as they are rendered by the GPU, eliminating screen tearing and reducing stutter for when the frame rate is both higher and lower than the refresh rate of the display. This makes it perfect for situations where the frame rate varies, which happens a lot when gaming. Today, you can even find G-SYNC technology in gaming laptops!

AMD has a similar solution called FreeSync. However, this doesn't require a proprietary chip in the monitor. In FreeSync, the AMD Radeon driver, and the display firmware handle the communication. Generally, FreeSync monitors are less expensive than their G-SYNC counterparts, but gamers generally prefer G-SYNC over FreeSync as the latter may cause ghosting, where old images leave behind artifacts^{[17](#)}.

The Role and Purpose of Shaders

The flow for 3D/2D graphic processing is shown in Fig. 14.12.

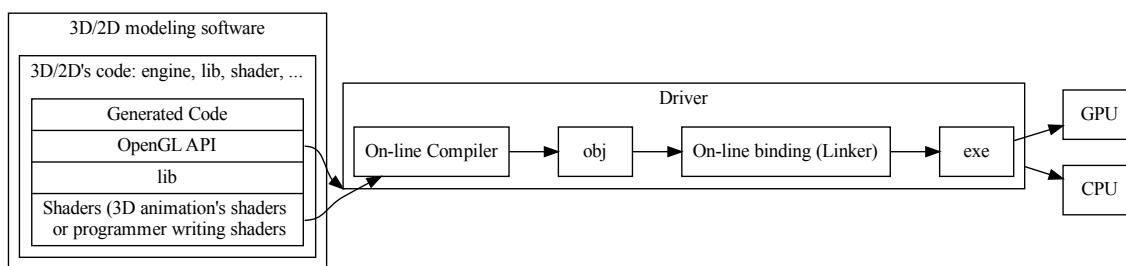


Fig. 14.12: OpenGL Flow

The compiled shaders are sent to the GPU when you call `glLinkProgram()`. That is the moment the driver uploads the compiled shader binaries into GPU-executable form.

¹⁷ <https://www.avadirect.com/blog/frame-rate-fps-vs-hz-refresh-rate/>

The `glLinkProgram()` is called when you finish preparing a shader program —**not when creating a mesh, and not when issuing a draw command**.

When a game actually call `glLinkProgram()` to re-link the shader, the shader need to be compiled and load to GPU.

Usually it is happen in game startup, level load, or creating a new shader variant (e.g., enabling shadows, fog, skinning).

Games switch shaders constantly —sometimes hundreds of times per frame —but they do not re-link them.

When playing a video game, different materials, effects and rendering passes will applying to difference shaders.

Examples of switching shaders:

- When the player enters a snowy biome, ice meshes use the ice shader.
- The axe blade uses a metal PBR shader. Sparks fly when the axe blade hits stone it switch to particle shader.

14.1.3 Basic geometry in computer graphics

This section introduces the basic geometry math used in computer graphics. The complete concept can be found in the book *Computer Graphics: Principles and Practice, 3rd Edition*, authored by John F. et al. However, the book contains over a thousand pages.

It is very comprehensive and may take considerable time to understand all the details.

Color

- Additive colors in light are shown in Fig. 14.13¹⁸¹⁹.
- In the case of paints, additive colors produce shades and become light gray due to the addition of darker pigments²⁰.

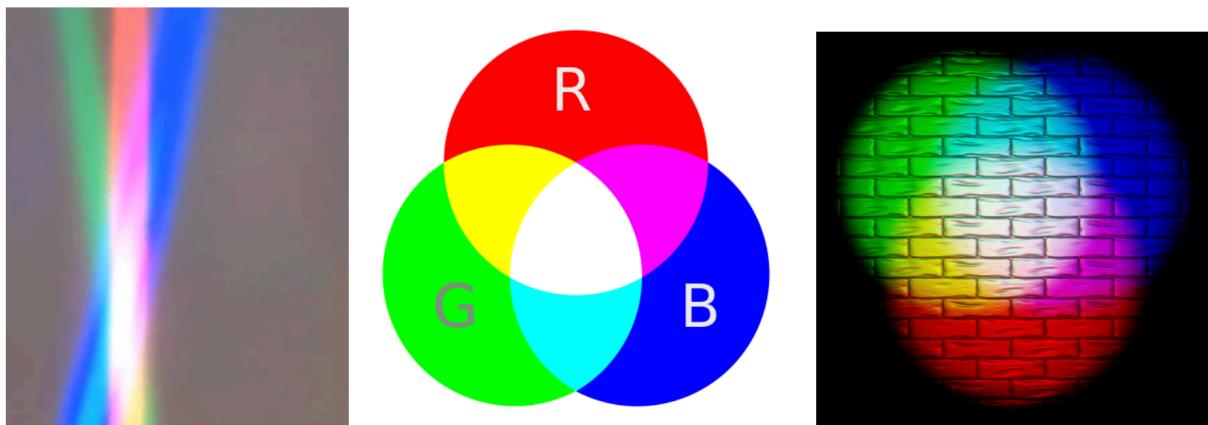


Fig. 14.13: Additive colors in light

Note

Additive colors

I know it doesn't match human intuition. However, additive RGB colors in light combine to produce white light, while additive RGB in paints result in light gray paint. This makes sense because light has no shade. This result stems from the way human eyes perceive color. Without light, no color can be sensed by the eyes.

¹⁸ https://en.wikipedia.org/wiki/RGB_color_model

¹⁹ https://www.youtube.com/watch?v=kEnz_3miiAc

²⁰ <https://www.tiktok.com/@tonesterpaints/video/7059565281227853102>

Computer engineers should understand that exploring the underlying reasons falls into the realms of physics or the biology of the human eye structure.

Transformation

Objects (Triangle/Quad) can be moved in 2D/3D using matrix representation, as explained in this wiki page²¹.

The rotation matrix used is derived from another wiki page²².

Every computer graphics book covers the topic of transformation of objects and their positions in space. Chapter 4 of the *Blue Book: OpenGL SuperBible, 7th Edition* provides a short but useful 40-page description of transformation concepts. It is a good material for understanding the basics.

The following Quaternion Product (Hamilton product) is from the wiki²³ since it is not covered in the book.

$$ij = -ji = k, jk = -kj = i, ki = -ik = j.$$

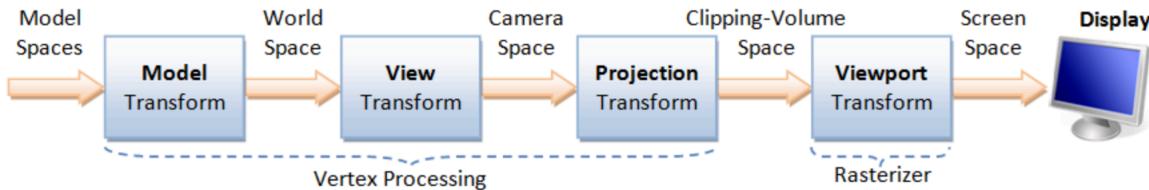
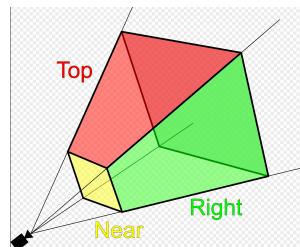


Fig. 14.14: Coordinates Transform Pipeline^{Page 665, 1}

Details for Fig. 14.14 can be found on the website^{Page 665, 1}.

Projection



Only objects within the cone between near and far planes are projected to 2D in perspective projection.

Perspective and orthographic projections (used in CAD tools) from 3D to 2D can be represented by transformation matrices as described in the previous section²⁴.

Cross Product

Both triangles and quads are polygons. So, objects can be formed with polygons in both 2D and 3D. The transformation in 2D or 3D is well covered in almost every computer graphics book. This section introduces the most important concept

²¹ https://en.wikipedia.org/wiki/Transformation_matrix

²² https://en.wikipedia.org/wiki/Rotation_matrix

²³ <https://en.wikipedia.org/wiki/Quaternion>

²⁴ https://en.wikipedia.org/wiki/3D_projection#Perspective_projection

and method for determining inner and outer planes. Then, a point or object can be checked for visibility during 2D or 3D rendering.

Any **area** of a polygon can be calculated by dividing it into triangles or quads. The area of a triangle or quad can be calculated using the cross product in 3D.

The cross product in **3D** is defined by the formula and can be represented with matrix notation, as shown here²⁵.

$$\mathbf{a} \times \mathbf{b} = \|a\| \|b\| \sin(\Theta) n$$

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

The cross product in **2D** is defined by a formula and can be represented with matrix notation, as proven here²⁶²⁷.

$$\mathbf{a} \times \mathbf{b} = \|a\| \|b\| \sin(\Theta)$$

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \end{vmatrix} = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix}$$

After the above matrix form is proven, the antisymmetry property may be demonstrated as follows:

$$a \times b = x \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix} = a_1 b_2 - a_2 b_1 =$$

$$-b_1 a_2 - (-b_2 a_1) = \begin{bmatrix} -b_1 & -b_2 \\ a_1 & a_2 \end{bmatrix} = x \begin{bmatrix} -b \\ a \end{bmatrix} = -b \times a$$

In 2D, any two points from P_i to P_{i+1} can form a vector and determine the inner or outer side.

For example, as shown in Fig. 14.15, Θ is the angle from $P_i P_{i+1}$ to $P_i P'_{i+1} = 180^\circ$.

Using the right-hand rule and counter-clockwise order, any vector $P_i Q$ between $P_i P_{i+1}$ and $P_i P'_{i+1}$, with angle θ such that $0^\circ < \theta < 180^\circ$, indicates the inward direction.



Fig. 14.15: Inward edge normals

Based on this observation, the rule for inward and outward vectors is shown in Fig. 14.15. Facing the same direction as a specific vector, the left side is inward and the right side is outward, as shown in Fig. 14.16.

For each edge $P_i - P_{i+1}$, the inward edge normal is the vector $\mathbf{x} v_i$; the outward edge normal is $- \mathbf{x} v_i$, where $\mathbf{x} v_i$ is the cross-product of v_i , as shown in Fig. 14.15.

²⁵ https://en.wikipedia.org/wiki/Cross_product

²⁶ <https://www.xarg.org/book/linear-algebra/2d-perp-product/>

²⁷ <https://www.nagwa.com/en/explainers/175169159270/>

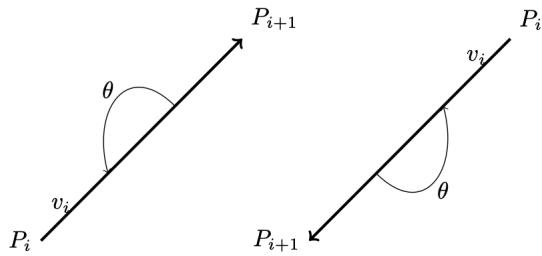


Fig. 14.16: Inward and outward in 2D for a vector.

A polygon can be created from a set of vertices. Suppose (P_0, P_1, \dots, P_n) defines a polygon. The line segments P_0P_1, P_1P_2 , etc., are the polygon's edges. The vectors $v_0 = P_1 - P_0, v_1 = P_2 - P_1, \dots, v_n = P_0 - P_n$ represent those edges.

Using counter-clockwise ordering, the left side is inward. Thus, the inward region of a polygon can be determined.

For a convex polygon with vertices listed in counter-clockwise order, the inward edge normals point toward the interior of the polygon, and the outward edge normals point toward the unbounded exterior. This matches our usual intuition.

However, if the polygon vertices are listed in clockwise order, the interior and exterior definitions are reversed.

This cross product has an important property: going from v to $\times v$ involves a 90° rotation in the same direction as the rotation from the positive x-axis to the positive y-axis.

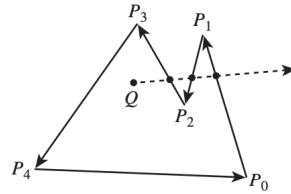


Fig. 14.17: Draw a polygon with vertices counter clockwise

As shown in Fig. 14.17, when drawing a polygon with vectors (lines) in counter-clockwise order, the polygon will be formed, and the two sides of each vector (line) can be identified²⁸.

Furthermore, whether a point is inside or outside the polygon can be determined.

One simple method to test whether a point lies inside or outside a simple polygon is to cast a ray from the point in any fixed direction and count how many times it intersects the edges of the polygon.

If the point is outside the polygon, the ray will intersect its edges an even number of times. If the point is inside the polygon, it will intersect the edges an odd number of times²⁹.

In the same way, by following the counter-clockwise direction to create a 2D polygon step by step, a 3D polygon can be constructed.

As shown in Fig. 14.18 from the wiki^{Page 688, 25}, the inward direction is determined by $a \times b < 0$, and the outward direction is determined by $a \times b > 0$ in OpenGL.

Replacing a and b with x and y , as shown in Fig. 14.19, the positive Z-axis ($z+$) represents the outer surface, while the negative Z-axis ($z-$) represents the inner surface³⁰.

²⁸ Figure 7.19 of Book: Computer graphics principles and practice 3rd edition

²⁹ https://en.wikipedia.org/wiki/Point_in_polygon

³⁰ Normals are used to differentiate the front- and back-face, and for other processing such as lighting. Right-hand rule (or counter-clockwise) is used in OpenGL. The normal is pointing outwards, indicating the outer surface (or front-face). https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

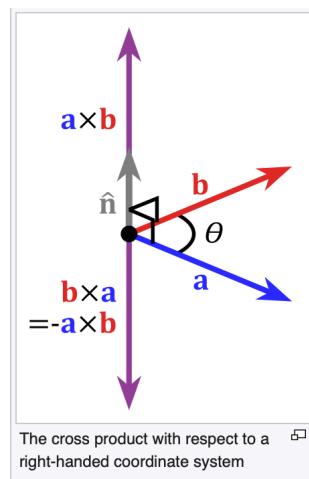


Fig. 14.18: Cross product definition in 3D

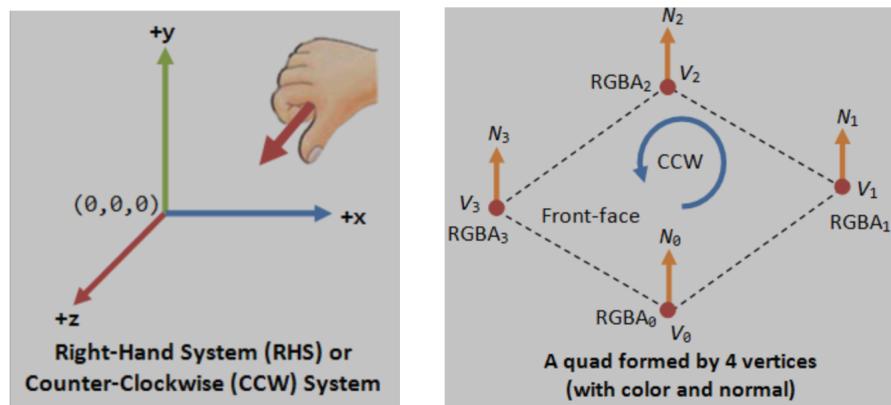


Fig. 14.19: OpenGL pointing outwards, indicating the outer surface (z axis is +)

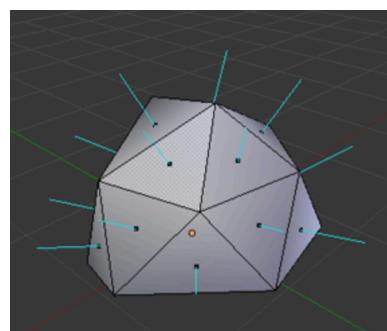


Fig. 14.20: 3D polygon with directions on each plane

When we build every triangle with CCW, then we can defined a consistent outer surface (front face). The Fig. 14.20 shows an example of a 3D polygon created from 2D triangles. The direction of the plane (triangle) is given by the line perpendicular to the plane.

This means:

- ✓ The mesh has a fixed “outer” and “inner”

(based on CCW in object space)

Cast a ray from the 3D point along the X-axis and count how many intersections with the outer object occur. Depending on the number of intersections along each axis (even or odd), you can understand if **the point (or the camera) is inside or outside**³¹.

An odd number means inside, and an even number means outside. As shown in Fig. 14.21, points on the line passing through the object satisfy this rule.

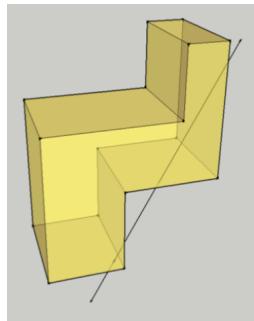


Fig. 14.21: Point is inside or outside of 3D object

How does OpenGL render (draw) the inner face of a triangle?

OpenGL does NOT determine front/back in world space.

When the camera moves to the inner space of a object:

- The projection changes
- The triangle’s screen-space orientation changes
- CCW ↔ CW flips
- So the GPU flips front/back classification

OpenGL uses counter clockwise and pointing outwards as default^{Page 680, 15.}

```
// unit cube
// A cube has 6 sides and each side has 4 vertices, therefore, the total number
// of vertices is 24 (6 sides * 4 verts), and 72 floats in the vertex array
// since each vertex has 3 components (x,y,z) (= 24 * 3)
//   v6-----v5
//   / \     / \
//   v1-----v0 \
//   / \     / \
//   / v7----/-v4
//   / \     / \
//   v2-----v3
```

(continues on next page)

³¹ <https://stackoverflow.com/questions/63557043/how-to-determine-whether-a-point-is-inside-or-outside-a-3d-model-computationally>

(continued from previous page)

```
// vertex position array
GLfloat vertices[] = {
    .5f, .5f, .5f, -.5f, .5f, -.5f, -.5f, .5f, .5f, -.5f, .5f, // v0, v1, v2, v3
    ↪ (front)
    .5f, .5f, .5f, .5f, -.5f, .5f, .5f, -.5f, -.5f, .5f, .5f, -.5f, // v0, v3, v4, v5
    ↪ (right)
    .5f, .5f, .5f, .5f, .5f, -.5f, -.5f, -.5f, .5f, .5f, .5f, // v0, v5, v6, v1
    ↪ (top)
    -.5f, .5f, .5f, -.5f, .5f, -.5f, -.5f, -.5f, -.5f, .5f, // v1, v6, v7, v2
    ↪ (left)
    -.5f, -.5f, -.5f, .5f, -.5f, -.5f, .5f, -.5f, -.5f, .5f, // v7, v4, v3, v2
    ↪ (bottom)
    .5f, -.5f, -.5f, -.5f, -.5f, -.5f, .5f, .5f, .5f, .5f, // v4, v7, v6, v5
    ↪ (back)
};
}
```

From the code above, we can see that OpenGL uses counter-clockwise and pointing outwards as the default. However, OpenGL provides `glFrontFace(GL_CW)` for clockwise winding³².

For a group of objects, a scene graph provides better animation support and saves memory³³.

14.2 OpenGL

14.2.1 Example of OpenGL program

The following example is from the OpenGL Red Book and its example code³⁶⁴².

References/triangles.vert

```
#version 400 core

layout( location = 0 ) in vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
```

References/triangles.frag

```
#version 450 core

out vec4 fColor;
```

(continues on next page)

³² <https://registry.khronos.org/OpenGL-Refpages/gl4/html/glFrontFace.xhtml>

³³ https://en.wikipedia.org/wiki/Scene_graph

³⁶ <http://www.opengl-redbook.com>

⁴² <https://github.com/openglredbook/examples>

(continued from previous page)

```

void main()
{
    fColor = vec4(0.5, 0.4, 0.8, 1.0);
}

```

References/01-triangles.cpp

```

1 //////////////////////////////////////////////////////////////////
2 //
3 //  Triangles.cpp
4 //
5 //////////////////////////////////////////////////////////////////
6
7 #include "vgl.h"
8 #include "LoadShaders.h"
9
10 enum VAO_IDs { Triangles, NumVAOs };
11 enum Buffer_IDs { ArrayBuffer, NumBuffers };
12 enum Attrib_IDs { vPosition = 0 };
13
14 GLuint VAOs[NumVAOs];
15 GLuint Buffers[NumBuffers];
16
17 const GLuint NumVertices = 6;
18
19 //-----
20 //
21 // init
22 //
23
24 void
25 init( void )
26 {
27     glGenVertexArrays( NumVAOs, VAOs ); // Same with glGenVertexArray( NumVAOs,
28     // VAOs );
29     // https://stackoverflow.com/questions/24441430/glgen-vs-glcreate-naming-
30     // convention
31     // Make the new VAO:VAOs[Triangles] active, creating it if necessary.
32     glBindVertexArray( VAOs[Triangles] );
33     // opengl->current_array_buffer = VAOs[Triangles]
34
35     GLfloat vertices[NumVertices][2] = {
36         { -0.90f, -0.90f }, { 0.85f, -0.90f }, { -0.90f, 0.85f }, // Triangle 1
37         { 0.90f, -0.85f }, { 0.90f, 0.90f }, { -0.85f, 0.90f } // Triangle 2
38     };
39
40     glCreateBuffers( NumBuffers, Buffers );
41
42     // Make the buffer the active array buffer.
43     glBindBuffer( GL_ARRAY_BUFFER, Buffers[ArrayBuffer] );
44     // Attach the active VBO:Buffers[ArrayBuffer] to VAOs[Triangles]

```

(continues on next page)

(continued from previous page)

```

43     // as an array of vectors with 4 floats each.
44     // Kind of like:
45     // opengl->current_vertex_array->attributes[attr] = {
46     //     type = GL_FLOAT,
47     //     size = 4,
48     //     data = opengl->current_array_buffer
49     // }
50     // Can be replaced with glVertexArrayVertexBuffer(VAOs[Triangles], Triangles,
51     // buffer[ArrayBuffer], ArrayBuffer, sizeof(vmath::vec2));,_
52     // →glVertexArrayAttribFormat(), ...
53     // in OpenGL 4.5.
54
55     glBufferStorage( GL_ARRAY_BUFFER, sizeof(vertices), vertices, 0 );
56
57     ShaderInfo shaders[] =
58     {
59         { GL_VERTEX_SHADER, "media/shaders/triangles/triangles.vert" },
60         { GL_FRAGMENT_SHADER, "media/shaders/triangles/triangles.frag" },
61         { GL_NONE, NULL }
62     };
63
64     GLuint program = LoadShaders( shaders );
65     glUseProgram( program );
66
67     glVertexAttribPointer( vPosition, 2, GL_FLOAT,
68                           GL_FALSE, 0, BUFFER_OFFSET(0) );
69     glEnableVertexAttribArray( vPosition );
70     // Above two functions specify vPosition to vertex shader at layout (location = 0)
71 }
72
73 //-----
74 // display
75 //
76
77 void
78 display( void )
79 {
80     static const float black[] = { 0.0f, 0.0f, 0.0f, 0.0f };
81
82     glClearBufferfv(GL_COLOR, 0, black);
83
84     glBindVertexArray( VAOs[Triangles] );
85     glDrawArrays( GL_TRIANGLES, 0, NumVertices );
86 }
87
88 //-----
89 //
90 // main
91 //
92
93 #ifdef _WIN32

```

(continues on next page)

(continued from previous page)

```

94 int CALLBACK WinMain(
95     _In_ HINSTANCE hInstance,
96     _In_ HINSTANCE hPrevInstance,
97     _In_ LPSTR     lpCmdLine,
98     _In_ int        nCmdShow
99 )
100 #else
101 int
102 main( int argc, char** argv )
103 #endif
104 {
105     glfwInit();
106
107     GLFWwindow* window = glfwCreateWindow(800, 600, "Triangles", NULL, NULL);
108
109     glfwMakeContextCurrent(window);
110     gl3wInit();
111
112     init();
113
114     while (!glfwWindowShouldClose(window))
115     {
116         display();
117         glfwSwapBuffers(window);
118         glfwPollEvents();
119     }
120
121     glfwDestroyWindow(window);
122
123     glfwTerminate();
124 }
```

Init():

- Generate Vertex Array VAOs and bind VAOs[0].

(glGenVertexArrays(NumVAOs, VAOs); glBindVertexArray(VAOs[Triangles]); glCreateBuffers(NumBuffers, Buffers);)

A vertex-array object holds various data related to a collection of vertices. Those data are stored in buffer objects and managed by the currently bound vertex-array object.

- glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);

Because there are many different places where buffer objects can be in OpenGL, when we bind a buffer, we need to specify what we'd like to use it for. In our example, because we're storing vertex data into the buffer, we use GL_ARRAY_BUFFER. The place where the buffer is bound is known as the binding target.

- According to the counter-clockwise rule in the previous section, triangle primitives are defined in variable *vertices*. After binding OpenGL VBO Buffers[0] to *vertices*, vertex data will be sent to the memory of the server (GPU).

Think of the “active” buffer as just a global variable, and there are a bunch of functions that use the active buffer instead of taking using a parameter. These global state variables are the ugly side of OpenGL⁴³ and can be replaced with *glVertexArrayVertexBuffer()*, *glVertexArrayAttribFormat()*, etc. Then call *glBindVertexArray(vao)* be-

⁴³ <https://stackoverflow.com/questions/21652546/what-is-the-role-of-glbindingvertexarrays-vs-glbindingbuffer-and-what-is-their-relatio>

fore drawing in OpenGL 4.5⁴⁴⁴⁵.

- glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

During GPU rendering, each vertex position will be held in *vPosition* and passed to the “triangles.vert” shader through the *LoadShaders(shaders)* function.

glfwSwapBuffers(window):

- You’ve already used double buffering for animation. Double buffering is done by making the main color buffer have two parts: a front buffer that’s displayed in your window; and a back buffer, which is where you render the new image. When you swap the buffers (by calling *glfwSwapBuffers()*, for example), the front and back buffers are exchanged⁸⁵.

display():

- Bind VAOs[0], set render mode to GL_TRIANGLES and send vertex data to Buffer (gpu memory, OpenGL pipeline). Next, GPU will do rendering pipeline described in next section.

The triangles.vert has input *vPosition* and no output variable, so using *gl_Position* default variable without declaration. The triangles.frag has not defined input variable and has defined output variable *fColor* instead of using *gl_FragColor*.

The “in” and “out” in shaders above are “type qualifier”. A type qualifier is used in the OpenGL Shading Language (GLSL) to modify the storage or behavior of global and locally defined variables. These qualifiers change particular aspects of the variable, such as where they get their data from and so forth⁵⁰.

Though attribute and varying are removed from later version 1.4 of OpenGL, many materials in website using them⁵¹⁵². It’s better to use “in” and “out” to replace them as the following code. OpenGL has a few ways to binding API’s variable with shader’s variable. *glVertexAttrib** as the following code and *glBindAttribLocation()*⁵³, ...

replace attribute and varying with in and out

```
uniform float scale;
layout (location = 0) attribute vec2 position;
// layout (location = 0) in vec2 position;
layout (location = 1) attribute vec4 color;
// layout (location = 1) in vec4 color;
varying vec4 v_color;
// out v_color

void main()
{
    gl_Position = vec4(position*scale, 0.0, 1.0);
    v_color = color;
}
```

```
// OpenGL API
GLfloat attrib[] = { x * 0.5f, x * 0.6f, x* 0.4f, 0.0f };
// Update the value of input attribute 1 : layout (location = 1) in vec4 color
glVertexAttrib4fv(1, attrib);
```

⁴⁴ <https://registry.khronos.org/OpenGL-Refpages/gl4/html/glBindVertexBuffer.xhtml>

⁴⁵ Page 152 of Blue book: OpenGL SuperBible 7th Edition.

⁸⁵ Section of Color Buffer, page 222-223 of book “OpenGL Programming Guide 9th Edition”^{Page 692, 36.}

⁵⁰ <[https://www.khronos.org/opengl/wiki>Type_Qualifier_\(GLSL\)](https://www.khronos.org/opengl/wiki>Type_Qualifier_(GLSL))>

⁵¹ <[https://www.khronos.org/opengl/wiki>Type_Qualifier_\(GLSL\)#Removed_qualifiers](https://www.khronos.org/opengl/wiki>Type_Qualifier_(GLSL)#Removed_qualifiers)>

⁵² <https://github.com/vispy/vispy/issues/242>

⁵³ <[https://www.khronos.org/opengl/wiki/Layout_Qualifier_\(GLSL\)](https://www.khronos.org/opengl/wiki/Layout_Qualifier_(GLSL))>

```

varying vec4 v_color;
// in vec4 v_color;

void main()
{
    gl_FragColor = v_color;
}

```

An OpenGL program is made of two shaders⁴⁸⁴⁹:

- The vertex shader is (commonly) executed once for every vertex we want to draw. It receives some attributes as input, computes the position of this vertex in space and returns it in a variable called `gl_Position`. It also defines some varyings.
- The fragment shader is executed once for each pixel to be rendered. It receives some varyings as input, computes the color of this pixel and returns it in a variable called `fColor`.

Since we have 6 vertices in our buffer, this shader will be executed 6 times by the GPU (once per vertex)! We can also expect all 6 instances of the shader to be executed in parallel, since a GPU have so many cores.

14.2.2 3D Rendering

3D animation is the process of creating moving images by manipulating digital objects within a three-dimensional space. 3D rendering is the process of converting 3D models into 2D images on a computer³⁴.

Based on the previous section of 3D modeling, the 3D modeling tool will generate a 3D vertex model and OpenGL code. Then, programmers may manually modify the OpenGL code and add or update shaders.

In section *SW Stack and Data Flow*, we mentioned the GPU will generate the rendering image for each frame according the 3D Inforamtion and Uniform Updates sent from CPU, and write each of the final frame of data in the form of color pixels to framebuffer (video memory) as Fig. 14.9.

Animation Parameters

✓ CPU only updates small animation parameters named **Uniform Updates** as appeared in Fig. 14.8; GPU computes the heavy per-vertex work.

The 3D animation will trigger the 3D rendering process for each 2D image drawing accordiding the **Uniform Updates**.

The “small animation parameters” updated by the CPU are formally called:

- ✓ Uniform updates
- ✓ Constant buffer updates
- ✓ Per-frame / per-draw constants
- ✓ Bone matrix palette updates (for skinning)
- ✓ Morph weight updates (for morphing)

These are the correct technical terms used in modern graphics pipelines.

↑ The Proper Term: “Uniform Updates”

The most accurate and universal name is:

- ✓ Uniform updates

⁴⁸ <https://engineering.monstar-lab.com/en/post/2022/03/01/Introduction-To-GPUs-With-OpenGL/>

⁴⁹ <https://glumpy.github.io/modern-gl.html>

³⁴ https://en.wikipedia.org/wiki/3D_rendering

or

- ✓ Updating uniform buffers

Because the CPU is updating uniform data that the GPU reads during shading.

Examples of uniform data:

- bone matrices
- morph weights
- animation time
- material parameters
- camera matrices
- light parameters

These are small, constant-for-the-draw values.

↳ More Specific Terms Used in Game Engines

1. Animation Parameters

Used in animation systems:

- “animation parameters”
- “skinning parameters”
- “bone palette”
- “morph weights”

2. Per-Frame Constants

Used in engine architecture:

- “frame constants”
- “per-frame constant buffer”
- “global shader constants”

3. Per-Draw Constants

Used in render pipelines:

- “per-draw uniform block”
- “per-object constant buffer”
- “material constant buffer”

↳ In Modern APIs (GL, Vulkan, DirectX)

OpenGL

- Uniforms
- Uniform Buffer Objects (UBOs)
- Shader Storage Buffer Objects (SSBOs)

DirectX

- Constant Buffers (CBuffers)

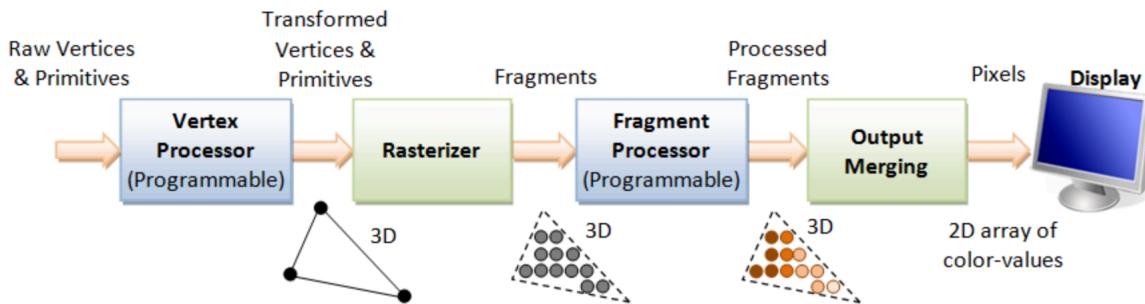
Vulkan

- Descriptor sets
- Uniform buffers

All refer to the same concept: small CPU-updated data that the GPU reads during shading.

3D Rendering Pipeline

The steps are shown in Fig. 14.22.



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z) , and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Fig. 14.22: 3D Graphics Rendering Pipeline^{Page 665, 1}

- A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.

From the previous Fig. 14.8 and Fig. 14.9 in section *SW Stack and Data Flow*, we introduce the 3D animation data are classified as follows:

- **Vertex Data = 3D model information** (the mesh (geometry), such as VBO/VAO)
- **Animation Parameters = per-frame uniform updates** (transforms, bone matrices, camera, materials, ...)

The complete steps of 3D Rendering pipeline, **excluding animation** are shown in the Fig. 14.23 from the OpenGL website³⁵ and in the Fig. 14.24. The website also provides a description for each stage.

The Red Book and Blue Book show only **Vertex Specification** and **Vertex Data** in the rendering flow because they **never show Animation Parameters as part of the rendering flow**. The animation flow from CPU to GPU is shown in Fig. 14.25, based on Fig. 14.22.

Each draw call may correspond to:

- one mesh
- one submesh
- one meshlet (in mesh-shader pipelines)
- or many meshes batched together

Although the Rendering Pipeline shown in Fig. 14.23 and Fig. 14.24 do not explicitly include per-frame animation flow—because the inputs are labeled **Vertex Specification** and **Vertex Data** and they do not show Animation Parameters as part of the rendering process—the pipeline is still applicable.

However the following table from OpenGL rendering pipeline Figure 1.2 and its stages from the book *OpenGL Programming Guide, 9th Edition*^{Page 692, 36} is broad enough to cover animation.

³⁵ https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

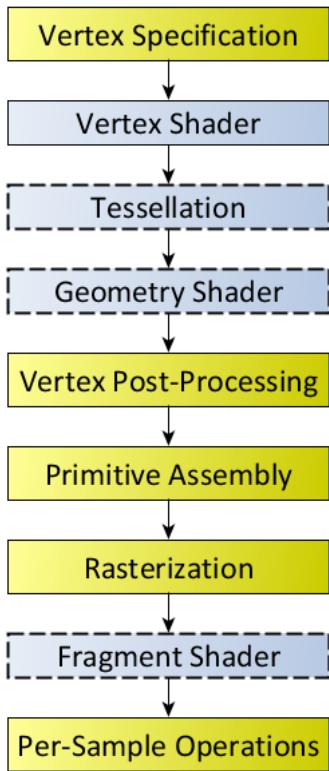


Fig. 14.23: Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages. Shaders with dashed outlines indicate optional shader stages.

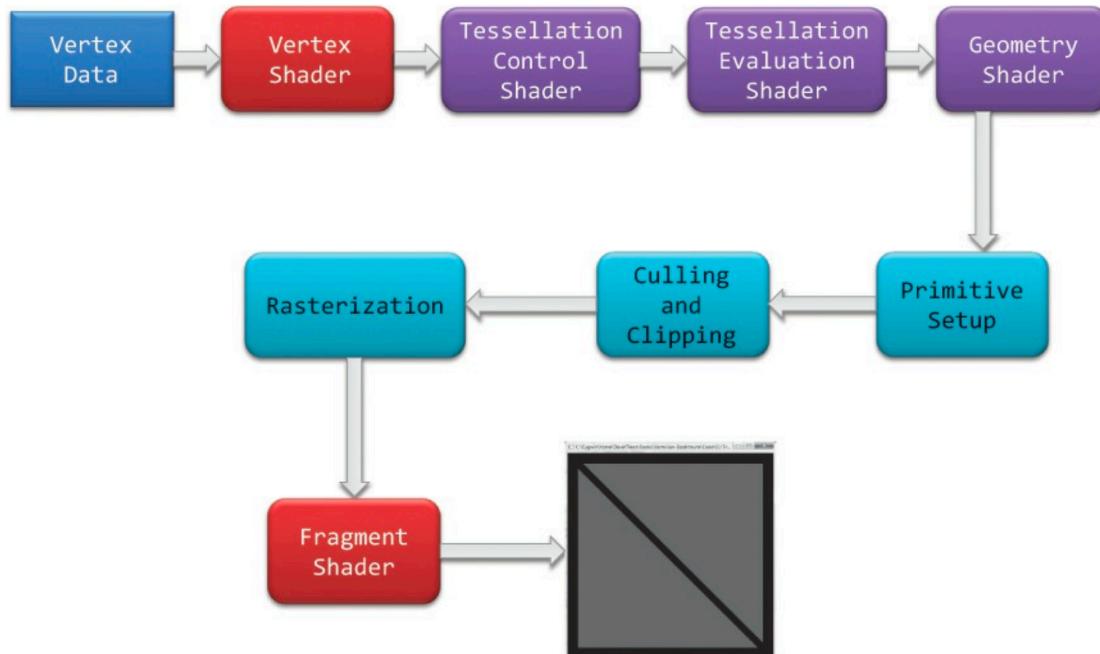


Figure 1.2 OpenGL pipeline

Fig. 14.24: OpenGL pipeline

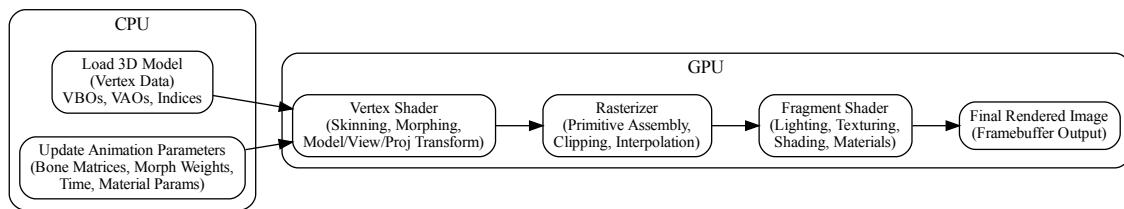


Fig. 14.25: CPU and GPU Pipeline For Shaders

Table 14.3: OpenGL rendering pipeline from page 10 and page 36 of book “OpenGL Programming Guide 9th Edition”^{[Page 692, 36](#)} and ^{[Page 699, 35](#)}.

Stage.	Description
Vertex Shading	Vertex → Vertex and other data such as color for later passes. For each vertex issued by a drawing command, a vertex shader processes the data associated with that vertex. Vertex Shader: provides the Vertex → Vertex transformation effects controlled by the users.
Tessellation Shading	Create more detail on demand when zoomed in. After the vertex shader processes each vertex, the tessellation shader stage (if active) continues processing. The tessellation stage is actually divided into two shaders known as the tessellation control shader and the tessellation evaluation shader . A single patch from Tessellation Control Shader (TCS) and Tessellation Evaluation Shader (TVS) can generate millions of micro-triangles . See reference below.
Geometry Shading	Vertex → Primitives. Allows additional processing of geometric primitives. This stage may create new primitives before rasterization. The Geometry shading stage is another optional stage that can modify entire geometric primitives within the OpenGL pipeline. This stage operates on individual geometric primitives allowing each to be modified. In this stage, you might generate more geometry from the input primitive, change the type of geometric primitive (e.g., converting triangles to lines), or discard the geometry altogether. If activated, a geometry shader receives its input either after vertex shading has completed processing the vertices of a geometric primitive or from the primitives generated from the tessellation shading stage, if it’s been enabled. Geometry Shader: provides the Vertex → Geometric Primitives transformation effects controlled by the users. See Chapter 10 of the Red Book ^{39} . Geometry Shader (GS) can generate both more vertices and more primitives than it receives.
Primitive Assembly	The previous shading stages all operate on vertices, with the information about how those vertices are organized into geometric primitives being carried along internal to OpenGL. The primitive assembly stage organizes the vertices into their associated geometric primitives in preparation for clipping and rasterization.
Clipping	Clipping hidden parts. Occasionally, vertices will be outside of the viewport—the region of the window where you’re permitted to draw—and cause the primitive associated with that vertex to be modified so none of its pixels are outside of the viewport. This operation is called clipping and is handled automatically by OpenGL.
Rasterization	Vertex → Fragment. or Geometric Primitives → Fragment. The job of the rasterizer is to determine which screen locations are covered by a particular piece of geometry (point, line, or triangle). Knowing those locations, along with the input vertex data, the rasterizer linearly interpolates the data values for each varying variable in the fragment shader and sends those values as inputs into your fragment shader. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.
Fragment Shading	Fragment → Fragment. Determine color for each pixel. In this stage, a fragment’s color and depth values are computed and then sent for further processing in the fragment-testing and blending parts of the pipeline. The final stage where you have programmable control over the color of a screen location is fragment shading. In this shader stage, you use a shader to determine the fragment’s final color (although the next stage, per-fragment operations, can modify the color one last time) and potentially its depth value. Fragment shaders are very powerful, as they often employ texture mapping to augment the colors provided by the vertex processing stages. A fragment shader may also terminate processing a fragment if it determines the fragment shouldn’t be drawn; this process is called fragment discard. A helpful way of thinking about the difference between shaders that deal with vertices and fragment shaders is this: vertex shading (including tessellation and geometry shading) determines where on the screen a primitive is, while fragment shading uses that information to determine what color that fragment will be.

Table 14.4: Continue OpenGL rendering pipeline from page 10 and page 36 of book “OpenGL Programming Guide 9th Edition”^{Page 692, 36} and^{Page 699, 35}.

Stage.	Description
Per-Fragment Operations	During this stage, a fragment’s visibility is determined using depth testing (also commonly known as z-buffering) and stencil testing. If a fragment successfully makes it through all of the enabled tests, it may be written directly to the framebuffer, updating the color (and possibly depth value) of its pixel, or if blending is enabled, the fragment’s color will be combined with the pixel’s current color to generate a new color that is written into the framebuffer .
Compute shading stage	Compute shader: may be applied in any stage. This is not part of the graphical pipeline like the stages above, but stands on its own as the only stage in a program. A compute shader processes generic work items, driven by an application-chosen range, rather than by graphical inputs like vertices and fragments. Compute shaders can process buffers created and consumed by other shader programs in your application. This includes framebuffer post-processing effects or really anything you want. Compute shaders are described in Chapter 12 of Red Book, “Compute Shaders” ³⁹ .

- Tessellation Shading: The core problem that Tessellation deals with is the static nature of 3D models in terms of their detail and polygon count. The thing is that when we look at a complex model such as a human face up close we prefer to use a highly detailed model that will bring out the tiny details (e.g. skin bumps, etc). A highly detailed model automatically translates to more triangles and more compute power required for processing. ... One possible way to solve this problem using the existing features of OpenGL is to generate the same model at multiple levels of detail (LOD). For example, highly detailed, average and low. We can then select the version to use based on the distance from the camera. This, however, will require more artist resources and often will not be flexible enough. ...Let’s take a look at how Tessellation has been implemented in the graphics pipeline. The core components that are responsible for Tessellation are two new shader stages and in between them a fixed function stage that can be configured to some degree but does not run a shader. The first shader stage is called Tessellation Control Shader (TCS), the fixed function stage is called the Primitive Generator (PG), and the second shader stage is called Tessellation Evaluation Shader (TES). Some GPU havn’t this fixed function stage implemented in HW and even havn’t provide these TCS, TES and Gemoetry Shader. User can write Compute Shaders instead for this on-fly detail display. This surface is usually defined by some polynomial formula and the idea is that moving a CP has an effect on the entire surface. ...The group of CPs is usually called a Patch³⁷. Chapter 9 of Red Book^{Page 692, 36} has details.

³⁹ Page 36 of book “OpenGL Programming Guide 9th Edition”^{Page 692, 36}.

³⁷ <https://ogldev.org/www/tutorial30/tutorial30.html>

Table 14.5: Data Flow Through the OpenGL Shader Pipeline

Shader Stage	Input Data (from CPU or previous stage)	Output Data (to next stage)	How GPU Hardware Uses These Data (with Stage Name)
Vertex Shader	<ul style="list-style-type: none"> • Per-vertex attributes: <ul style="list-style-type: none"> – Positions (vec3/vec4) – Normals, tangents – Texture coordinates – Vertex colors – Skinning weights/indices • Uniforms and UBOs • Textures / samplers 	<ul style="list-style-type: none"> • gl_Position (clip-space) • Varyings • Optional point size 	<ul style="list-style-type: none"> • Vertex Processing Stage: <ul style="list-style-type: none"> – ALUs transform vertices – Writes positions into Primitive Assembly – Stores varyings in interpolation registers
Tessellation Control Shader (TCS)	<ul style="list-style-type: none"> • Patch control points • Uniforms • Per-patch attributes 	<ul style="list-style-type: none"> • Modified control points • Tessellation levels 	<ul style="list-style-type: none"> • Tessellation Control Stage: <ul style="list-style-type: none"> – Writes tessellation levels to fixed-function tessellator – Stores control points in patch memory
Tessellation Evaluation Shader (TES)	<ul style="list-style-type: none"> • Tessellated coordinates (u,v,w) • Patch control points • Uniforms 	<ul style="list-style-type: none"> • gl_Position • Varyings 	<ul style="list-style-type: none"> • Tessellation Evaluation Stage: <ul style="list-style-type: none"> – ALUs compute final vertex positions – Outputs to Primitive Assembly – Sends varyings to interpolation hardware
Geometry Shader	<ul style="list-style-type: none"> • Assembled primitives • All varyings 	<ul style="list-style-type: none"> • Zero or more primitives • New varyings • New gl_Position 	<ul style="list-style-type: none"> • Geometry Processing Stage: <ul style="list-style-type: none"> – Allocates per-primitive scratch memory – Emits new primitives – Expands or reduces geometry
Rasterizer (Fixed Function)	<ul style="list-style-type: none"> • Primitives (triangles/lines/points) • Per-vertex varyings 	<ul style="list-style-type: none"> • Fragments • Interpolated varyings • gl_FragCoord 	<ul style="list-style-type: none"> • Rasterization Stage: <ul style="list-style-type: none"> – Barycentric units interpolate varyings – Generates fragments – Sends fragments to fragment shader cores

Table 14.6: Data Flow Through the OpenGL Shader Pipeline Continue

Fragment Shader	<ul style="list-style-type: none"> Interpolated varyings Textures / samplers Uniforms <code>gl_FragCoord</code> 	<ul style="list-style-type: none"> <code>gl_FragColor</code> or user-defined outputs Depth override (optional) 	• Fragment Processing Stage: <ul style="list-style-type: none"> ALUs compute pixel color Texture units fetch texels Outputs color/depth to ROP
Output Merger / ROP (Fixed Function)	<ul style="list-style-type: none"> Fragment shader outputs Depth/stencil values Blending state 	<ul style="list-style-type: none"> Final framebuffer color Updated depth/stencil buffers 	• Output Merger Stage: <ul style="list-style-type: none"> Performs depth/stencil tests Applies blending Writes final pixels to framebuffer memory Handles MSAA resolve

A varying is a piece of data that:

- Comes out of the vertex shader
- Gets interpolated by the rasterizer
- Arrives as input to the fragment shader

It is called **varying** because its value **varies across the surface of a triangle**.

Table 14.7: Examples of Common Varyings

Varying Name	Meaning	Why It Varies Across the Primitive
<code>vNormal</code>	Surface normal at each vertex	Lighting requires a smoothly changing normal across the triangle so per-pixel shading can compute correct diffuse and specular terms
<code>vUV</code>	Texture coordinates	Each pixel needs its own UV to sample the correct texel from the texture
<code>vColor</code>	Vertex color (per-vertex material tint)	Enables smooth color gradients or per-vertex painting effects
<code>vWorldPos</code>	World-space position of the vertex	Used for per-pixel lighting, reflections, shadows, and screen-space effects; must be interpolated so each fragment knows its own world position

For 2D animation, the model is created by 2D only (1 face only), so it only can be viewed from the same face of model. If you want to display different faces of model, multiple 2D models need to be created and switch these 2D models from face(flame) to face(flame) from time to time³⁸.

Mobile GPU 3D Rendering

The traditional desktop GPUs is **IMR —Immediate-Mode Rendering**: Cache misses dominate bandwidth.

TBDR —Tile-Based Deferred Rendering: Cache misses are nearly eliminated.

³⁸ <https://tw.video.search.yahoo.com/search/video?fr=yfp-search-sb&p=2d+animation#id=12&vid=46be09edf57b960ae79e9cd077ee1ea&action=view>

Note

Idea:

1. TBDR divides the whole frame into small tiles that fit entirely into on-chip **SRAM**.
2. Remove stages of Tessellation Control Shader (TCS), Tessellation Evaluation Shader (TES) and Geometry Shader (GS) since they are optional stages are shown in Fig. 14.24. Instead, developers use **compute shaders** before the graphics pipeline to generate meshlets, perform LOD selection, or add extra geometric detail for close-up **room-in** effects is shown as Fig. 14.28.

★ TBDR reduces **cache-miss** rate by roughly **10x–50x** compared to IMR, because all intermediate color/depth/stencil traffic stays in on-chip tile memory instead of going to L2/DRAM.

★ Desktop GPUs adopt IMR partly because GS/Tess/Mesh Shaders cannot run efficiently on TBDR. In addition, desktop GPUs adopt IMR because they have the **power**, **bandwidth**, and architectural freedom to **support unpredictable geometry pipelines** and massive workloads that would break TBDR's tile-based constraints.

TBDR — Tile-Based Deferred Rendering

⚠ For **low power** mobile device, mobile GPUs use **tile-based** rendering to **reduce the traffic to DRAM** as described below:

The traditional desktop GPUs is IMR — Immediate-Mode Rendering:

1. IMR: “Draw call arrives → render immediately”

CPU issues DrawCall #1

- GPU transforms vertices
- GPU rasterizes fragments
- GPU writes to DRAM

It never waits to see the rest of the frame.

2. TBDR: “Draw call arrives → store geometry, don't render yet”. TBDR processes it into two phases as follows:

Phase 1 — Binning (Full-Frame Geometry Processing) is shown as Fig. 14.26.

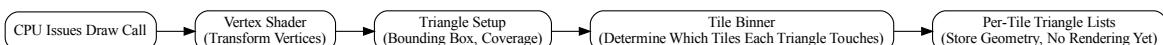


Fig. 14.26: TBDR Pipeline

When a draw call arrives on a TBDR GPU:

- It runs the vertex shader
- It transforms all triangles
- It determines which tiles each triangle touches
- It stores triangle references in **per-tile lists** is shown as follows:

Example of per-tile lists

```
Tile 0 → triangles: [T1, T7, T8, T20]
Tile 1 → triangles: [T2, T3, T7]
Tile 2 → triangles: [T4, T5, T6, T9, T10]
...
```

Phase 2 — Tile Rendering (Deferred Shading)

For each tile:

- Load tile's triangle list
- Rasterize only those triangles
- Shade only visible fragments
- Keep all intermediate buffers in on-chip **SRAM**
- Write final tile to DRAM once

★ As you can see, tile is a small part of rendering frame. In Phase 2 — Tile Rendering, GPU rendering each tile and keep the rendering result of each **tile in SRAM**.

TBDR Rendering

✓ Rendering flow:

- Vertex Shader → Primitive Setup → **Tile-Based Culling and Clipping** → Rasterization → Fragment Shader

TBDR architectures depend on:

- predictable geometry counts,
- small on-chip tile memory,
- minimal external memory traffic.

⚠ As described in section [3D Rendering Pipeline](#), Geometry Shader (GS) can generate both more vertices and more primitives than it receives. A single patch from Tessellation Control Shader (TCS) and Tessellation Evaluation Shader (TVS) can generate millions of micro-triangles. GS and Tessellation introduce **unbounded geometry amplification**, which breaks these assumptions and forces expensive DRAM spills for TBDR as shown in [Fig. 14.27](#),

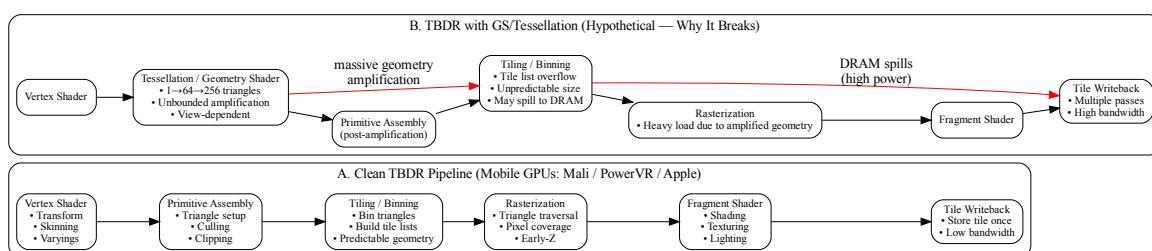


Fig. 14.27: CPU and GPU Pipeline For Shaders in Mobile Device

This is why Mali, PowerVR, Apple, and Adreno mobile GPUs all omit these stages⁴⁰⁴¹.

⁴⁰ ARM Developer: *Why mobile GPUs avoid geometry shaders* <https://developer.arm.com/documentation/102476/latest/>

⁴¹ Imagination Technologies: *Why Geometry Shaders Are Not Supported* <https://www.imgtec.com/blog/why-powervr-does-not-support-geometry-shaders/>

Developers manually invoke **compute shaders** to generate meshlets or additional geometry, adding extra geometric detail for close-up **zoom-in** effects. Both Mali and PowerVR GPUs then run the standard vertex shader on the generated results.

✓ Step 1 —Developer dispatches a compute shader

This compute shader can do things like:

- break a big mesh into meshlets as Fig. 14.28. The mesh and meshlets are described in the *Mesh-Shader Pipeline* next section.
- generate more vertices for detail (subdivision, displacement)
- perform LOD selection
- cull invisible meshlets
- generate new index/vertex buffers

This is developer-controlled, not automatic.

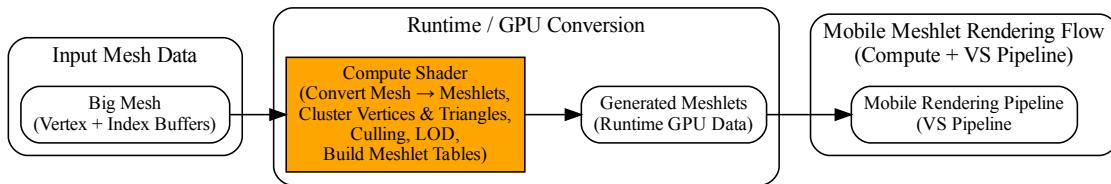


Fig. 14.28: CPU and GPU Pipeline For Shader's in Mobile Device

The compute shader writes results into:

- SSBOs
- vertex buffers
- index buffers

These buffers now contain the final geometry you want to render.

✓ Step 2 —Developer issues a normal draw call

The Mali and PowerVR's rendering flow is illustrated as Fig. 14.29.

- Modern mobile engines instead use **compute shaders** for culling, LOD, meshlet prep, and procedural geometry.

✓ Geometry Shaders are notoriously inefficient even on desktop GPUs. GPU vendors (NVIDIA + AMD + Intel) designed the mesh-shader pipeline described in the section *Mesh-Shader Pipeline*.

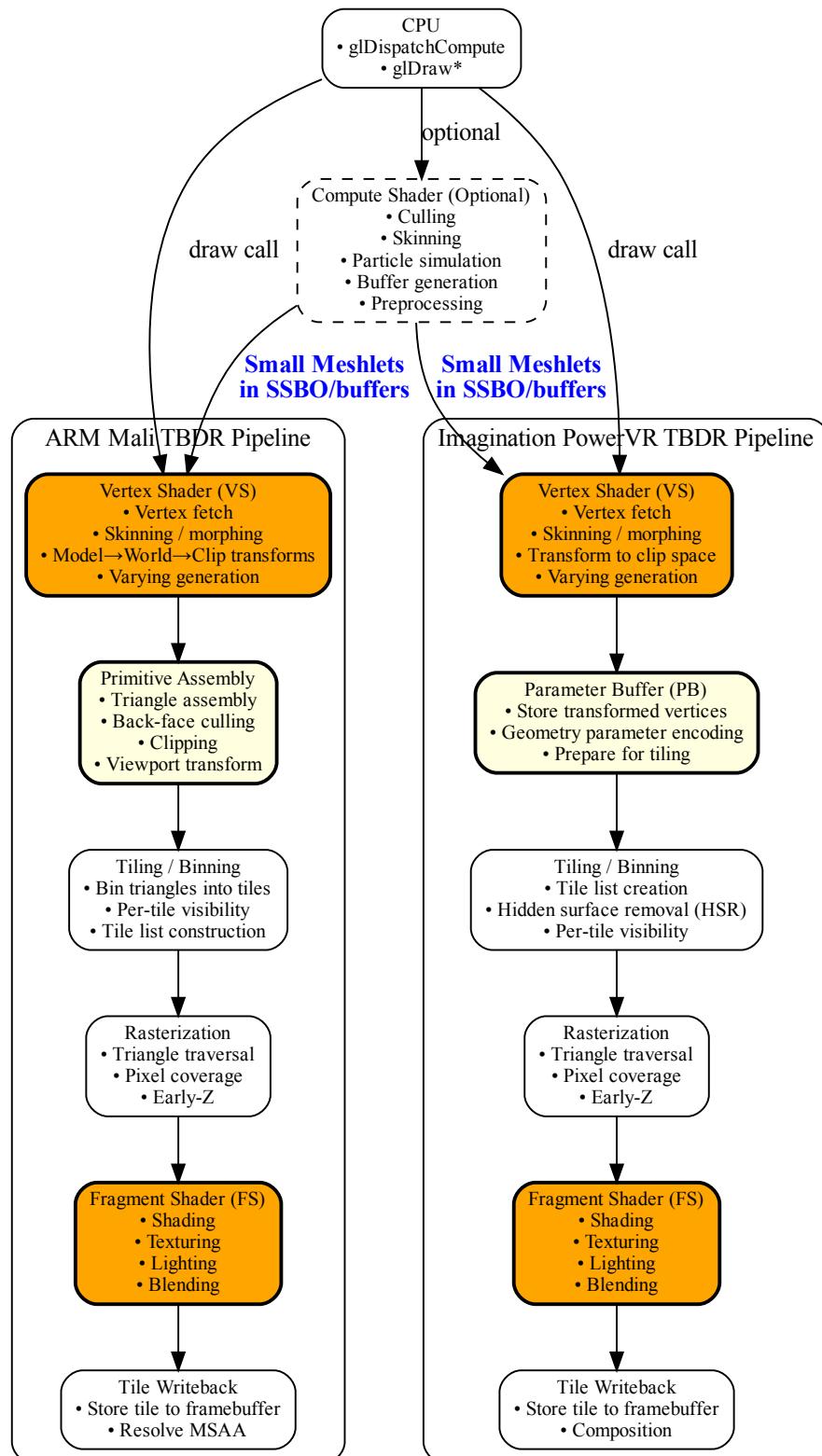
Mesh-Shader Pipeline

A single 3D object can contain 1 mesh, multiple meshes or hundreds of meshes (complex characters, vehicles, weapons).

Reasons

The purpose of converting **a mesh into small clusters (meshlets)** is to give the GPU **small, coherent**, cullable, cache-friendly work units that dramatically improve parallelism, memory locality, and LOD efficiency.

Raw meshes can have anywhere from thousands to millions of vertices/triangles, while meshlets intentionally restrict clusters to ~32–128 vertices and ~32–256 triangles to maximize GPU efficiency.



Motivation

NVIDIA, AMD, and Intel all needed:

- a compute-like geometry pipeline
- meshlet-based processing
- better culling
- GPU-driven rendering
- a **replacement** for VS → TCS → TES → GS

So the vendors co-designed the hardware pipeline.

✓ Microsoft and Khronos (Vulkan) each standardized it in their own APIs

✓ **Solution:** As shown in Fig. 14.30.

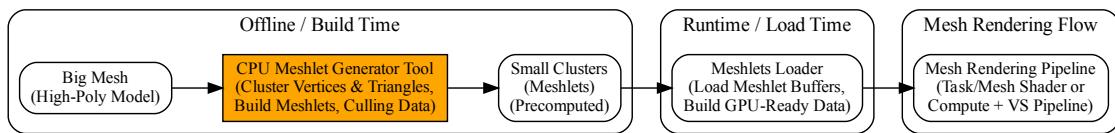


Fig. 14.30: Meshlet Offline To Render

Rendering Pipeline

✓ GPU vendors (NVIDIA + AMD + Intel) designed the mesh-shader pipeline:

3D Modeling Tool Output (big mesh)

→ CPU Meshlet Generator Tool (offline)

- Converting **big mesh** into **small clusters (meshlets)** to maximize GPU efficiency.

→ Precomputed meshlets (static clusters)

→ Task Shader (optional)

→ Mesh Shader

The animation flow from CPU to GPU for **Traditional**, **Compute Shader** based and **Mesh Shader** are shown in Fig. 14.31, Fig. 14.32 and Fig. 14.33.

Mesh shading (Vulkan VK_EXT_mesh_shader, similarly in NV mesh shader) replaces the fixed vertex-input + VS + optional tess/GS stages with a compute-like geometry pipeline:

As in Fig. 14.33, NVIDIA/AMD desktop provide mesh-shader to do the following pipeline.

Task Shader Responsibilities

The **Task Shader** acts as a coarse-grained work distributor.

Key responsibilities:

- Perform coarse culling at the meshlet or instance level.
- Select appropriate LODs for distant geometry.
- Build a compact list of meshlets to be processed.

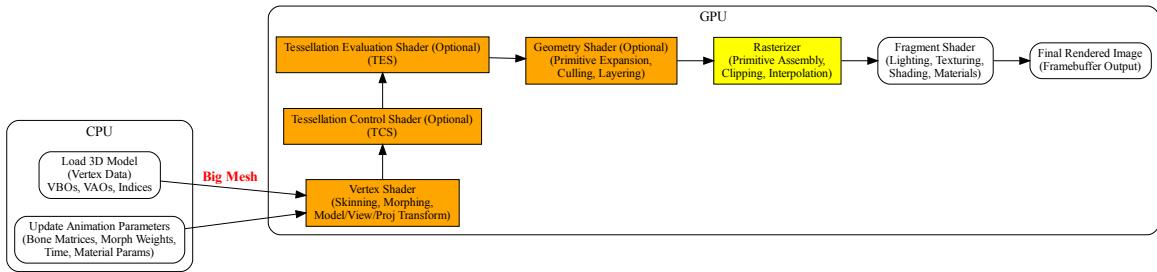


Fig. 14.31: CPU and GPU Traditional Pipeline For Shaders

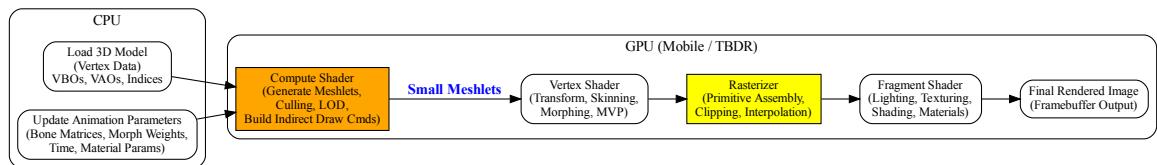


Fig. 14.32: CPU and GPU Mobile Pipeline For Shaders

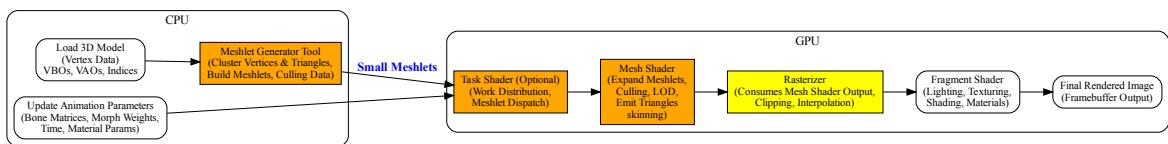


Fig. 14.33: CPU and GPU Mesh Shader Pipeline For Shaders

- Determine how many mesh shader workgroups to launch.
- Pass a payload (task data) to mesh shader workgroups.

The task shader does *not* emit vertices or primitives.

Mesh Shader Responsibilities

The **mesh shader** replaces the vertex shader, tessellation, and often the geometry shader. It operates on meshlets inside workgroups.

Key responsibilities:

- Load meshlet vertices and indices from GPU memory.
- Apply transforms, skinning, morphing, and procedural deformation.
- Perform fine-grained culling: - frustum culling - backface culling - small triangle culling - cluster-level culling
- Generate the final set of vertices and primitives.
- Emit primitives directly to the rasterizer.

Because mesh shaders run in workgroups, they can use shared memory and synchronize threads, enabling efficient reuse of vertex data.

Why Meshlets Fit GPU Architecture Well

Meshlets align naturally with GPU hardware for several reasons:

- **Workgroup-Friendly:** Each meshlet maps cleanly to a single workgroup, keeping memory usage predictable and minimizing divergence.
- **Cache Efficiency:** Meshlets maximize vertex reuse and reduce memory bandwidth by grouping spatially local geometry.
- **Hierarchical Culling:** - Task shader: coarse culling of entire meshlets. - Mesh shader: fine culling of individual primitives.
- **Reduced CPU Overhead:** The GPU can perform culling, LOD selection, and primitive generation without CPU intervention, enabling GPU-driven rendering.
- **Scalable Parallelism:** Each meshlet is processed independently, allowing thousands of workgroups to run in parallel across GPU SMs.

Both Mobile GPU and Mesh-Shader GPU convert big mesh to small meshlets and render them efficiently using GPU SIMT execution and memory hierarchy. The comparison is shown in the following table.

Comparison: Mobile GPU (Compute-Shader Based) vs Desktop Mesh-Shader GPU

The **Mesh Shader** is similar to the previous section of **Mobile Compute Shader** based Meshlets as the following table:

Table 14.8: Mobile GPU vs Desktop Mesh-Shader GPU —Concept Comparison

Concept	Mobile GPU (Compute-Shader Based)	Desktop Mesh-Shader GPU
Meshlet generation	Compute Shader generates meshlets at runtime	CPU Meshlet Generator Tool (offline)
Tile-based	Yes	No
Work distribution	Compute Shader dispatch groups handle distribution	Task Shader distributes meshlet workloads
Meshlet expansion	Vertex Shader processes vertices after compute pre-processing	Mesh Shader expands meshlets and emits triangles
Culling & LOD	Compute Shader performs culling and LOD before raster	Task + Mesh Shader perform culling and LOD selection
Draw submission	Compute Shader writes indirect draw commands	Mesh Shader emits primitives directly to rasterizer
Pipeline family	Traditional Pipeline (VS → Raster → FS)	Mesh-Shader Pipeline (Task → Mesh → Raster → FS)

Summary

Meshlets and the mesh-shader pipeline transform geometry processing into a compute-like workflow. By organizing geometry into small, cache-friendly clusters and distributing work across task and mesh shaders, modern GPUs achieve higher throughput, better culling efficiency, and reduced CPU overhead compared to the traditional vertex-processing pipeline.

3D modeling tools do NOT generate meshlets. Meshlets are always generated later, using specialized meshlet-generation tools, most commonly:

- NVIDIA meshlet generator (NV_mesh_shader ecosystem)
- meshoptimizer (Khronos-recommended, open source)
- Engine-specific meshlet builders (Unreal, Frostbite, etc.)

So the meshlet conversion happens after the model is exported —not inside Blender, Maya, 3ds Max, etc.

Animation Example

Listing 14.1: Example GPU skinning

```
vec4 skinnedPos = vec4(0.0);
for (int i = 0; i < 4; ++i) {
    skinnedPos += boneMatrices[boneIndex[i]] * vec4(position, 1.0) * boneWeight[i];
}
gl_Position = projection * view * model * skinnedPos;
```

Here:

- **position, boneIndex, boneWeight = vertex attributes**
- **boneMatrices, model, view, projection = uniforms**

The vertex shader combines them.

✓ Why boneIndex[] and boneWeight[] are 3D Model Information

These two arrays describe how the mesh is bound to the skeleton.

They are part of the static mesh data, created during rigging in Blender/Maya/etc.

`boneIndex[]` → tells which bone

- For each vertex: which bones influence it
- Example: { 3, 7, 12, 0 }

`boneWeight[]` → tells how much

- For each vertex: how much each bone influences it
- Example: { 0.5, 0.3, 0.2, 0.0 }

These values never change during animation. They are baked into the mesh and stored in the VBO as vertex attributes.

✓ Why `boneMatrices[]` is Animation Parameters

`boneMatrices[]` → tells where the bone is this frame

- Example: `boneMatrices[3]` (upper arm bone this frame)

[0.87 -0.49 0.00 0.12] [0.49 0.87 0.00 0.03] [0.00 0.00 1.00 0.00] [0.00 0.00 0.00 1.00]

This matrix might represent:

- a 30° rotation of the upper arm
- plus a small translation (0.12, 0.03, 0.0)

Animation Parameters are dynamic per-frame data, such as:

- bone matrices
- animation time
- morph weights
- blend factors
- procedural animation inputs

These change every frame.

✓ OpenGL API Commands That Trigger GPU Skinning

Overview

In OpenGL, animation is not built into the API. Instead, animation occurs because the application updates *Animation Parameters* (such as bone matrices) and the *vertex shader* interprets them. The GPU performs the animation math during the draw call.

The following sections describe the exact OpenGL commands involved in triggering GPU-based vertex animation.

1. Updating Animation Parameters (Uniforms or UBOs)

Animation Parameters such as `boneMatrices[]` are updated every frame. They are supplied to the vertex shader as uniforms or through a uniform buffer object (UBO).

Uniform array example:

```
GLint loc = glGetUniformLocation(program, "boneMatrices");
glUniformMatrix4fv(loc, boneCount, GL_FALSE, boneMatrixData);
```

Uniform Buffer Object example:

```
glBindBuffer(GL_UNIFORM_BUFFER, boneUBO);
glBufferSubData(GL_UNIFORM_BUFFER, 0, size, boneMatrixData);
glBindBufferBase(GL_UNIFORM_BUFFER, bindingPoint, boneUBO);
```

These commands send the per-frame animation data to the GPU.

2. Binding Vertex Data (Mesh Information)

Static mesh data such as positions, normals, `boneIndex []` and `boneWeight []` is stored in vertex buffer objects (VBOs) and attached to a vertex array object (VAO).

```
glBindVertexArray(vao);

glBindBuffer(GL_ARRAY_BUFFER, vboPositions);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, stride, offset);
 glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, vboBoneIndex);
glVertexAttribIPointer(1, 4, GL_UNSIGNED_BYTE, stride, offset);
 glEnableVertexAttribArray(1);

glBindBuffer(GL_ARRAY_BUFFER, vboBoneWeight);
glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, stride, offset);
 glEnableVertexAttribArray(2);
```

These commands provide the static 3D model information to the vertex shader.

3. Activating the Shader Program

The vertex shader containing the skinning logic must be activated before drawing.

```
glUseProgram(program);
```

This step ensures that the GPU will execute the correct vertex shader when the draw call is issued.

4. Issuing the Draw Call (Animation Trigger)

The draw call is the moment when the GPU executes the vertex shader for each vertex. This is where animation actually happens.

```
glDrawElements(GL_TRIANGLES, indexCount, GL_UNSIGNED_INT, 0);
```

or:

```
glDrawArrays(GL_TRIANGLES, 0, vertexCount);
```

The vertex shader runs once per vertex, combining:

- vertex attributes (`position`, `boneIndex []`, `boneWeight []`)
- animation parameters (`boneMatrices []`)

to compute the animated vertex position.

Summary Table

Purpose	Data Type	OpenGL API	Static or Dynamic
Mesh data (positions, bone indices, bone weights)	Vertex Attributes	glVertexAttribPointer glEnableVertexAttribArray	Static (stored in VBO)
Animation Parameters (bone matrices)	Uniforms / UBO	glUniformMatrix4fv glBufferSubData	Dynamic (updated every frame)
Activate shader program	Shader Program	glUseProgram	Per draw
Trigger animation	Draw Call	glDrawElements / glDrawArrays	Per frame

Conclusion

OpenGL does not provide a built-in animation system. Instead, animation occurs because the application updates Animation Parameters each frame and the vertex shader applies animation math during the draw call. The GPU performs the animation only when the draw command is issued.

14.2.3 GLSL (GL Shader Language)

OpenGL is a standard specification for designing 2D and 3D graphics and animation in computer graphics. To support advanced animation and rendering, OpenGL provides a large set of APIs (functions) for graphics processing. Popular 3D modeling and animation tools—such as Maya, Blender, and others—can utilize these APIs to handle 3D-to-2D projection and rendering directly on the computer.

The hardware-specific implementation of these APIs is provided by GPU manufacturers, ensuring that rendering is optimized for the underlying hardware.

Background

In the previous section *SW Stack and Data Flow* described **how each frame is generated** to display the **movement animation or skinning effects** using the small animation parameters stored in 3D model and sent from CPU.

Based on description of section *SW Stack and Data Flow*, we know the animation can be implemented using **fixed-function skinning**. The following are the animation examples for **shader-less era**.

- ✓ Some consoles and mobile GPUs did have fixed-function skinning.
- ✓ In those systems, you could upload bone matrices and let hardware animate vertices.
- ✗ **But you could not change the formulas —only use the built-in ones.**

The following console GPUs did have fixed-function skinning:

PlayStation 2 (PS2) —VU0/VU1 Microcode

PS2 had fixed hardware instructions for:

- skinning
- morphing
- matrix blending

Developers could upload bone matrices and let the hardware do the blending. No shaders existed yet.

Nintendo GameCube / Wii —XF Unit

The GameCube GPU had a fixed-function transform unit that supported:

- matrix palette skinning (up to 10 matrices)

- per-vertex weighted blending

Again, no shaders —but hardware skinning existed.

The previous section *Role and Purpose of Shaders* also explained different visual effects can be achieved by **switching shaders** to shappingly different materials across frames.

✗ However the **fixed-function pipeline (OpenGL 1.x / early 2.x without shaders)** has:

- no per-vertex programmable math
- no access to bone matrices
- no ability to blend multiple positions
- no ability to apply time-based deformation
- no ability to read custom vertex attributes
- no ability to modify vertex positions except via the model-view matrix

✗ As result the shader-less (fixed-function) pipeline in early OpenGL did not support GPU-based skinning. Skinning had to be implemented on the CPU, which imposed limitations on both **animation capability and performance**, as described below:

Major Disadvantages of a Shader-less (Fixed-Function) Pipeline

- **No GPU-side animation**
 - Cannot perform skinning, morphing, or procedural deformation on the GPU.
 - All animation must be computed on the CPU, causing performance bottlenecks.
- **Limited lighting and materials**
 - Only fixed-function lighting is available.
 - No custom BRDFs, PBR workflows, toon shading, or stylized effects.
- **No procedural or time-based effects**
 - Cannot implement UV animation, distortion, dissolve, holograms, or particle effects.
 - No access to noise functions or time-driven logic in the pipeline.
- **No post-processing**
 - Motion blur, bloom, depth of field, color grading, and screen-space effects are impossible.
- **Rigid data flow**
 - Cannot define custom vertex attributes, varyings, or uniform buffers.
 - Material and animation systems cannot be data-driven.
- **Poor scalability and performance**
 - CPU must update all animated geometry every frame.
 - GPU parallelism is unused, limiting scene complexity.
- **Deprecated and non-portable**
 - Fixed-function pipeline is removed in modern OpenGL core profiles.
 - Not compatible with contemporary engines or hardware.

✓ All modern consoles (PS5, PS5 Pro, PS6-class hardware of Sony, Switch 2 of Nintendo) use **programmable shader architectures** rather than fixed-function animation hardware. Fixed-function animation is now **obsolete**.

- Sony's current and upcoming GPUs are based on AMD RDNA architectures, which are fully programmable shader GPUs.
- Nintendo's upcoming Switch 2 uses a custom Nvidia Ampere GPU.

Examples

An OpenGL program typically follows a structure like the example below:

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's
    →constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red
    →color
}
```

Fragment shader

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader (same name and same
    →type)

void main()
{
    FragColor = computeColorOfThisPixel(...);
```

OpenGL user program

```
int main(int argc, char ** argv)
{
    // init window, detect user input and do corresponding animation by calling opengl
    →api
    ...
}
```

The last `main()` function in an OpenGL application is written by the user, as expected. Now, let's explain the purpose of the first two main components of the OpenGL pipeline.

As discussed in the *Concepts of Computer Graphics* textbook, OpenGL provides a rich set of APIs that allow programmers to render 3D objects onto a 2D computer screen. The general rendering process follows these steps:

1. The user sets up lighting, textures, and object materials.
2. The system calculates the position of each vertex in 3D space.

3. The GPU and rendering pipeline automatically determine the color of each pixel based on lighting, textures, and interpolation.
4. The final image is displayed on the screen by writing pixel colors to the framebuffer.

To give programmers the flexibility to add custom effects or visual enhancements—such as modifying vertex positions for animation or applying unique coloring—OpenGL provides two programmable stages in the graphics pipeline:

- **Vertex Shader:** Allows the user to customize how vertex coordinates are transformed and processed.
- **Fragment Shader:** Allows the user to define how each pixel (fragment) is shaded and colored, enabling effects like lighting, textures, and transparency.

These shaders are written by the user and compiled at runtime, providing powerful control over the rendering process.

OpenGL uses fragment shader instead of pixel is : “Fragment shaders are a more accurate name for the same functionality as Pixel shaders. They aren’t pixels yet, since the output still has to pass several tests (depth, alpha, stencil) as well as the fact that one may be using antialiasing, which renders one-fragment-to-one-pixel non-true⁵⁴. Programmer is allowed to add their converting functions that compiler translates them into GPU instructions running on GPU processor. With these two shaders, new features have been added to allow for increased flexibility in the rendering pipeline at the vertex and fragment level⁵⁵. Unlike the shaders example here⁵⁶, some converting functions for coordinate in vertex shader or for color in fragment shade are more complicated according to the scenes of animation. Here is an example⁵⁷. In wiki shading page [Page 665, 3](#), Gouraud and Phong shading methods make the surface of object more smooth by glsl. Example glsl code of Gouraud and Phong shading on OpenGL api are here⁵⁸. Since the hardware of graphic card and software graphic driver can be replaced, the compiler is run on-line meaning driver will compile the shaders program when it is run at first time and kept in cache after compilation⁵⁹.

The shaders program is C-like syntax and can be compiled in few mini-seconds, add up this few mini-seconds of on-line compilation time in running OpenGL program is a good choice for dealing the cases of driver software or gpu hardware replacement⁶⁰.

Goals

Goals of GLSL Shader Language:

GLSL was designed for real-time graphics using programmable GPUs.

1. Programmable Pipeline:
 - Custom control over vertex, fragment, and other pipeline stages
 - Enables dynamic effects, lighting, animation, and transformations
2. GPU Acceleration
 - Executes on GPU cores for massive parallel performance
 - Optimized for matrix and vector operations common in graphics
3. Cross-Platform Compatibility:
 - Runs consistently across OSes and hardware via OpenGL

⁵⁴ <https://community.khronos.org/t/pixel-vs-fragment-shader/52838>

⁵⁵ https://en.m.wikipedia.org/wiki/OpenGL_Shading_Language

⁵⁶ <https://learnopengl.com/Getting-started/Shaders>

⁵⁷ <https://www.youtube.com/watch?v=LySSoYyfVU> at 5:25 from beginning: combine different textures.

⁵⁸ <https://github.com/rangle/Gouraud-Shading-and-Phong-Shading>

⁵⁹ Compiler and interpreter: (<https://www.guru99.com/difference-compiler-vs-interpreter.html>). AOT compiler: compiles before running; JIT compiler: compiles while running; interpreter: runs (reference <https://softwareengineering.stackexchange.com/questions/246094/understanding-the-differences-traditional-interpreter-jit-compiler-jit-interp>). Both online and offline compiler are AOT compiler. User call OpenGL api to run their program and the driver call online compiler to compile user’s shaders without user compiling their shader before running their program. When user run a CPU program of C language, he must compile C program before running the program. This is offline compiler.

⁶⁰ <https://community.khronos.org/t/offline-glsl-compilation/61784>

- Avoids vendor lock-in for portable shader code
4. C-Like Syntax
 - Familiar syntax for developers used to C-style languages
 - Supports functions, loops, conditionals, and custom types
 5. Fine-Grained Rendering Control
 - Direct access to geometry, color, texture, lighting parameters
 - Enables advanced effects like shadows, fog, reflections
 6. Real-Time Interactivity
 - Responds to user input, time, and animations at runtime
 - Suitable for games, simulations, and creative tools
 7. Minimal Host Dependency
 - Executes within the graphics driver context
 - No need for external libraries, file I/O, or system calls

GLSL vs. C: Feature Overview

GLSL expands upon C for GPU-based graphics programming.

Additions to C:

1. Specialized Data Types
 - `vec2`, `vec3`, `vec4`: float vectors
 - `mat2`, `mat3`, `mat4`: float matrices
 - `bvec`, `ivec`, `uvec`, `dvec`: boolean and integer vectors
 - `sampler2D`, `samplerCube`: texture samplers
2. Pipeline Qualifiers
 - `attribute`, `varying` (legacy)
 - `in`, `out`, `inout`: stage and parameter I/O
 - **uniform**: uniform variables are set externally by the host application (e.g., OpenGL) and remain constant across all shader invocations for a draw call.
 - `layout(location = x)`: set GPU variable locations
 - precision qualifiers: `lowp`, `mediump`, `highp`
3. Built-in Functions
 - `texture()`, `reflect()`, `refract()`, `normalize()`
 - `mix()`, `smoothstep()`: interpolation and blending
 - `dot()`, `cross()`, `transpose()`, `inverse()`: math ops
 - `dFdx()`, `dFdy()`, `fwidth()`: pixel derivatives
4. Swizzling
 - `.xyzw`, `.rgba`, `.stpq` access vector components
 - e.g., `vec4 pos = vec3(1, 2, 3).xyzx`

5. Shader-Specific Keywords

- `discard`: drop fragments early
- `gl_Position`, `gl_FragColor`, `gl_VertexID`: built-ins
- `subroutine`, `patch`, `sample`: advanced pipeline control

Removals and Restrictions:

1. No Pointers or Memory Access
 - No `*` or `&` operators
 - No `malloc`, `free`
2. No File I/O or Standard C Libs
 - No `stdio.h`, `printf()`, `fopen()`
3. No Recursion
 - Recursive functions not allowed
4. No `#include` Support
 - Files can't be included via preprocessor
5. Limited Control Flow
 - `goto` not allowed
 - Loops must be statically determinable in many cases for compiler optimization as follows:

Example for loops must be statically determinable in many cases

```
const int MAX_LIGHTS = 10;
for (int i = 0; i < MAX_LIGHTS; ++i) {
    // Safe: MAX_LIGHTS is a compile-time constant
}
```

6. Restricted C Keywords

- `typedef`, `union`, `enum`, `class`, `namespace`, `inline`, etc.
- Reserved or disallowed

Notes:

- Changes help GPU execute safely in parallel
- Designed for real-time, interactive graphics

GLSL Qualifiers by Shader Stage

The CPU and GPU Pipeline For Shaders is introduced in section [3D Rendering Pipeline](#). The Fig. 14.34 is the summary of GLSL Qualifiers below.

Vertex Shader:

- `in`: Receives per-vertex attributes from buffer objects, it is **3D Model Information** described in Fig. 14.34.
- `out`: Passes data to next stage (e.g., fragment shader)
- `uniform`: Global parameters like matrices or lighting, it is **Animation Parameters**, also referred to as **Uniform Updates** described in Fig. 14.34.

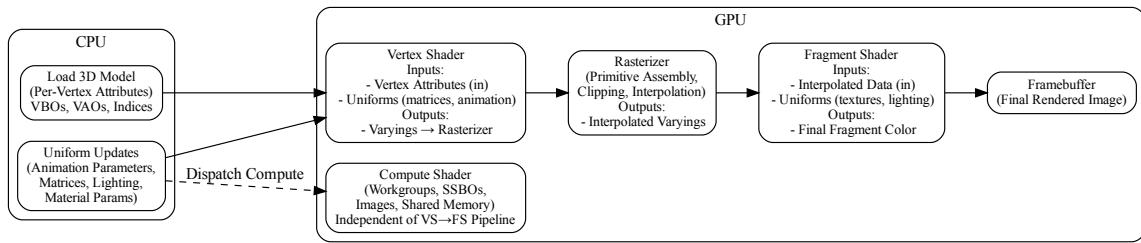


Fig. 14.34: Shaders input and output

- `layout(location = x)`: Binds input/output to attribute index
- `const`: Compile-time constants
- Cannot use interpolation qualifiers on inputs

Fragment Shader:

- `in`: Receives **interpolated data** from previous stage as shown in Fig. 14.34.
- `out`: Writes **Final Fragment Color** to FrameBuffer
- `uniform`: Global parameters like **textures** or **lighting** as shown in Fig. 14.34. **Uniform data remains unchanged** across all pipeline stages and is shared by all shaders in the pipeline. This means that uniform data represents global parameters for 3D GPU rendering.
- `flat`: Disables interpolation; uses provoking vertex
- `smooth`: Enables perspective-correct interpolation (default)
- `noperspective`: Linear interpolation in screen space
- `centroid`: Samples within primitive area (for multisampling)
- `sample`: Per-sample interpolation (GLSL 4.0+)
- `discard`: Terminates fragment processing early

Compute Shader:

- `layout(local_size_x = x)`: Defines workgroup size
- `uniform`: Input parameters from host
- `buffer`: Shader storage buffer access
- `shared`: Shared memory among invocations in a workgroup
- `image2D, image3D`: Direct image access
- `coherent, volatile, restrict`: Memory access control
- `readonly, writeonly`: Access mode for image/buffer
- `Compute shader`: **may be applied in any stage** as described in section *3D Rendering Pipeline*.

Common Across Stages:

- `const`: Immutable values
- `uniform`: Host-set global parameters

- layout(binding = x): Bind uniform/buffer/image to index
- precise: Ensures consistent computation
- invariant: Prevents variation across shader executions

Notes:

- attribute and varying are deprecated (use in/out instead)
- Interpolation qualifiers only affect fragment shader inputs
- Uniforms are shared across all stages and remain constant

Examples of GLSL Qualifiers by Shader Stage

```
// =====
// Vertex Shader: Qualifier Summary (GLSL)
// =====

// Vertex inputs
layout(location = 0) in vec3 aPosition;      // in: per-vertex attribute
layout(location = 1) in vec3 aNormal;

// Outputs to fragment shader
out vec3 vNormal;                           // out: passes to next stage

// Uniforms
uniform mat4 uModelMatrix;                  // uniform: global parameter
uniform mat4 uViewProjectionMatrix;

// Constants
const float PI = 3.14159265;                // const: compile-time constant

void main() {
    vNormal = aNormal;
    gl_Position = uViewProjectionMatrix * uModelMatrix * vec4(aPosition, 1.0);
}

// =====
// Fragment Shader: Qualifier Summary (GLSL)
// =====

// Inputs from vertex shader
in vec3 vNormal;                            // in: interpolated input

// Output to framebuffer
out vec4 fragColor;                         // out: final pixel color

// Uniforms
uniform vec3 uLightDirection;               // uniform: shared global input
uniform vec3 uBaseColor;

// Interpolation control
// flat in vec3 vFlatColor;                  // flat: no interpolation
// smooth in vec3 vSmoothColor;              // smooth: default interpolation
```

(continues on next page)

(continued from previous page)

```

// noperspective in vec3 vLinearColor; // noperspective: screen-space linear

void main() {
    float brightness = max(dot(normalize(vNormal), uLightDirection), 0.0);
    fragColor = vec4(uBaseColor * brightness, 1.0);
}

// =====
// Compute Shader: Qualifier Summary (GLSL)
// =====

#version 430

// Workgroup size
layout(local_size_x = 16, local_size_y = 16) in;

// Shared memory
shared float tileData[256]; // shared: intra-group memory

// Uniforms
uniform float uTime; // uniform: global input

// Buffer access
layout(std430, binding = 0) buffer DataBuffer {
    float values[];
};

// Image access
layout(binding = 1, rgba32f) uniform image2D uImage;

// Memory qualifiers
// coherent, volatile, restrict, readonly, writeonly

void main() {
    uint idx = gl_GlobalInvocationID.x;
    values[idx] += sin(uTime); // buffer write
    imageStore(uImage, ivec2(idx, 0), vec4(values[idx])); // image write
}

```

14.2.4 OpenGL Shader Compiler

The OpenGL standard is defined in⁶¹. OpenGL is primarily designed for desktop computers and servers, whereas OpenGL ES is a subset tailored for embedded systems⁶².

Although shaders represent only a small part of the entire OpenGL software/hardware stack, implementing a compiler for them is still a significant undertaking. This is because a large number of APIs need to be supported. For instance, there are over 80 texture-related APIs alone⁶³.

A practical approach to implementing such a compiler involves generating LLVM extended intrinsic functions from the shader frontend (parser and AST generator). These intrinsics can then be lowered into GPU-specific instructions in the

⁶¹ <https://www.khronos.org/registry/OpenGL-Refpages/>

⁶² https://en.wikipedia.org/wiki/OpenGL_ES

⁶³ All the api listed in section 8.9 of https://www.khronos.org/registry/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.html#texture-functions

LLVM backend. The overall workflow is illustrated as follows:

Fragment shader

```
#version 320 es
uniform sampler2D x;
out vec4 FragColor;

void main()
{
    FragColor = texture(x, uv_2d, bias);
}
```

llvm-ir

```
...
!1 = !{"sampler_2d"}
!2 = !{i32 SAMPLER_2D} ; SAMPLER_2D is integer value for sampler2D, for example:_0x0f02
; A named metadata.
!x_meta = !{!1, !2}

define void @main() #0 {
    ...
    %1 = @llvm.gpu0.texture(metadata !x_meta, %1, %2, %3); ; %1: %sampler_2d, %2: %uv_
_2d, %3: %bias
    ...
}
```

asm of gpu

```
...
// gpu machine code
load $1, tex_a;
sample2d_inst $1, $2, $3 // $1: tex_a, $2: %uv_2d, $3: %bias

.tex_a // Driver set the index of gpu descriptor registers here
```

As shown at the end of the code above, the `.tex_a` memory address contains the Texture Object, which is bound by the driver during online compilation and linking. By binding a Texture Object (software representation) to a Texture Unit (hardware resource) via OpenGL API calls, the GPU can access and utilize Texture Unit hardware efficiently. This binding mechanism ensures that texture sampling and mapping are executed with minimal overhead during rendering.

For more information about LLVM extended intrinsic functions, please refer to⁶⁴.

```
gvec4 texture(gsample2D sampler, vec2 P, [float] bias);
```

GPUs provide *Texture Units* to accelerate texture access in fragment shaders. However, *Texture Units* are expensive hardware resources, and only a limited number are available on a GPU. To manage this limitation, the OpenGL driver can associate a *Texture Unit* with a *sampler* variable using OpenGL API calls. This association can be updated or switched between shaders as needed. The following statements demonstrate how to bind and switch *Texture Units* across shaders:

⁶⁴ <http://jonathan2251.github.io/lbd/funccall.html#add-specific-backend-intrinsic-function>

⁶⁵ <http://ogldev.atspace.co.uk/www/tutorial16/tutorial16.html>

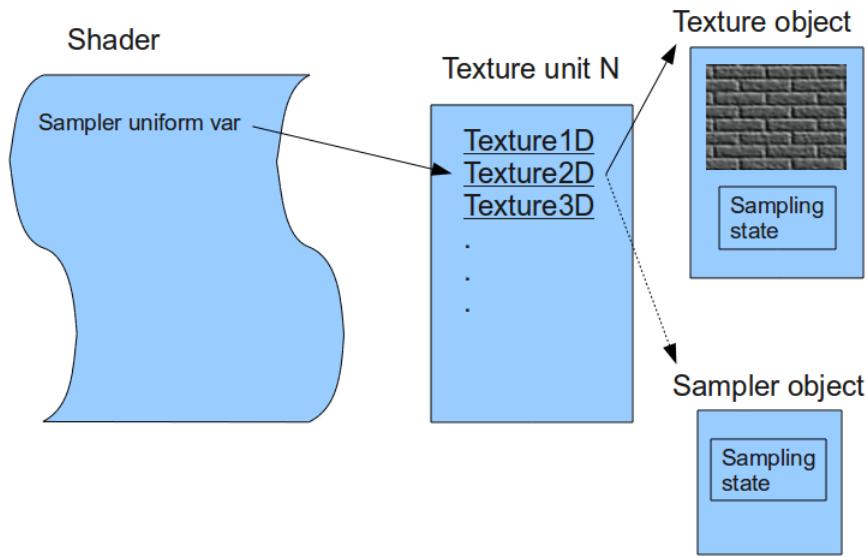


Fig. 14.35: Relationships between the texturing concept^{[Page 726, 65](#)}.

As shown in Fig. 14.35, the texture object is not bound directly to a shader (where sampling operations occur). Instead, it is bound to a *texture unit*, and the index of this texture unit is passed to the shader. This means the shader accesses the texture object through the assigned texture unit. Most GPUs support multiple texture units, though the exact number depends on the hardware capabilities^{[Page 726, 65](#)}.

A *texture unit*—also known as a *Texture Mapping Unit (TMU)* or *Texture Processing Unit (TPU)*—is a dedicated hardware component in the GPU that performs texture sampling operations.

The *sampler* argument in the texture sampling function refers to a *sampler2D* (or similar) uniform variable. This variable represents the texture unit index used to access the associated texture object^{[Page 726, 65](#)}.

Sampler Uniform Variables:

OpenGL provides a set of special uniform variables for texture sampling, named according to the texture target: *sampler1D*, *sampler2D*, *sampler3D*, *samplerCube*, etc.

You can create as many *sampler uniform variables* as needed and assign each one to a specific texture unit index using OpenGL API calls. Whenever a sampling function is invoked with a sampler uniform, the GPU uses the texture unit (and its bound texture object) associated with that sampler^{[Page 726, 65](#)}.

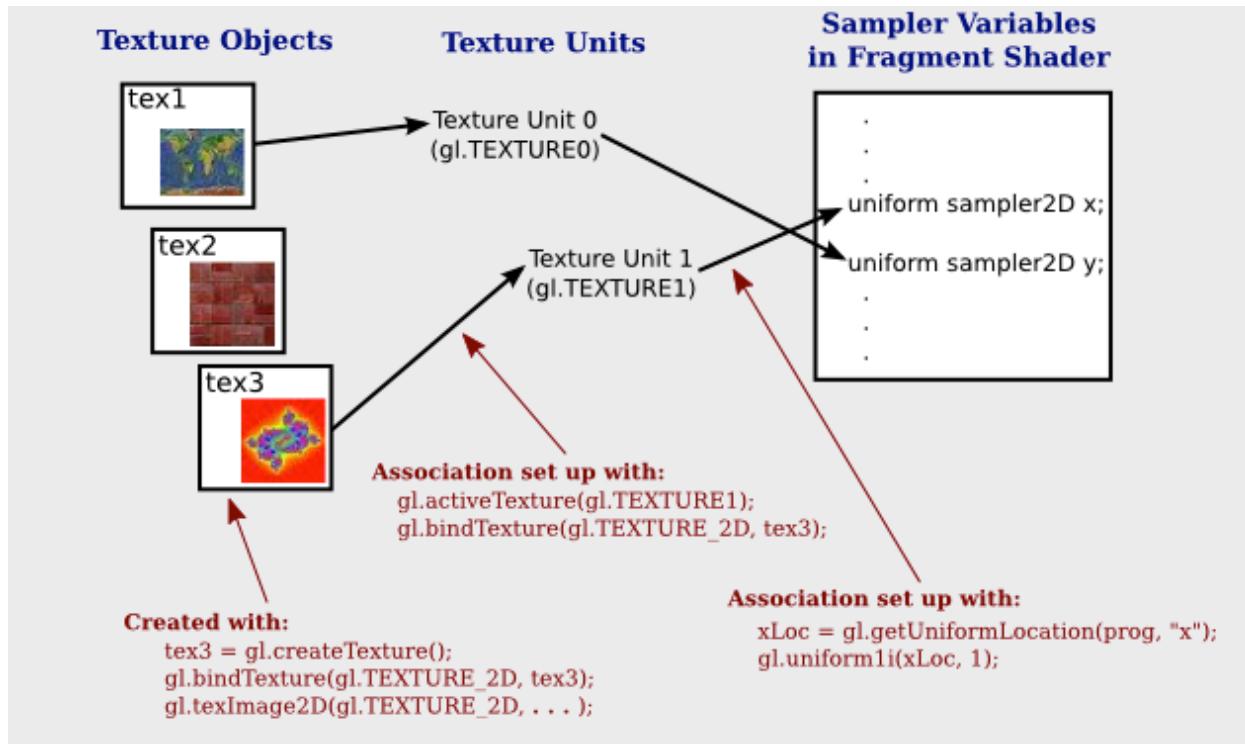
As shown in Fig. 14.36, the Java API function *gl.bindTexture()* binds a *Texture Object* to a specific *Texture Unit*. Then, using *gl.getUniformLocation()* and *gl.uniform1i()*, you associate the *Texture Unit* with a *sampler uniform variable* in the shader.

For example, *gl.uniform1i(xLoc, 1)* assigns *Texture Unit 1* to the sampler variable at location *xLoc*. Similarly, passing 2 would refer to *Texture Unit 2*, and so on⁶⁶.

The following figure illustrates how the OpenGL driver reads metadata from a compiled GLSL object, how the OpenGL API links *sampler uniform variables* to *Texture Units*, and how the GPU executes the corresponding texture instructions.

Explaining the detailed steps for the figure above:

⁶⁶ <http://math.hws.edu/graphicsbook/c6/s4.html>


 Fig. 14.36: Binding sampler variables^{Page 727, 66}.

1. To enable the GPU driver to bind the *texture unit*, the frontend compiler must pass metadata for each *sampler uniform variable* (e.g., *sampler_2d_var* in this example)⁶⁹ to the backend. The backend then allocates and embeds this metadata in the compiled binary file⁶⁷.
2. During the on-line compilation of the GLSL shader, the GPU driver reads this metadata from the compiled binary file. It constructs an internal table mapping each *sampler uniform variable* to its attributes, such as *{name, type, location}*. This mapping allows the driver to properly populate the *Texture Descriptor* in the GPU's memory, linking the variable to a specific *texture unit*.
3. API:

```
xLoc = gl.getUniformLocation(prog, "x"); // prog: GLSL program, xLoc: location of  
→sampler variable "x"
```

This API call queries the location of the *sampler uniform variable* named “*x*” from the internal table that the driver created after parsing the shader metadata.

The returned *xLoc* value corresponds to the location field associated with “*x*”, which will later be used to bind a specific *texture unit* to this sampler variable via *gl.uniform1i(xLoc, unit_index)*.

SAMPLER_2D is the internal representation (usually an integer) that identifies a *sampler2D* type in the shader.

4. API:

```
gl.uniform1i(xLoc, 1);
```

This API call binds the sampler uniform variable *x* (located at *xLoc*) to **Texture Unit 1**. It works by writing the integer value *1* to the internal GLSL program memory at the location of the sampler variable *x*, as indicated by *xLoc*.

⁶⁹ The type of ‘sampler uniform variable’ called “sampler variables”. <http://math.hws.edu/graphicsbook/c6/s4.html>

⁶⁷ This can be done by LLVM metadata. <http://llvm.org/docs/LangRef.html#namedmetadatastructure> <http://llvm.org/docs/LangRef.html#metadata>

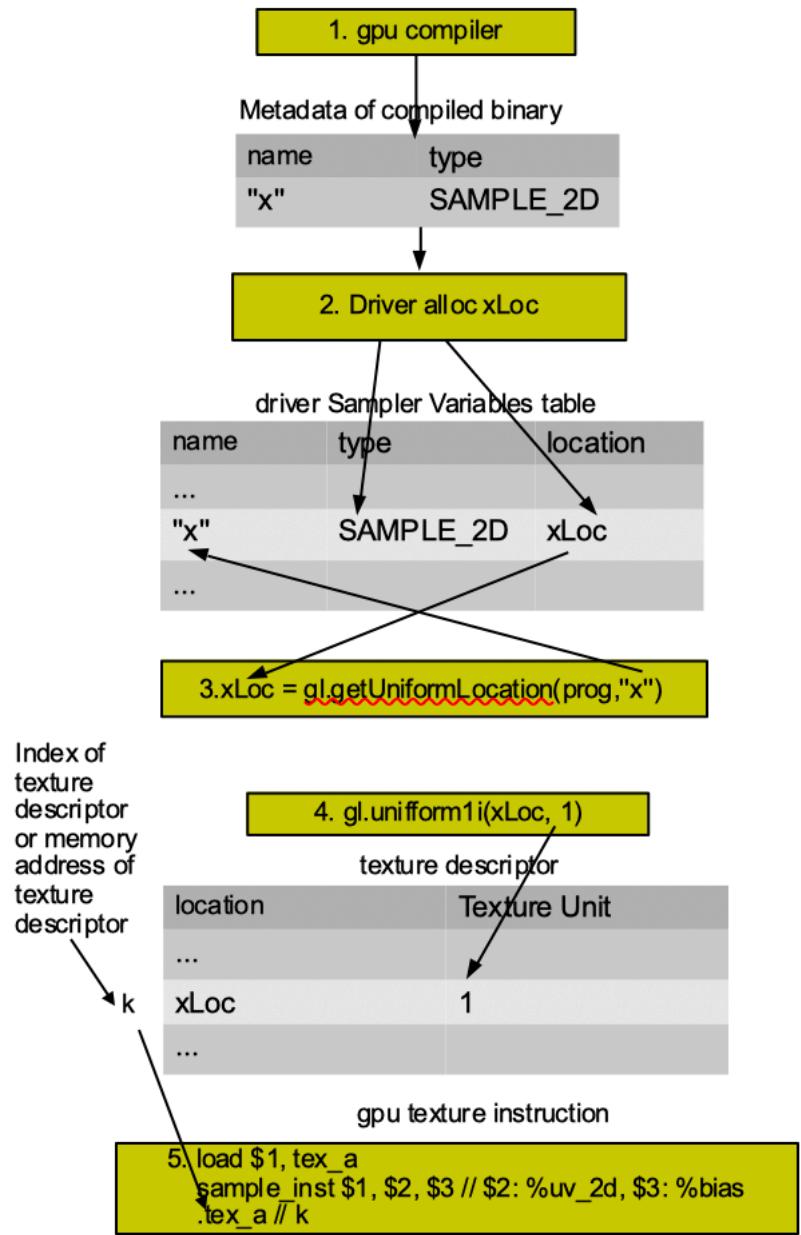


Fig. 14.37: Associating Sampler Variables and gpu executing texture instruction

```
{xLoc, 1} : 1 is 'Texture Unit 1', xLoc is the memory address of 'sampler uniform
↳variable' x
```

After this call, the OpenGL driver updates the **Texture Descriptor** table in GPU memory with this $\{xLoc, 1\}$ information.

Next, the driver associates the memory address or index of the GPU's texture descriptor with a hardware register or pointer used during fragment shader execution. For example, as shown in the diagram, the driver may write a pointer k to the $.tex_a$ field in memory.

This $.tex_a$ address is used by the GPU to locate the correct **Texture Unit** and access the texture object during shader execution.

5.

```
// gpu machine code
load $1, tex_a;
sample2d_inst $1, $2, $3 // $1: tex_a, $2: %uv_2d, $3: %bias

.tex_a // Driver set the index of gpu descriptor registers here at step 4
```

When executing the texture instructions from glsl binary file on gpu, the corresponding 'Texture Unit 1' on gpu will be executed through texture descriptor in gpu's memory because $.tex_a: \{xLoc, 1\}$. Driver may set texture descriptor in gpu's texture descriptors if gpu provides specific texture descriptors in architecture⁷⁰.

For instance, Nvidia texture instruction as follow,

```
// the content of tex_a bound to texture unit as step 5 above
tex.3d.v4.s32.s32 {r1,r2,r3,r4}, [tex_a, {f1,f2,f3,f4}];

.tex_a
```

The content of tex_a bound to texture unit set by driver as the end of step 4. The pixel of coordinates (x,y,z) is given by (f1,f2,f3) user input. The f4 is skipped for 3D texture.

Above tex.3d texture instruction load the calculated color of pixel (x,y,z) from texture image into GPRs $(r1,r2,r3,r4)=(R,G,B,A)$. And fragment shader can re-calculate the color of this pixel with the color of this pixel at texture image⁶⁸.

If it is 1d texture instruction, the tex.1d as follows,

5. GPU Execution of Texture Instruction

```
// GPU machine code
load $1, tex_a;
sample2d_inst $1, $2, $3 // $1: tex_a, $2: %uv_2d, $3: %bias

.tex_a // Set by driver to index of GPU descriptor at step 4
```

When the GPU executes the texture sampling instruction (e.g., sample2d_inst), it uses the $.tex_a$ address, which was assigned by the driver in step 4, to access the appropriate **Texture Descriptor** from GPU memory. This descriptor corresponds to **Texture Unit 1** because of the earlier API call:

```
gl.uniform1i(xLoc, 1);
```

⁷⁰ When performing a texture fetch, the addresses to read pixel data from are computed by reading the GPRs that hold the texture descriptor and the GPRs that hold the texture coordinates. It's mostly just general purpose memory fetching. <https://www.gamedev.net/forums/topic/681503-texture-units/>

⁶⁸ page 84: tex instruction, p24: texture memory https://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf

If the GPU hardware provides dedicated **texture descriptor registers** or memory structures, the driver maps `.tex_a` to those structures^{[Page 730, 70](#)}.

Example (NVIDIA PTX texture instruction):

```
// The content of tex_a is bound to a texture unit, as in step 4
tex.3d.v4.s32.s32 {r1,r2,r3,r4}, [tex_a, {f1,f2,f3,f4}];

.tex_a
```

Here, the `.tex_a` register holds the texture binding information set by the driver. The vector $\{f1, f2, f3\}$ represents the 3D coordinates (x, y, z) provided by the shader or program logic. The $f4$ value is ignored for 3D textures.

This `tex.3d` instruction performs a texture fetch from the bound 3D texture and loads the resulting color values into general-purpose registers:

- $r1$: Red
- $r2$: Green
- $r3$: Blue
- $r4$: Alpha

The **fragment shader** can then use or modify this color value based on further calculations or blending logic^{[Page 730, 68](#)}.

If a 1D texture is used instead, the texture instruction would look like:

```
// For compatibility with prior versions of PTX, the square brackets are not
// required and .v4 coordinate vectors are allowed for any geometry, with
// the extra elements being ignored.
tex.1d.v4.s32.f32 {r1,r2,r3,r4}, [tex_a, {f1}];
```

Since the ‘Texture Unit’ is a limited hardware accelerator on the GPU, OpenGL provides APIs that allow user programs to bind ‘Texture Units’ to ‘Sampler Variables’. As a result, user programs can balance the use of ‘Texture Units’ efficiently through OpenGL APIs without recompiling GLSL. Fast texture sampling is one of the key requirements for good GPU performance^{[Page 727, 66](#)}.

In addition to the API for binding textures, OpenGL provides the `glTexParameterI` API for texture wrapping^{[71](#)}. Furthermore, the texture instruction for some GPUs may include S# and T# values in the operands. Similar to associating ‘Sampler Variables’ to ‘Texture Units’, S# and T# are memory locations associated with texture wrapping descriptor registers. This allows user programs to change wrapping options without recompiling GLSL.

Even though the GLSL frontend compiler always expands function calls into inline functions, and LLVM intrinsic extensions provide an easy way to generate code through LLVM’s target description (TD) files, the GPU backend compiler is still somewhat more complex than the CPU backend.

(However, considering the effort required for the CPU frontend compiler such as Clang, or toolchains like the linker and GDB/LLDB, the overall difficulty of building a CPU compiler is not necessarily less than that of a GPU compiler.)

Here is the software stack of the 3D graphics system for OpenGL on Linux^{[Page 677, 12](#)}. The Mesa open source project website is here^{[72](#)}.

⁷¹ <https://learnopengl.com/Getting-started/Textures>

⁷² <https://www.mesa3d.org/>

14.3 GPU Architecture

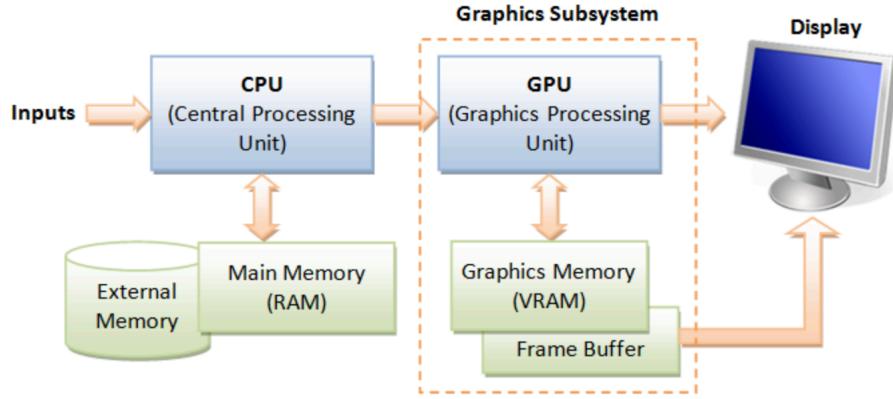


Fig. 14.38: Computer Graphics Hardware (figure from book^{Page 665, 1)}

14.3.1 GPU Hardware Units

A GPU (graphics processing unit) is built as a massively generic parallel processor of SIMD/SIMT architecture with several specialized processing units inside shown as Fig. 14.39 from the [section Graphics HW and SW Stack](#).

From compiler's view, GPU is shown as Fig. 14.40.

A GPU is not just “many cores”—it’s a mix of general-purpose compute clusters, specialized units, and the memory subsystem. It corresponds to the block diagram graph shown in Fig. 14.40.

The stages of the OpenGL rendering pipeline and the GPU hardware units that accelerate them as shown in Fig. 14.41:

Compute Cluster

- **Role:** Provide large-scale data-parallel execution. Each GPU contains many Streaming Multiprocessors (SMs) or Compute Units (CUs), each capable of executing thousands of threads in parallel.
- **Components:**
 - **Warp Scheduler** —Schedules groups of threads (Warps/Wavefronts), issues instructions in SIMT (Single Instruction, Multiple Threads) fashion.
 - **Registers** —Per-thread private storage, the fastest memory level.
 - **Shared Memory / L1 Cache** —On-chip memory close to the SM. Shared Memory is explicitly managed by the programmer for cooperation across threads, while L1 acts as a transparent cache.
 - **ALUs (FP/INT)** —Execute floating-point and integer arithmetic. They form the bulk of compute resources inside an SM.
 - **SFUs (Special Function Units)** —Execute transcendental functions such as sin, cos, exp, and reciprocal approximations.
 - **Load/Store Units** —Handle global, local, and shared memory access, interact with coalescing and caching logic.
 - **RegLess Staging Operands (RSO)** —Temporary operand buffers used to hide instruction and memory latencies.
- **Usage:**
 - Run programmable shaders (vertex, fragment, geometry, compute).

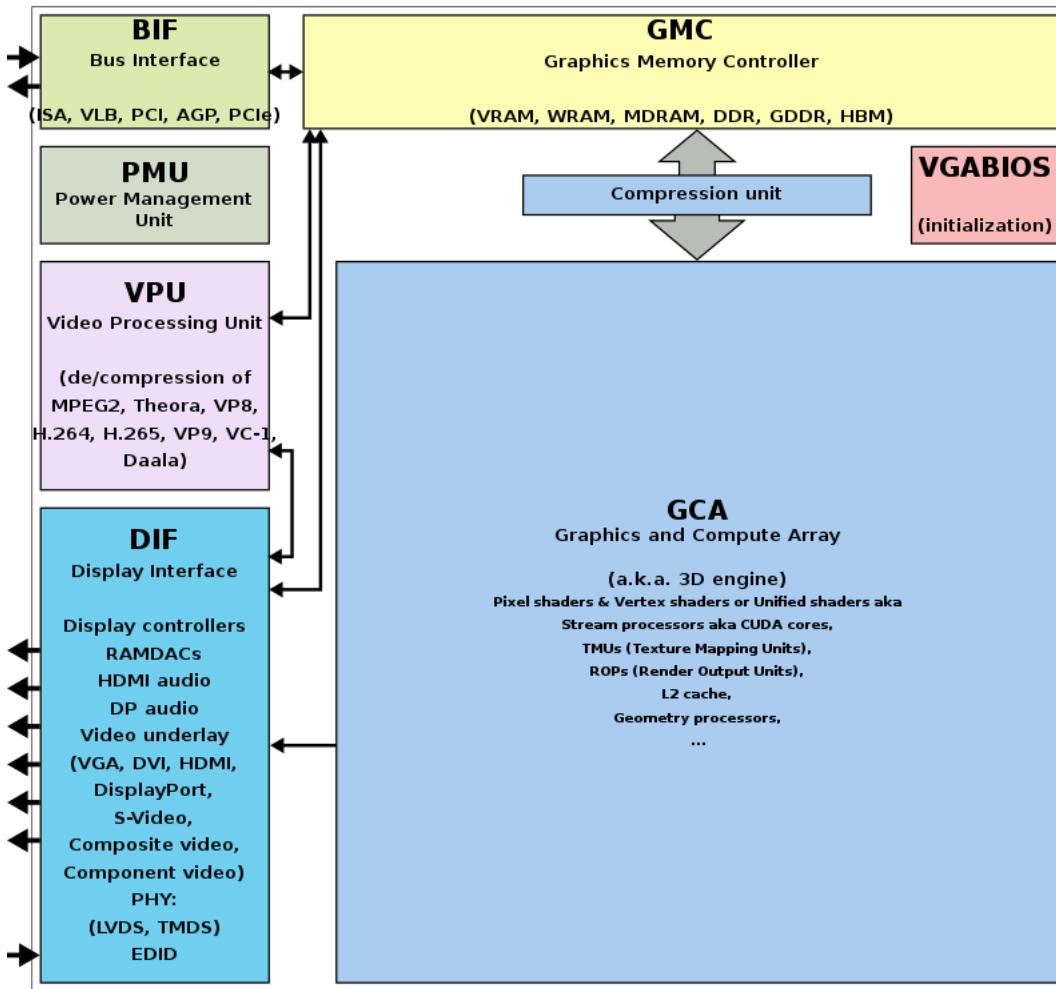


Fig. 14.39: Components of a GPU: GPU has accelerated video decoding and encoding^{Page 677, 9}

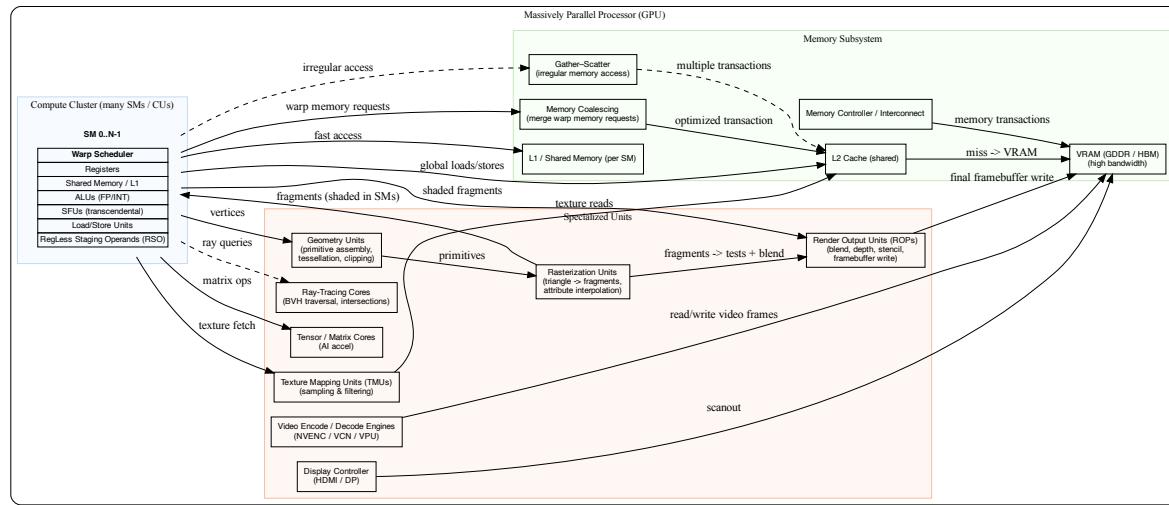


Fig. 14.40: Components of a GPU: SIMD/SIMT + several specialized processing units

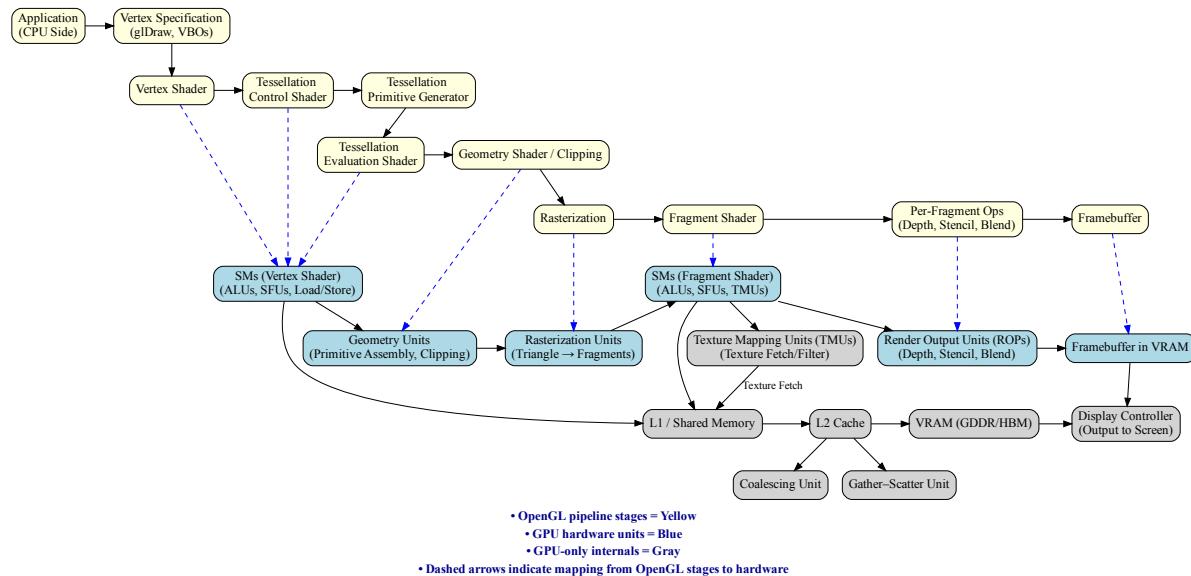


Fig. 14.41: The stages of OpenGL pipeline and GPU's acceleration components

- Perform general-purpose compute workloads (GPGPU).
- Issue texture fetch requests to TMUs.
- Interact with memory hierarchy via load/store units.
- Offload certain operations to Tensor or Ray-Tracing units.

Specialized Units

- **Role:** Accelerate fixed-function or specialized stages of the graphics and compute pipeline that are inefficient to run purely in SMs.
- **Components and Usage:**
 - **Geometry Units** —Assemble input vertices into primitives (points, lines, triangles). Perform tessellation (subdivide patches into smaller primitives), clipping (discard geometry outside view), and geometry shading.
Usage: Corresponds to the geometry/tessellation stage in the graphics pipeline.
 - **Rasterization Units** —Convert vector-based primitives into fragments (potential pixels). Interpolate per-vertex attributes (texture coordinates, normals, colors) across the surface of each primitive.
Usage: Bridge between geometry and fragment stages; produces fragments for SM fragment shading.
 - **Texture Mapping Units (TMUs)** —Fetch texture data from memory, apply filtering (bilinear, trilinear, anisotropic), and compute texel addresses (wrap, clamp).
Usage: Invoked during fragment shading inside SMs to provide sampled texture values.
 - **Render Output Units (ROPs)** —Handle late-stage pixel processing. Perform blending operations (alpha, additive), depth and stencil tests, and write final pixel values to the framebuffer in VRAM.
Usage: Final step of the graphics pipeline before display scanout.
 - **Tensor / Matrix Cores** —Perform fused-multiply-add (FMA) on large matrix tiles. Designed for machine learning, AI inference, and linear algebra.
Usage: Accelerate deep learning workloads or matrix-heavy compute kernels.
 - **Ray-Tracing Units (RT Cores)** —Traverse bounding volume hierarchies (BVH) and perform ray-primitive intersection tests in hardware.
Usage: Enable real-time ray tracing⁹⁵ by offloading intersection work from SMs.
 - **Video Engines** —Dedicated ASICs for video codec operations such as H.264/H.265/AV1 encode and decode.
Usage: Media playback, streaming, and video encoding without occupying SMs.
 - **Display Controller** —Reads final framebuffer images from VRAM and drives display interfaces like HDMI and DisplayPort.
Usage: Outputs rendered frames to monitors or VR headsets.

Memory Subsystem

- **Role:** Deliver high-bandwidth data access to thousands of threads while minimizing latency through caching and access optimization.
- **Components:**
 - **L1 / Shared Memory** —Closest to SMs. Shared Memory is explicitly used by programs for intra-block communication, while L1 acts as an automatic cache.
Usage: Boosts performance by keeping frequently accessed data close to execution units.

⁹⁵ <[https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics))>

- **L2 Cache** —Shared across all SMs. Reduces redundant traffic to VRAM and improves latency for reused data.

Usage: Provides intermediate caching layer for both compute and graphics.

- **VRAM (GDDR / HBM)** —External high-bandwidth DRAM. Stores textures, framebuffers, vertex/index buffers, and large compute datasets.

Usage: The main memory backing for all GPU workloads.

- **Interconnect / Memory Controller** —Orchestrates memory requests, manages access to VRAM, and ensures fairness between SMs.

Usage: Handles scheduling and distribution of memory transactions.

- **Memory Coalescing Unit** —Combines multiple per-thread memory requests from a Warp into fewer, wider transactions. Most effective for contiguous access patterns.

Usage: Improves memory bandwidth efficiency and reduces wasted cycles.

- **Gather—Scatter Unit** —Handles irregular or sparse memory accesses where coalescing is not possible. May break requests into multiple smaller transactions.

Usage: Supports workloads such as sparse matrix operations, graph traversal, or irregular data structures.

Data Flow Highlights

- **Graphics pipeline path:** Vertex data → Geometry Units → Rasterization Units → Fragment Shading (SMs) → TMUs (texture fetch) → ROPs (blend/depth/stencil) → VRAM (framebuffer).
- **Compute path:** SMs execute general-purpose kernels → optional offload to Tensor or RT cores → interact with caches → VRAM.
- **Memory behavior:** SMs issue memory requests → Coalescing Unit optimizes if possible → L2 cache → VRAM. For irregular access (e.g., sparse data), Gather—Scatter generates multiple VRAM transactions.
- **Display path:** Final framebuffer stored in VRAM → Display Controller → HDMI / DP scanout.

All Together

GPU provides the following hardware to accelerate graphics rendering pipeline as follows:

☒ Simplified Flow (OpenGL → Hardware)

1. Vertex Fetch → VRAM & Memory Controllers.
2. Vertex Shader → SM cores + Geometry Units.
3. Geometry/Tessellation → SM core + Geometry Units.
4. Rasterization → Rasterization units.
5. Fragment Shader → SM cores + TMUs (texture sampling).
6. Depth/Stencil/Blending → ROPs.
7. Framebuffer Write → L2 cache & VRAM → Display Controller.

Variable Rate Shading (VRS) Support

By utilizing certain GPU units as outlined below, Variable Rate Shading (VRS) can be supported⁹⁶.

- Rasterizer (Rasterization Units):
 - Decides how many fragments per pixel (or group of pixels) will actually be shaded.

⁹⁶ <https://developer.nvidia.com/vrworks/graphics/variablerateshading>

- Instead of generating 1 fragment per pixel, it may shade 1 fragment for a 2×2 or 4×4 block and reuse that result.
- Fragment Shader Cores (SMs/CUs):
 - Still run the shading code, but at a reduced frequency (fewer fragment invocations).
- ROPs (and pipeline integration):
 - Apply results to the framebuffer, handling blending/depth as usual.

14.3.2 SM (SIMT)

Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where a single central “Control Unit” broadcasts an instruction to multiple “Processing Units” for them to all optionally perform simultaneous synchronous and fully-independent parallel execution of that one instruction. **Each PU has its own independent data and address registers, its own independent Memory, but no PU in the array has a Program counter**⁷³.

Summary:

- Each Control Unit has a Program Counter (PC) and has tens of Processor Unit (PU).
- Each Processor Unit (PU) has its General Purpose Register Set (GPR) and stack memory.
- The PU is a pipeline execution unit compared to CPU architecture.

SM Hardware

The leading NVIDIA GPU architecture is illustrated in Fig. 14.42, where the scoreboard is shown without the mask field. This represents a SIMT pipeline with a scoreboard.

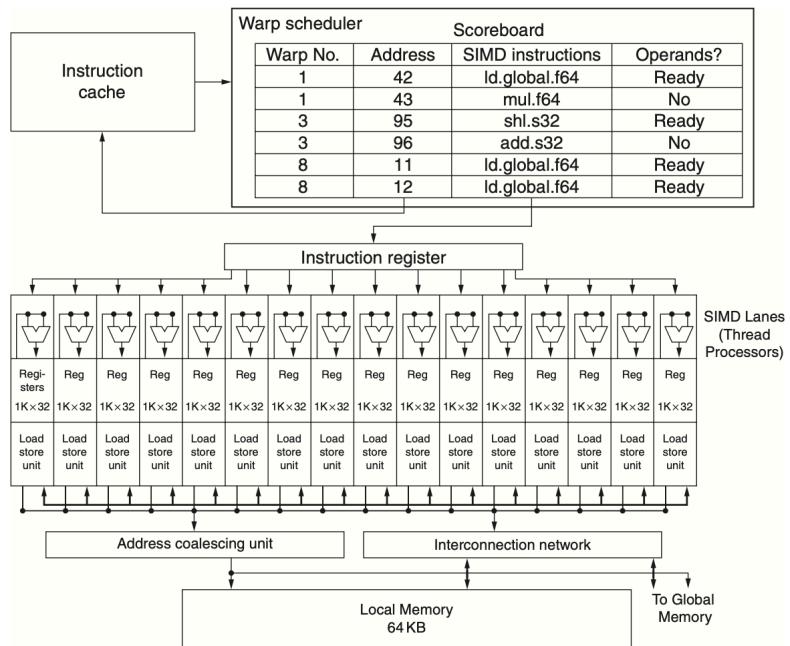


Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

Fig. 14.42: Simplified block diagram of a Multithreaded SIMD Processor. (figure from book^{Page 738, 79})

⁷³ https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads

Note

A SIMD Thread executed by SIMD Processor, a.k.a. SM, has 16 Lanes.

⁷⁹ The SIMD Thread Scheduler includes a scoreboard that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded SIMD Processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor (see Chapter 3), just that it is scheduling threads of SIMD instructions. Thus, GPU hardware has two levels of hardware schedulers: (1) the Thread Block Scheduler that assigns Thread Blocks (bodies of vectorized loops) to multi-threaded SIMD Processors, which ensures that Thread Blocks are assigned to the processors whose local memories have the corresponding data, and (2) the SIMD Thread Scheduler within a SIMD Processor, which schedules when threads of SIMD instructions should run. Book Figure 4.14 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

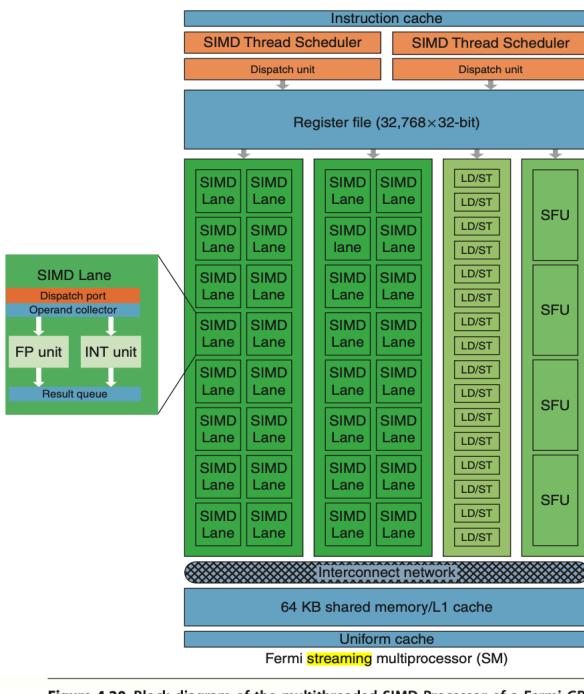


Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Fermi GPU. Each SIMD Lane has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The four Special Function units (SFUs) calculate functions such as square roots, reciprocals, sines, and cosines.

Fig. 14.43: Streaming Multiprocessor SM figure from book⁸⁰.



Fig. 14.44: SM figure from websites⁷⁶⁷⁷.

- Streaming Multiprocessor SM has two 16-way SIMD units and four special function units. Fermi has 32 SIMD Lanes and Cuda cores. SM has L1 and Read Only Cache (Uniform Cache) GTX480 has 48 SMs.
- In Fermi, ALUs run at twice the clock rate of rest of chip. So each decoded instruction runs on 32 pieces of data on the 16 ALUs over two ALU clocks⁷⁸. However after Fermi, the ALUs run at the same clock rate of rest of chip.
- As Fig. 14.43 in Fermi and Volta, it can dual-issue “float + integer” or “integer + load/store” but cannot dual-issue “float + float” or “int + int”.
- Uniform cache: used for storing constant variables in OpenGL (see *uniform of Pipeline Qualifiers*) and in OpenCL/CUDA.

Configurable maximum resident warps and allocated registers per thread as follows:

- Example: Fermi SM (SM 2.x)
 - Hawdware limit:
 - * Total registers per SM = $32,768 \times 32\text{-bit}$
 - * Max Warps per SM = 48
 - * Max threads per SM = 1536
 - * Max registers/thread = 63

⁸⁰ Book Figure 4.20 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

⁷⁶ <https://www.tomshardware.com/reviews/geforce-gtx-480,2585-18.html>

⁷⁷ https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_architecture_Whitepaper.pdf?utm_source=chatgpt.com

⁷⁸ https://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/lectures/02_multicore.pdf

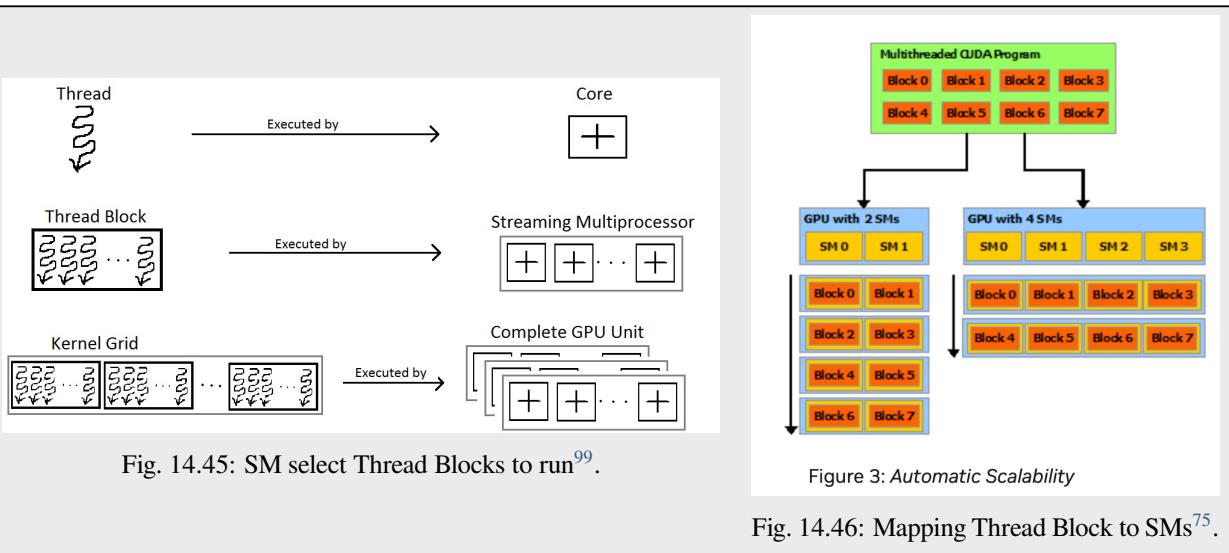
- Configuration: If each thread uses R registers:
 - * Max resident threads = $\text{floor}(32768 / R)$
 - * Max resident Warps = $\text{floor}(\text{Max resident threads} / 32)$
 - * E.g. **R=32: Max resident threads = 32768/32 = 1024, Max resident Warps = 1024/32 = 32.**
- After Fermi, the hardware limit for Maxwell, Pascal, Volta and Ampere are:
 - Hardware limit:
 - * Each SM includes 32 Cuda cores and Lanes → 32 active threads.
 - * Total registers per SM = 64K x 32-bit
 - * Max Warps per SM = 64
 - * Max threads per SM = 2048 (64 Warps x 32 threads)
 - * Max registers/thread = 255
 - Notes:
 - * Registers per thread: max number of registers **compiler can allocate to a thread**.
 - * The “registers per thread”limit (255) is a hardware/compiler limit, but the actual number used depends on the kernel. If a kernel uses too many registers per thread, occupancy drops (fewer threads can be resident).
 - * The “max threads per SM = 2048”is a theoretical upper limit; actual resident threads will also depend on shared memory usage, number of thread-blocks per SM, and register usage.

Note

A SIMD thread executed by a Multithreaded SIMD processor, also known as an SM, processes 32 elements.

As configuration above, the 32,768 registers per SM can be configured to each thread allocated 32 registers, Max resident Warps = 32.

Fermi has a mode bit that offers the choice of using 64 KB of SRAM as a 16 KB L1 cache with 48 KB of Local Memory or as a 48 KB L1 cache with 16 KB of Local Memory⁹².



SM Scheduling

- A GPU is built around an array of Streaming Multiprocessors (SMs). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors⁷⁵.
- Nvidia's GPUs:
Fermi (2010), Kepler (2012), Maxwell (2014), Pascal (2016), Volta (2017), Turing (2018), Ampere (2020), Ada Lovelace (2022), and Hopper (2022, for data centers).
- Two levels of scheduling:
 - Level 1: Thread Block Scheduler

For Fermi/Kepler/Maxwell/Pascal (pre-Volta): Warp-synchronous SIMT (lock-step in Warp):

A Warp includes 32 threads in Fermi GPU. Each Streaming Multiprocessor SM includes 32 Lanes in Fermi GPU, as shown in Fig. 14.45, the Thread Block includes a Warp (32 threads). According Fig. 14.46, more than one block can be assigned and run on a same SM.

When an SM executes a Thread Block, all the threads within the block are executed at the same time. If any thread in a Warp is not ready due to operand data dependencies, the scheduler switches context between Warps. During a context switch, all the data of the current Warp remains in the register file so it can resume quickly once its operands are ready⁹⁹.

⁹² Page 306 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

⁹⁹ <[https://en.wikipedia.org/wiki/Thread_block_\(CUDA_programming\)](https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming))>

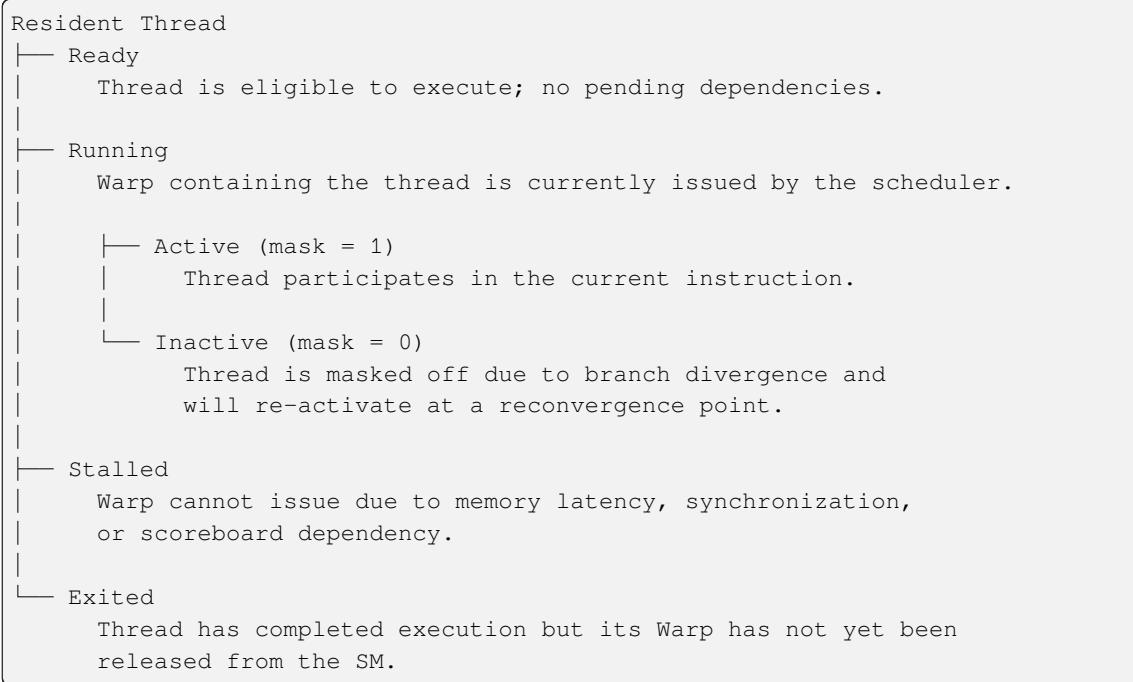
⁷⁵ <<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warps>>

Once a Thread Block is launched on a multiprocessor (SM), all of its Warps are **resident** until their execution finishes. Thus a new block is not launched on an SM until there is sufficient number of free registers for all Warps of the new block, and until there is enough free shared memory for the new block^{[Page 741, 99](#)}.

- Level 2: Warp Scheduler

Manages CUDA threads (resident threads) within the same Warp.

A **resident thread** is a thread whose execution context has been allocated on an SM (registers, Warp slot, shared memory). Once resident, the thread is always in exactly one of the following execution states.



Threads retain their registers and per-thread local memory during the stalled state. Therefore, **the context switch incurs almost no overhead compared to CPU threads**.

- * No pipeline flush: illustrate below.
- * No register save/restore
- * No stack frame swapping
- * **No OS involvement**
- * Takes roughly **1 cycle**

No pipeline flush because:

For Fermi/Kepler/Maxwell/Pascal (pre-Volta): Warp-synchronous SIMT (lock-step in Warp):

- * No data is saved/restored when switching to another Warp
- * Switching Warps = selecting a different Warp in the Warp scheduler
- * No pipeline flush

On an NVIDIA GPU, no pipeline flush occurs when a Warp stalls because the Warp's next instruction is **never issued until its operands are ready** as illustrated in *Warp scheduling in Level 1*. The stalled Warp simply stops issuing instructions, and its pipeline slot is taken by another ready Warp. When the stall condition clears, the Warp re-enters the pipeline by issuing the stalled instruction anew. No state is saved or restored.

For Volta, Turing, Ampere, Hopper: Independent Thread Scheduling:

- * No pipeline flush
 - Stalled threads simply do not issue instructions.
 - Other threads in the same warp continue issuing independently.
 - No pipeline flush needed and No data is saved/restored because instructions are tracked per thread, not per warp.

SIMT and SPMD Pipelines

This section illustrates the difference between SIMT and SPMD pipelines using the same pipeline stages: Fetch (F), Decode (D), Execute (E), Memory (M), and Writeback (W).

A GPU contains many SMs. The execution model between SMs is MIMD (Multiple Instructions, Multiple Data) when running different programs, or SPMD (Single Program, Multiple Data). However, within a single SM, the execution model is SIMD/SIMT.”

Low-end GPUs implement SIMD in their pipelines, where all instructions are executed in lockstep. High-end GPUs, however, approximate SPMD in their pipelines, meaning that instructions are interleaved within the pipeline, as shown below.

SPMD Programming Model vs SIMD/SIMT Execution

In the SISD of CPU, a thread is a single pipeline execution unit which can be issued at any specific address.

In a multi-core CPU running SPMD, each core can schedule and execute instructions at any program counter (PC). For example, core-1 may execute I(1–10), while core-2 executes I(31–35). **For GPU, however, within an SM, it is not possible to schedule thread-1 to execute I(1–10) while thread-2 executes I(31–35).**

As result, **there is no mainstream GPU that is truly hardware-SPMD** (where each thread has its own independent pipeline). All modern GPUs (NVIDIA, AMD, Intel) implement SPMD as a programming model, but under the hood they execute in SIMD lock-step groups (Warps or Wavefronts). GPUs expose an **SPMD programming model** (each thread runs the same kernel on different data). However, the hardware actually executes instructions in **SIMD/SIMT lock-step groups**.

An example to illustrate the difference between Pascal SIMT, Volta SIMT and SPMD.

Divergent Kernel Example:

```
=====
if (tid % 2 == 0) {           // even threads: long loop
    for (...) { loop_body } // many iterations
} else {                      // odd threads: short path
    C[tid] = A[tid] + B[tid];
}
```

Legend: F=Fetch, D=Decode, E=Execute, M=Memory, W=Writeback
S=Stall/masked-off, "..." = loop continues

=====
Pascal (lock-step SIMT with SIMT stack)

Cycle →	0	1	2	3	4	5	6	7	8	9	10	11	12	...
T0 even:	F	D	E	M	W	F	D	E	M	W	F	D	...	
T1 odd :	S	S	S	S	S	S	S	S	S	S	S	S	...	

(continues on next page)

(continued from previous page)

```
(Odd threads wait until even path completes, then:
... F D E M W → done
```

```
=====
Volta (SIMT with independent thread scheduling)
=====
```

```
Cycle → 0 1 2 3 4 5 6 7 8 9 10 11 ...
T0 even: F D E M W F D E M W F D ...
T1 odd : F D E M W done
(Odd thread issues its short path early,
interleaved with even loop instructions)
```

```
=====
True SPMD (CPU-like, fully independent threads)
=====
```

```
Cycle → 0 1 2 3 4 5 6 7 8 9 ...
T0 even: F D E M W F D E M W ...
T1 odd : F D E M W done
(Threads fetch/execute independently —
odd thread finishes immediately)
```

Note

SPMD and MIMD

When run a single program across all cores, SPMD and MIMD pipelines look the same.

The subsection *Mapping data in GPU* includes more details in Lanes masked.

Scoreboard purpose:

- GPU scoreboard = in-order issue, out-of-order completion
- CPU reorder buffer (ROB) = out-of-order issue + completion, but retire in-order - CPUs use a ROB to support out-of-order issue and retirement.

Comparsion of Volta and Pascal

In a lock-step GPU without divergence support, the scoreboard entries include only {Warp-ID, PC (Instruction Address), ...}. With divergence support (as in real-world GPUs), the scoreboard entries expand to {Warp-ID, PC, mask, ...}.

Volta (Cuda thread/SIMD Lane with PC, Program Counter and Call Stack)

GPU scoreboard = in-order issue, out-of-order completion

- SIMT GPU before Volta = scoreboard contains: { Warp ID + PC + Active Mask }
- Volta = scoreboard contains: { Warp ID + PC per thread (+ readiness per thread) }

Example for mutex¹⁰²

```
//  
__device__ void insert_after(Node *a, Node *b)
```

(continues on next page)

¹⁰² See the same Figures from <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

(continued from previous page)

```
{
    Node *c;
    lock(a); lock(a->next);
    ...
    unlock(c); unlock(a);
}
```

Assume that the mutex is contended across SMs but not within the same SM. On average, each thread spends 10 cycles executing the insert_after operation without resource contention, and 20 cycles when accounting for contention. Therefore, the average total execution time for 32 threads in an SM is:

- Volta: 20 cycles
- Pascal: 640 cycles (20 cycles \times 32 threads, due to lack of independent progress inside a Warp)

14.3.3 Processor Units and Memory Hierarchy in NVIDIA GPU⁸¹

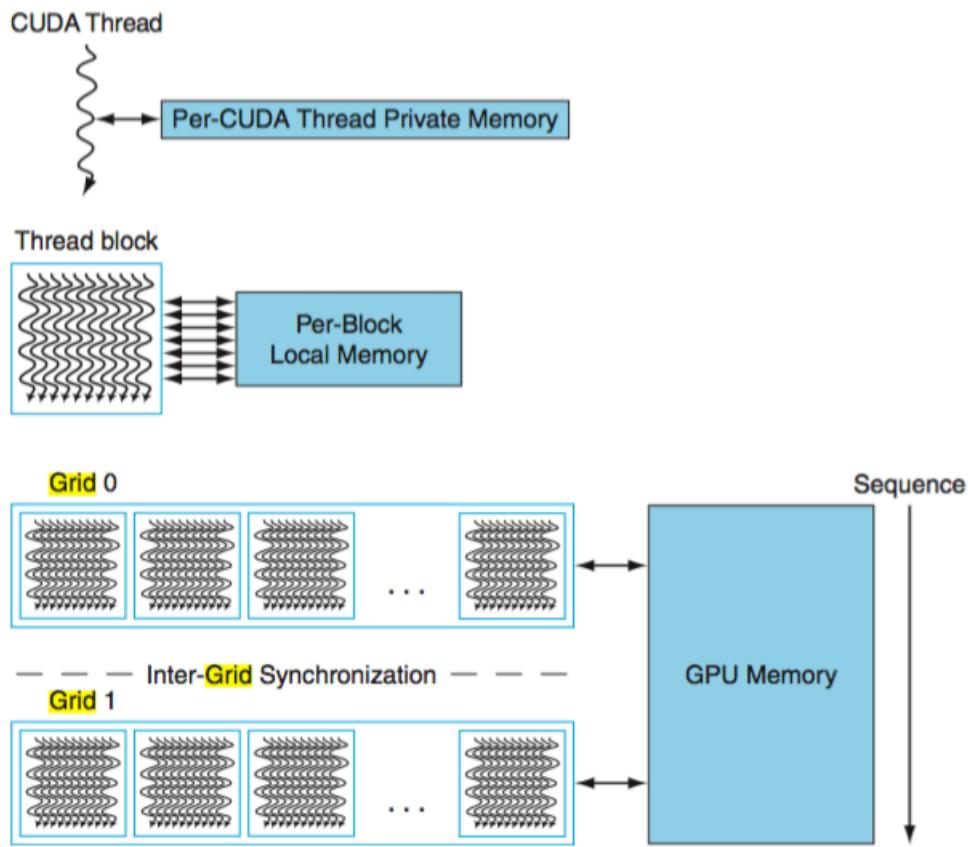


Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Fig. 14.47: GPU memory (figure from book [Page 746, 82](#)).

⁸¹ chatgpt: Give me a memory hierarchy for L1, L2, local memory, shared memory for these processing units of hierarchy in reST and separate dot graph.

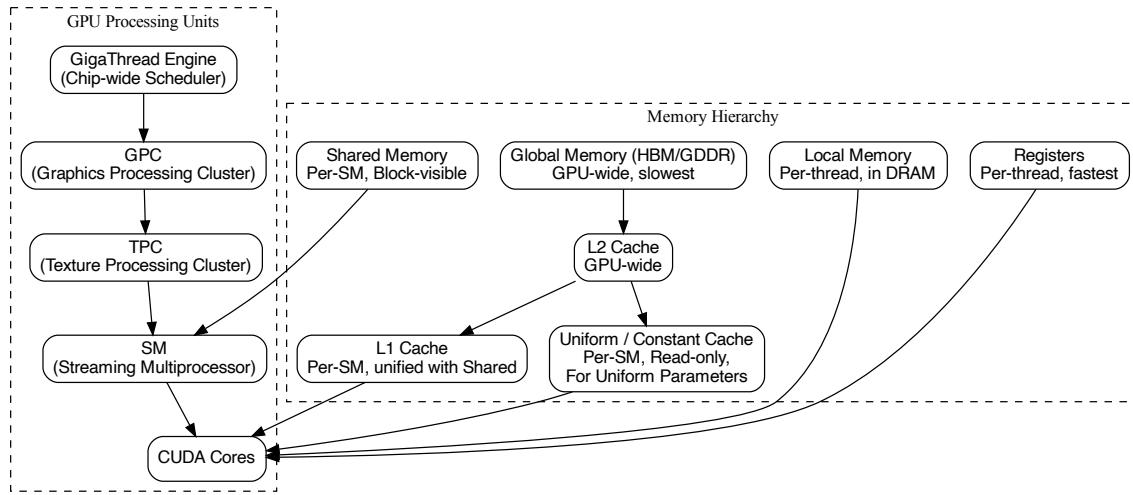


Fig. 14.48: Processor Units and Memory Hierarchy in NVIDIA GPU **Local Memory is shared by all threads and Cached in L1 and L2**. In addition, the **Shared Memory** is provided to use per-SM, not cacheable.

Illustrate L1, L2 and Global Memory used by SM and whole chip of GPU as Fig. 14.49.

The Fig. 14.48 illustrates the memory hierarchy in NVIDIA GPU. The Cache flow for 3D Model Information, Animation Parameters, and GLSL Variables is as follows:

3D Model Information:

- VBO/IBO → Global → L2 → L1 → Registers
- Material constants → Uniform Cache → Registers

Animation Parameters:

- Bone matrices → Uniform Cache → Registers
- Morph targets → Global → L2 → L1 → Registers
- Shared bone data (compute) → Shared Memory

GLSL Variables:

- uniform → Uniform Cache
- in (vertex attributes) → Global → L2 → L1
- out (varyings) → Registers → Interpolators
- buffer (SSBO) → Global → L2 → L1
- shared → Shared Memory
- local arrays → Registers or Local Memory

More details of the NVIDIA GPU memory hierarchy are described as follows:

- **Registers**
 - Per-thread, fastest memory, located in CUDA cores, as illustrated also in Fig. 14.43.

⁸² Book Figure 4.17 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

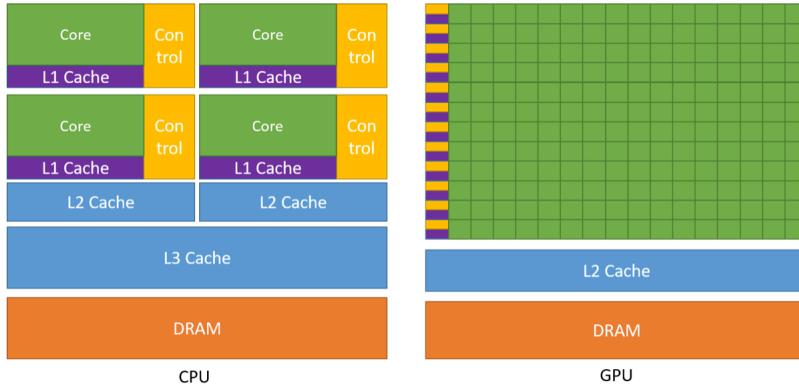


Figure 1: The GPU Devotes More Transistors to Data Processing

Fig. 14.49: **L1 Cache: Per-SM, Coherent across all 16 Lanes in the same SM.** L2 Cache: Coherent across all SMs and GPCs. Global Memory (DRAM: HBM/GDDR). Both HBM and GDDR are DRAM. GDDR (Graphics DDR) — optimized for GPUs (GDDR5, GDDR6, GDDR6X). HBM (High Bandwidth Memory) — 3D-stacked DRAM connected via TSVs (Through-Silicon Vias) for extremely high bandwidth and wide buses Page 741, 75.

- Configurable maximum resident warps and allocated registers per thread following Fig. 14.43.
- Latency: ~1 cycle.
- **Uniform / Constant cache**
 - Stored constant variables in OpenGL and OpenCL/CUDA, as illustrated in Fig. 14.43.
- **Local Memory**
 - Per-thread, stored in global DRAM.
 - Cached in L1 and L2.
 - Latency: high, depends on cache hit/miss.
- **Shared Memory**
 - **Per-SM, shared across threads in a Thread Block as shown in Fig. 14.43.**
 - **On-chip, programmer-controlled.**
 - Latency: ~20 cycles.
- **L1 Cache**
 - Per-SM, unified with shared memory.
 - Hardware-managed.
 - Latency: ~20 cycles.
- **L2 Cache**
 - Shared across the entire GPU chip.
 - **Coherent across all SMs and GPCs as shown in Fig. 14.49.**
- **Global Memory (DRAM: HBM/GDDR)**
 - Visible to all SMs across all GPCs.
 - Highest latency (~400–800 cycles).

GPU Hierarchy Context

- **GigaThread Engine (chip-wide scheduler)**
 - Contains multiple GPCs.
 - * Fermi (2010): up to 4 GPCs per chip.
 - * Pascal GP100 (Tesla P100): 6 GPCs.
 - * Volta GV100 (Tesla V100): 6 GPCs.
 - Distributes Thread Blocks to all GPCs.
- **GPC (Graphics Processing Cluster)**
 - Contains multiple TPCs.
- **TPC (Texture Processing Cluster)**
 - Groups 1–2 SMs.
- **SM (Streaming Multiprocessor)**
 - Contains CUDA cores, registers, shared memory, L1 cache.
- **CUDA Cores**
 - Execute threads with registers and access the memory hierarchy.
- A Warp of 32 threads is mapped across 16 Lanes. If each Lane has 2 Chimes, it may support dual-issue or time-sliced execution as Fig. 14.51.

In the following matrix multiplication code, the 8096 elements of matrix $A = B \times C$ are mapped to Thread Blocks, SIMD Threads, Lanes, and Chimes as illustrated in the Fig. 14.52. In this example, it run on **time-sliced execution**.

Listing 14.2: MATMUL CUDA Example

```
// Invoke MATMUL with 256 threads per Thread Block
__host__
int nblocks = (n + 255) / 512;
matmul<<<nblocks, 255>>>(n, A, B, C);
// MATMUL in CUDA
__device__
void matmul(int n, double A, double *B, double *C) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) A[i] = B[i] + C[i];
}
```

Explain the mapping and execution in Fig. 14.52 for *MATMUL CUDA Example* using the terminology from Fig. 14.50 and the previous sections of this book, presented in the table below.

⁹¹ Book Figure 4.12 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

⁷² Book Figure 4.13 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
Memory hardware	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
Memory hardware	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Figure 4.12 Quick guide to GPU terms used in this chapter. We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: Program Abstractions, Machine Objects, Processing Hardware, and Memory Hardware. Figure 4.21 on page 309 associates vector terms with the closest terms here, and Figure 4.24 on page 313 and Figure 4.25 on page 314 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

Fig. 14.50: Terms in Nvidia's gpu (figure from book^{Page 748, 91})

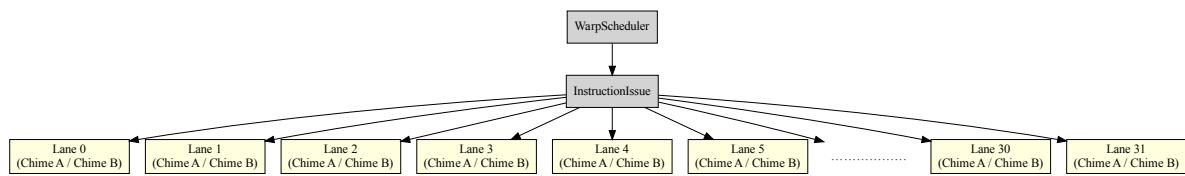


Fig. 14.51: In **dual-issue mode**, Chime A carries floating-point data while Chime B carries integer data—both issued by the same CUDA thread. In contrast, under **time-sliced execution**, Chime A and Chime B carry either floating-point or integer data independently, and are assigned to separate CUDA threads.

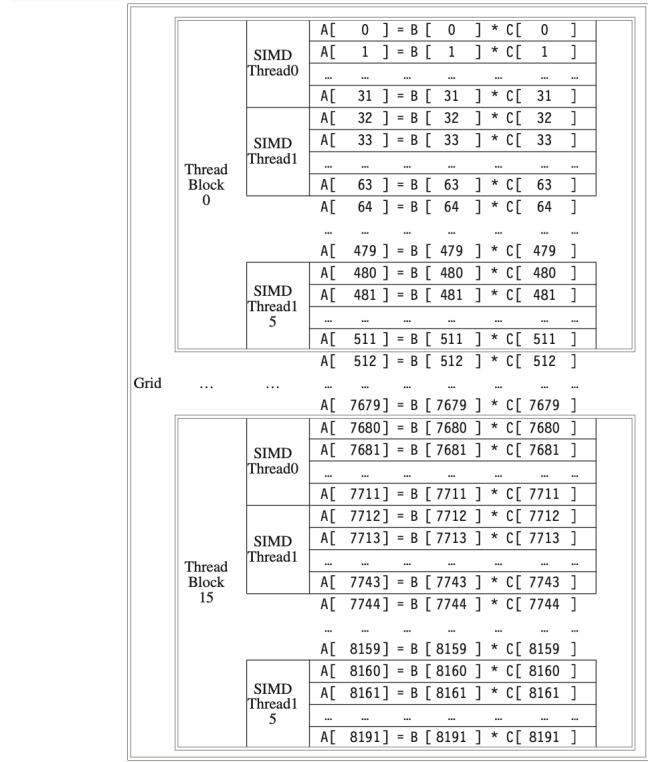


Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector–vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculated 32 elements per instruction, and in this example each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via Local Memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 16 for Tesla-generation GPUs and 32 for the later Fermi-generation GPUs.)

Fig. 14.52: Mapping 8192 elements of matrix multiplication for Nvidia's GPU (figure from [Page 748, 74](#)). SIMT: 16 SIMD threads in one Thread Block.

Table 14.9: Summary terms for GPU.

Terms	Structure	Description
Grid, Giga Thread Engine	Each loop (Grid) consists of multiple Thread Blocks.	Grid is Vectorizable Loop as Fig. 14.50. The hardware scheduler Guda Thread Engine schedules the Thread Blocks to SMs.
Thread Block	In this example, each Grid has 16 Giga Thread ⁹⁴ .	Each Thread Block is assigned 512 elements of the vectors to work on. As Fig. 14.52, it assigns 16 Thread Block to 16 SMs. Giga Thread is the name of the scheduler that distributes Thread Blocks to Multiprocessors, each of which has its own SIMD Thread Scheduler ⁹⁴ . More than one Block can be mapped to a same SM as the explanation in “Level 1: Thread Block Scheduler” for Fig. 14.46.
Streaming Multiprocessor, SM, GPU Core (Warp) ¹⁰⁰	Each SIMD Processor has 16 SIMD Threads.	Each SIMD processor includes local memory, as in Fig. 14.47. Local memory is shared among SIMD Lanes within a SIMD processor but not across different SIMD processors. A Warp has its own PC and may correspond to a whole function or part of a function. Compiler and runtime may assign functions to the same or different Warps ¹⁰¹ .
Cuda core	Fermi has 32 Cuda cores in a SM as Fig. 14.44.	A CUDA core is the scalar execution unit inside an SM. It is capable of executing one integer or floating-point instruction from one Lane of a Warp. The CUDA core is analogous to an ALU pipeline stage in a CPU.
Cuda Thread	Each SM can configure to have different number of resident threads.	Fermi can configure Max resident threads = $32768/32 = 1024$ for 32 registers/thread in a SM as mentioned earlier. A CUDA thread is the basic unit of execution defined in CUDA’s programming model. Each thread executes the kernel code independently with its own registers, program counter (PC), and per-thread local memory. Each Thread has its TLR (Thread Level Registers) allocated from Register file ($32768 \times 32\text{-bit}$) by SIMD Processor (SM) as Fig. 14.43.
SIMD Lane	Each SIMD Thread has 32 Lanes.	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. It is a vector instruction with processing 32-elements. A Warp of 32 threads is mapped across 32 Lanes. Lane = per-thread execution slot inside a Warp. If each Lane has 2 Chimes, it may support dual-issue or time-sliced execution as Fig. 14.51.
Chime	Each Lane has 2 Chimes.	A Chime represents one “attempt” or opportunity for issuing instructions from Warps. In Fermi (SM2.x): Each SM has 2 Warp schedulers. Each Warp scheduler has 2 dispatch units (dual-issue, but with constraints, it can issue “float + load/store” for “fadd and load C[i]” in this example).

References

- NVIDIA GPU Architecture Overview
- Understanding Warps and Threads

14.3.4 Memory Subsystem

Address Coalescing and Gather-scatter

Brief description is shown in Fig. 14.53.

⁹⁴ Figure 4.15 Floor plan of the Fermi GTX 480 GPU of A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design). **Giga Thread** is the name of the scheduler that distributes Thread Blocks to Multiprocessors, each of which has its own SIMD Thread Scheduler.

¹⁰⁰ Copilot: Is GPU core meaning SM in NVidia?

¹⁰¹ Book Figure 4.14 and 4.24 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

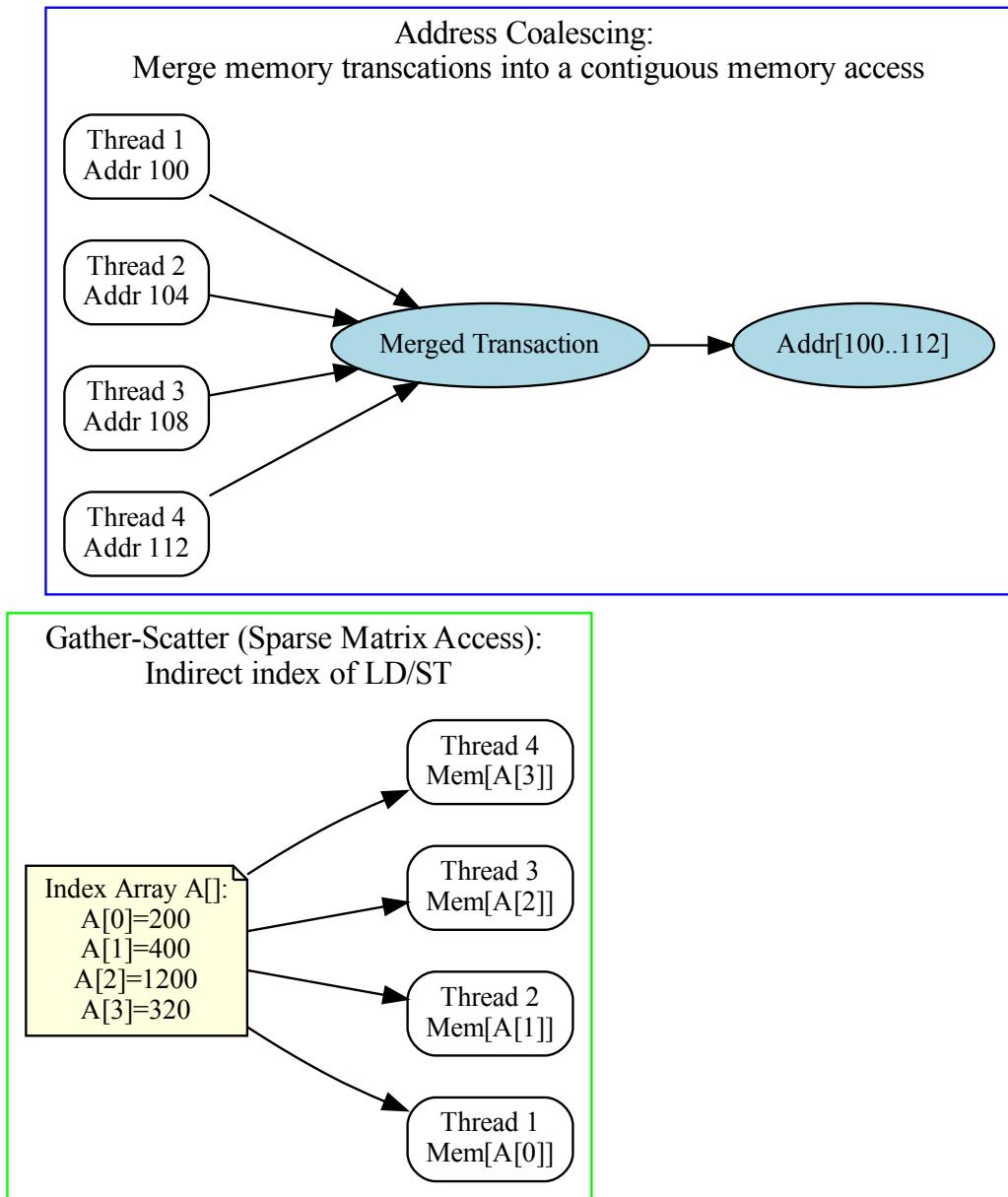


Fig. 14.53: Coalescing and Gather-scatter

The Load/Store Units (LD/ST) is important because memory latency is huge compared to ALU ops. Some GPUs provide Address Coalescing and gather-scatter to accelerate memory access.

- Address Coalescing: **Memory coalescing is the process of merging memory requests from threads in a Warp (NVIDIA: 32 threads, AMD: 64 threads) into as few memory transactions as possible.**
 - Cache miss (global memory/DRAM): Coalescing = big performance improvement.
 - Cache hit (L1/L2): Coalescing = smaller benefit, since cache line fetch already amortizes cost.
 - Note that unlike vector architectures, GPUs don't have separate instructions for sequential data transfers, strided data transfers, and gather-scatter data transfers. All data transfers are gather-scatter! To regain the efficiency of sequential (unit-stride) data transfers, GPUs include special Address Coalescing hardware to recognize when the SIMD Lanes within a thread of SIMD instructions are collectively issuing sequential addresses. That runtime hardware then notifies the Memory Interface Unit to request a block transfer of 32 sequential words. To get this important performance improvement, the GPU programmer must ensure that adjacent CUDA Threads access nearby addresses at the same time that can be coalesced into one or a few memory or cache blocks, which our example does⁹⁸.
- Gather-scatter data transfer: **HW support sparse vector access is called gather-scatter.** The VMIPS instructions are LVI (load vector indexed or gather) and SVI (store vector indexed or scatter)⁹⁷.

1. Address Coalescing in GPU Memory Transactions

Definition: Memory coalescing is the process of merging memory requests from threads in a Warp (NVIDIA: 32 threads, AMD: 64 threads) into as few memory transactions as possible.

How It Works:

- If threads access **contiguous and aligned addresses**, the hardware combines them into a single memory transaction.
- If threads access **strided or random addresses**, the GPU must issue multiple transactions, wasting bandwidth.

Examples:

- *Coalesced (efficient):*

```
// Each thread accesses consecutive elements
value = A[threadId];
```

→ One transaction for 32 threads.

- *Non-coalesced (inefficient):*

```
// Each thread accesses strided elements
value = A[threadId * 100];
```

→ Many transactions required due to striding.

2. Gather—Scatter in Sparse Matrix Access

Definition: Gather—scatter refers to memory operations where each GPU thread in a Warp loads from or stores to irregular memory addresses. This is common in sparse matrix operations, where non-zero elements are stored in compressed formats.

Sparse Matrix Example (CSR format):

- *CSR (Compressed Sparse Row)* stores three arrays:

- *values[]*: non-zero entries of the matrix

⁹⁸ Page 300 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

⁹⁷ Page 280 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

- `colIndex[]`: column indices for each non-zero
- `rowPtr[]`: index into `values[]` for each row
- Sparse matrix-vector multiplication (SpMV):

```
for row in matrix:
    for idx = rowPtr[row] to rowPtr[row+1]:
        col = colIndex[idx];           // gather index
        val = values[idx];            // gather nonzero
        y[row] += val * x[col];       // scatter result
```

Characteristics:

- **Gather:** Each thread loads from potentially scattered locations (`values[idx]` or `x[col]`).
- **Scatter:** Results may be written back to irregular output locations (`y[row]`).
- **Challenge:** These accesses often break memory coalescing, leading to multiple memory transactions. An example is shown as follows:

Summary:

- Gather—scatter is fundamental for sparse matrix access but typically results in non-coalesced memory patterns.
- Address coalescing is critical for high GPU throughput; restructuring data to improve coalescing often provides significant performance gains.

VRAM dGPU

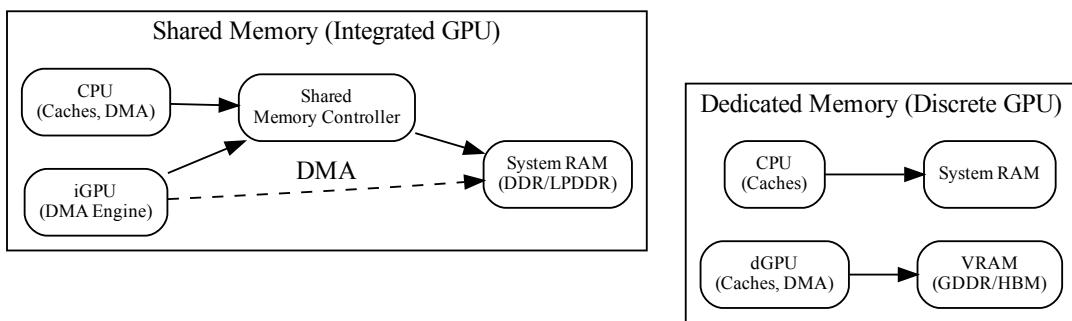


Fig. 14.54: iGPU versus dGPU

Reason:

1. Since CPU and GPU have different requirements, a shared memory design cannot match the performance of dedicated GPU memory.
2. In systems with shared memory (like integrated GPUs), both the CPU and GPU access the same physical memory (DRAM). This leads to several forms of contention:
 - a. Cache Coherency Overhead
 - b. DMA Contention

- c. Bus & Memory Controller Bottleneck

Advantages:

A discrete GPU has its own dedicated memory (VRAM) while an integrated GPU (iGPU) shares memory with the CPU as shown in Fig. 14.54.

Dedicated GPU memory (VRAM) outperforms shared CPU-GPU memory due to higher bandwidth, lower latency, parallel access optimization, and no contention with CPU resources.

Feature	Shared Memory (CPU + iGPU)	Dedicated GPU Memory (dGPU)
Bandwidth	Lower (DDR/LPDDR)	Higher (GDDR/HBM)
Latency	Higher	Lower
Parallel Access	Limited	Optimized for many threads
Cache Coherency	Required (with CPU)	Not required
DMA Bandwidth	Shared with CPU	GPU has exclusive DMA access
Memory Contention	Yes	No
Performance	Lower: Bandwidth bottlenecks, CPU-GPU interference and Cache/DMA conflicts	Higher: Wide memory bandwidth, Parallel thread access and Low latency memory access

Dedicated memory allows the GPU to run high-throughput workloads without interference from the CPU. It provides **(1). wide bandwidth, (2). optimized parallel access, and (3). low-latency paths**, avoiding cache and DMA conflicts for superior performance.**

(1). Wide bandwidth: Dedicated GPU memory (VRAM) is often based on GDDR6, GDDR6X, or HBM2/3, which are much faster than standard system RAM (DDR4/DDR5).

Typical bandwidths:

- GDDR6: ~448–768 GB/s
- HBM2: up to 1 TB/s+
- DDR5 (shared memory): ~50–80 GB/s

Impact: Faster access to textures, vertex buffers, and framebuffers—critical for rendering and compute tasks.

(2). Optimized parallel access:

- VRAM is optimized for the massively parallel architecture of GPUs.
- It allows thousands of threads to access memory simultaneously without stalling.

Shared system memory is optimized for CPU access patterns, not thousands of GPU threads.

(3). Low-latency paths:

- Dedicated memory is physically closer to the GPU die.
- No need to traverse the PCIe bus like discrete GPUs accessing system RAM.

In shared memory systems (like integrated GPUs), memory access may have to go through a memory controller shared with the CPU, adding delay.

RegLess-style architectures⁴⁶

Note

RegLess remains a research concept, not (as far as public evidence shows) a commercial design in shipping GPUs.

Difference: Add **Staging Buffer** between Register Files and Execution Unit.

This section outlines the transition from traditional GPU operand coherence using a monolithic register file and L1 data cache, to a RegLess-style architecture that employs operand staging and register file-local coherence.

✓ Operand Delivery in Traditional GPU: Fig. 14.55:

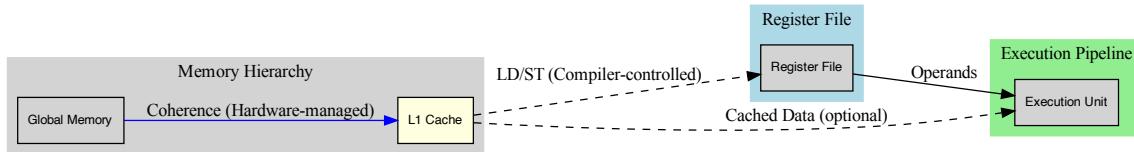


Fig. 14.55: Operand Delivery in Traditional GPU (Traditional Model: Register File + L1 Cache)

Architecture:

- Large monolithic register file per SM (e.g., 256KB, Maxwell, Pascal, Volta and Ampere have 64K x 32-bit register file per SM, see *Configurable maximum resident warps and allocated registers per thread*)
- Coherent with L1 data cache via write-through or write-back policies

Challenges:

- High energy cost due to cache coherence traffic
- Complex invalidation and synchronization logic
- Register pressure limits Warp occupancy (limit the number of active Warps)
- Redundant operand tracking across register file and cache

Example:

```
v1 = normalize(N)
v2 = normalize(L)
v3 = dot(v1, v2)
v4 = max(v3, 0.0)
v5 = mul(v4, color)

# All operands reside in register file and may be cached in L1
```

✓ Operand Delivery in RegLess GPU (with L1 Cache in LD Path): Fig. 14.56:

Description

- **Global Memory:** Source of all operands and data.

⁴⁶ <https://cccp.eecs.umich.edu/papers/jklooste-micro17.pdf>

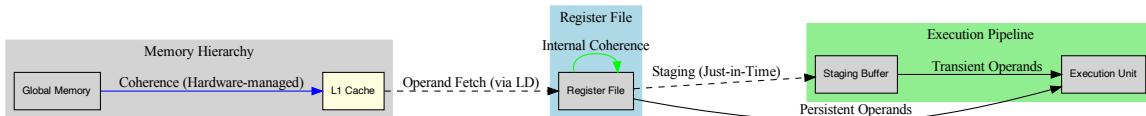


Fig. 14.56: Operand Delivery in RegLess GPU (with L1 Cache in LD Path)

- **L1 Cache:** Participates in memory hierarchy; may serve LD requests.
- **Register File:** Receives operands via LD; stages them into Staging Buffer for Transient Operands.
- **Staging Buffer:** Holds transient operands for immediate execution.
- **Execution Unit:** Consumes operands from Staging Buffer for Transient Operands and Register File for Persistent Operands.

Notes

- **L1 Cache is not part of staging**—it only serves LDs.
- **Dashed arrows:** Compiler-controlled operand movement.
- **Solid arrows:** Operand delivery to execution.
- **Green self-loop:** Internal coherence within Register File.

RegLess Model: Staging-Aware Register File

Architecture:

- Smaller register file (e.g., 64–128KB per SM)
- For Transient Operands, no L1 cache coherence required
- Operands staged dynamically based on lifetime

Key Concepts:

- Region slicing: compiler divides computation into operand regions
- Operand tagging: transient, intermediate, persistent
- Metadata compression: region-level hints, not per-instruction lifetimes

Benefits:

- ~75% reduction in register file size
- ~11% energy savings
- Simplified coherence model
- Improved Warp occupancy

Example with Operand Staging:

```
v1 = normalize(N)      # transient
v2 = normalize(L)      # transient
v3 = dot(v1, v2)       # intermediate
v4 = max(v3, 0.0)      # intermediate
v5 = mul(v4, color)    # persistent
```

(continues on next page)

(continued from previous page)

```
# v1 and v2 staged briefly, v3-v4 may be staged or registered, v5 fully
# registered
```

Compiler-Hardware Interface

Compiler Responsibilities:

- Emit structured IR with operand usage hints
- Slice computation graph into regions
- Avoid explicit staging register allocation

Hardware Responsibilities:

- Interpret operand lifetime metadata
- Dynamically stage operands or allocate registers
- For Transient Operands, eliminate L1 cache coherence logic

Metadata Compression Techniques:

- Region-level tagging
- Operand class encoding
- Profile-guided optimization
- Off-chip metadata tables (e.g., DEER)

Conclusion

The move to RegLess-style coherence simplifies GPU operand management, reduces energy, and enables more efficient shader execution. Compiler-guided operand staging and region slicing allow hardware to dynamically optimize operand placement without burdening the instruction stream with excessive metadata.

14.3.5 Specialized Units

As shown in *section GPU Hardware Units*, the stages of the OpenGL rendering pipeline and the GPU hardware units that accelerate them as shown in Fig. 14.57:

We now explain how these GPU hardware acceleration units—Geometry Units, Rasterization Units, Texture Mapping Units (TMUs), and Render Output Units (ROPs) — work together with SMs to provide GPU-ISA instructions that accelerate the graphics pipeline illustrated in Fig. 14.58 of section *3D Rendering*.

Figure illustrated in section 3D Rendering

Geometry Units

Function:

```
Raw Vertices & Primitives → Transformed Vertices & Primitives
```

Suppose the GLSL geometry shader looks like this:

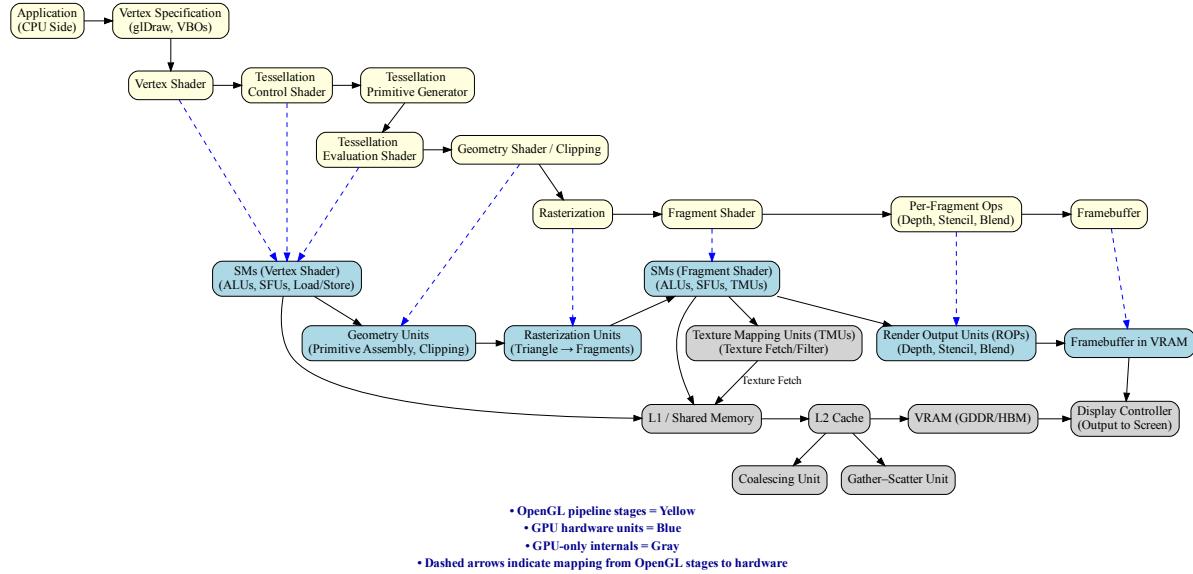
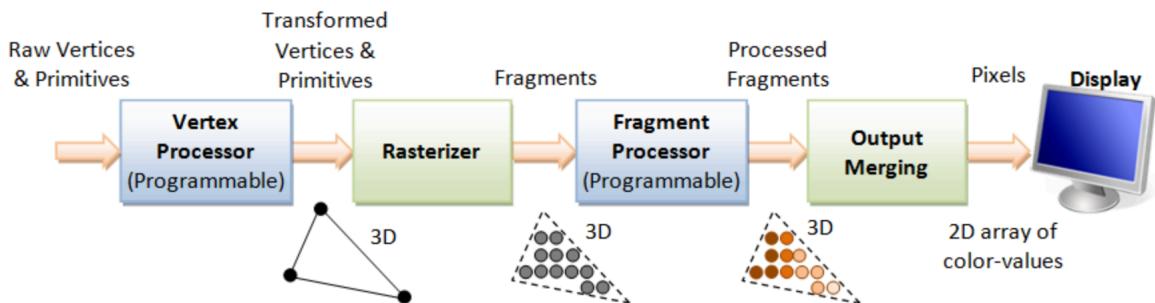


Fig. 14.57: The stages of OpenGL pipeline and GPU's acceleration components



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z) , and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

 Fig. 14.58: 3D Graphics Rendering Pipeline Page 665, 1

An example of GLSL geometry shader

```
#version 450
layout(triangles) in;
layout(line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();

    gl_Position = gl_in[1].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

The corresponding PTX instructions and pipeline flow as Fig. 14.59.

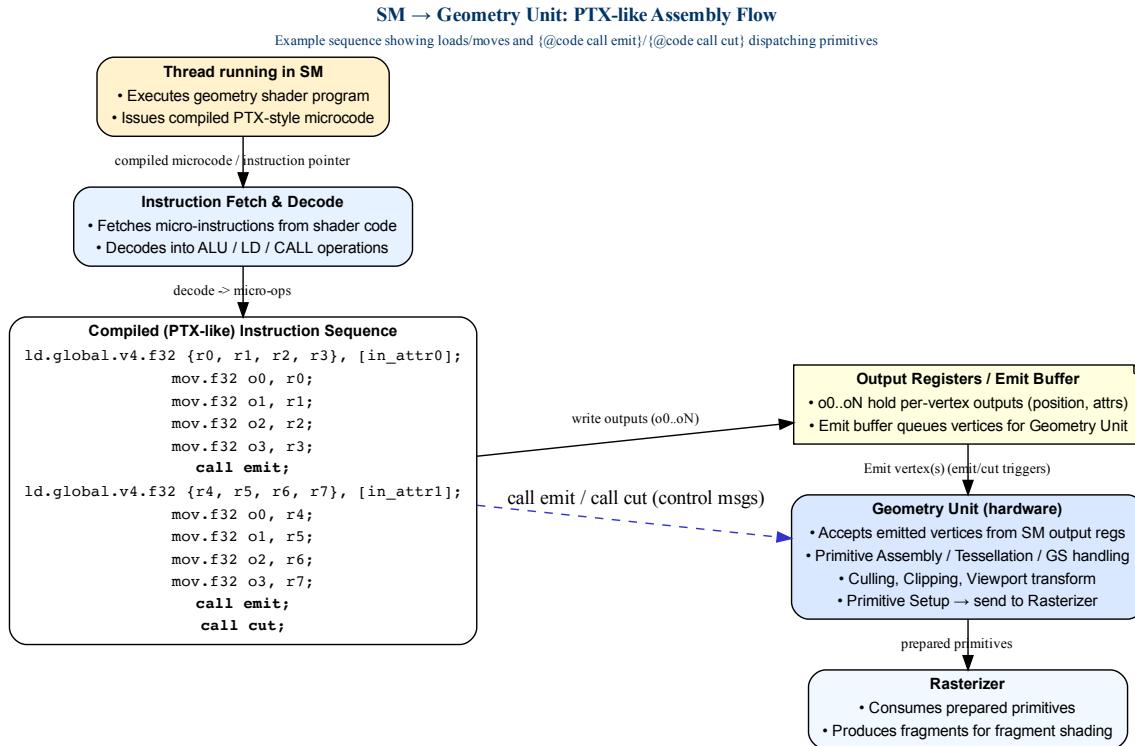


Fig. 14.59: Fetch a sequence of Geometry instructions and pass to Geometry Unit

The **Geometry Unit** in a GPU is a collection of fixed-function and programmable stages responsible for transforming assembled primitives (points, lines, triangles, patches) into screen-space primitives ready for rasterization. The `emit` and `cut` are compiler intrinsics that map to control messages to the Geometry Unit. When we say `emit` and `cut` in NVIDIA PTX (or HLSL/GLSL geometry shaders), they're not ALU instructions that run in the SM like `add` or `mul`. Instead, they act like special control instructions that tell the GPU's fixed-function Geometry Unit what to do with the vertex data currently in the SM's output registers illustrated in Fig. 14.60.

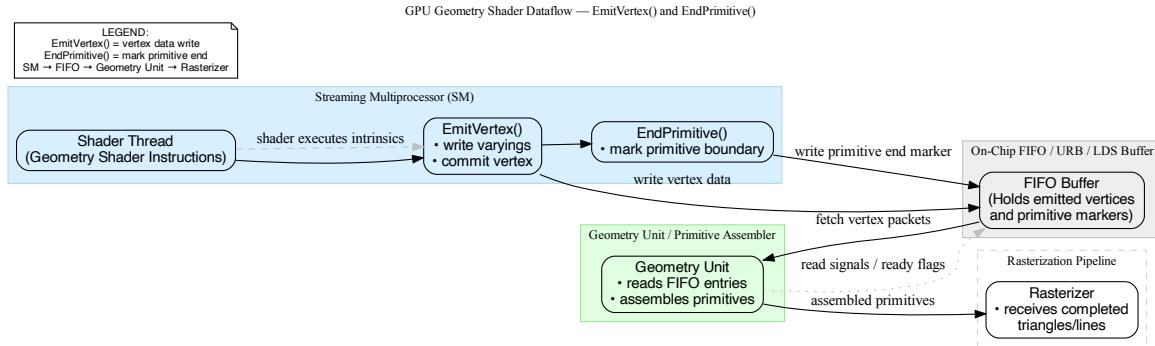


Fig. 14.60: Micro-level flow: SM → Geometry Unit via Emit/Cut

Unlike GLSL textures, which are converted into a specific hardware ISA, the Geometry Shader in Fig. 14.57 maps directly to the Geometry Units instead of the SMs.

Geometry Unit bridges the **vertex shading** stage and the **rasterization** stage as shown in Fig. 14.61.

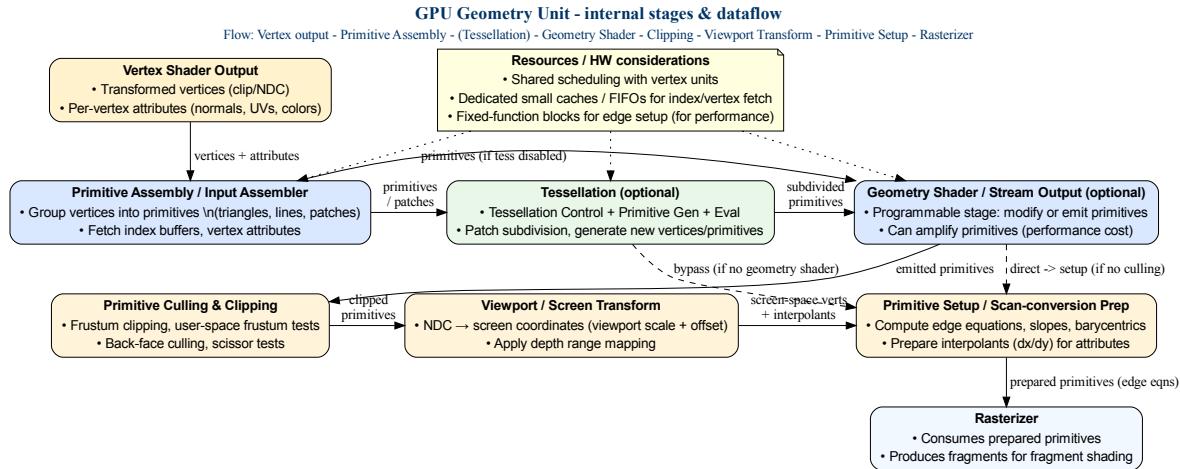


Fig. 14.61: Geometry Unit with its sub-functions (assembly, tessellation, clipping, viewport transform, etc.)

Role

- Organize and process geometry data after vertex shading.
- Perform primitive-level operations such as assembly, tessellation, clipping, viewport transform, and primitive setup.
- Provide hardware acceleration for geometry amplification or reduction before rasterization.

Components

- **Primitive Assembly (Input Assembler)**
 - Groups vertices into primitives (triangles, lines, patches).
 - Fetches indices and vertex attributes from memory.

- Prepares data structures for downstream geometry stages.
- **Tessellation Engine (optional, OpenGL 4.0+ / DirectX 11+)**
 - Subdivides patches into finer primitives.
 - Contains Tessellation Control Shader, Primitive Generator, and Tessellation Evaluation Shader.
 - Used in terrain rendering, displacement mapping, and adaptive LOD.
- **Geometry Shader (optional, programmable stage)**
 - Can generate new primitives or discard existing ones.
 - Enables shadow volume extrusion, point sprite expansion, or procedural geometry.
 - High flexibility but often limited in performance due to amplification.
- **Culling & Clipping**
 - Removes back-facing or out-of-view primitives.
 - Clips primitives against the view frustum or user-defined clipping planes.
 - Optimizes rendering by reducing fragment processing workload.
- **Viewport Transform**
 - Maps Normalized Device Coordinates (NDC) to screen-space pixel coordinates.
 - Applies viewport scaling, offset, and depth range mapping.
- **Primitive Setup**
 - Converts screen-space primitives into edge equations and interpolation rules.
 - Prepares slopes and barycentric coefficients for attribute interpolation in rasterization.
 - Ensures that per-fragment attributes (e.g., texture coordinates, normals) are interpolated correctly.

Usage

- Reduces workload on the fragment stage by culling invisible primitives.
- Provides tessellation and geometry shaders for advanced rendering effects.
- Ensures efficient and accurate rasterization setup.
- Works closely with specialized GPU fixed-function blocks such as **PolyMorph Engines** (NVIDIA) or **Geometry Processors** (AMD).

References

- Wikipedia —[Graphics pipeline](#)
- NVIDIA —[DirectX 11 GPU Architecture \(Geometry and PolyMorph Engine\)](#)
- Intel —[3D Pipeline Overview \(including Geometry Stage\)](#)
- LearnOpenGL —[Geometry Shader](#)
- Microsoft Docs —[Tessellation and Geometry Pipeline](#)

Rasterization Units¹⁰⁹

Function:

¹⁰⁹ copilot: Please provide detailed information about the Rasterization Unit and its pipeline, including a separated dot graph and relevant website references in reStructuredText (reST) format.

Transformed Vertices & Primitives → Fragments

Overview

The rasterization unit is a critical component of the graphics pipeline in modern GPUs. It converts geometric primitives (typically triangles) into fragments that correspond to pixels on the screen. This process is essential for rendering 3D scenes into 2D images.

The pipeline flow for Rasterization Units is shown as Fig. 14.62.

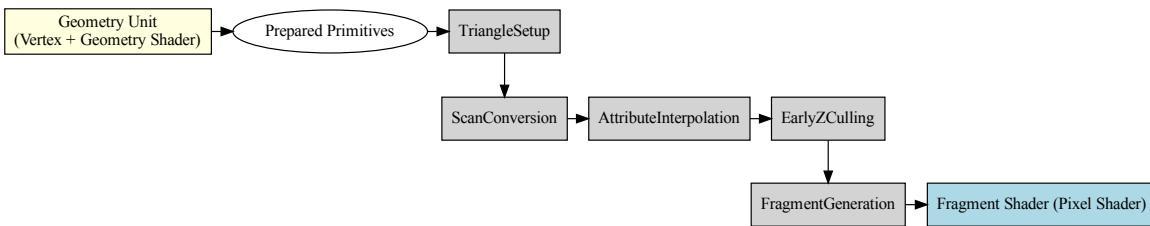


Fig. 14.62: Rasterization pipeline

Key Functions

- **Triangle Setup:** Computes edge equations and bounding boxes for each triangle.
- **Scan Conversion:** Determines which pixels are covered by the triangle.
- **Attribute Interpolation:** Calculates interpolated values (e.g., texture coordinates, depth) for each fragment.
- **Fragment Generation:** Produces fragment data for downstream shading and blending stages.

Hardware Architecture

Modern GPUs implement rasterization in highly parallel hardware blocks to maximize throughput. A simplified block diagram includes:

- **Primitive Assembly Unit:** Groups vertices into triangles.
- **Triangle Setup Engine:** Prepares edge equations and bounding boxes.
- **Rasterizer Core:** Performs scan conversion and fragment generation.
- **Early-Z Unit:** Performs early depth testing to discard hidden fragments.
- **Fragment Queue:** Buffers fragments for shading.

Optimization Techniques

- **Tile-Based Rasterization:** Divides the screen into tiles to reduce memory bandwidth.
- **Early-Z Culling:** Discards fragments before shading if they fail depth tests.
- **Compression:** Reduces data transfer costs between pipeline stages.

Use Cases

- Real-time rendering in games and simulations.
- 3D Gaussian Splatting acceleration for AI-based rendering.
- Mobile GPUs with power-efficient rasterization pipelines.

References

- GauRast: Enhancing GPU Triangle Rasterizers
- NVIDIA Ada GPU Architecture PDF
- Stanford CS248A Lecture on Rasterization

Texture Mapping Units (TMUs)^{Page 727, 66}

Function:

Fragments → Processed Fragments

Overview

A Texture Mapping Unit (TMU) is a fixed-function hardware block inside a GPU responsible for *fetching, filtering, and preparing texture data* that shaders (sampled in fragment or compute stages) use during rendering.

As explained in previous section *OpenGL Shader Compiler*, the texture instruction using TMU to accelerate calculation as the following explanation with Fig. 14.63.

TMUs sit between the shader cores (SMs/CUs) and the memory subsystem. They provide high-performance, specialized texture access operations that would be too slow or costly to emulate in general-purpose ALUs is shown as Fig. 14.63.

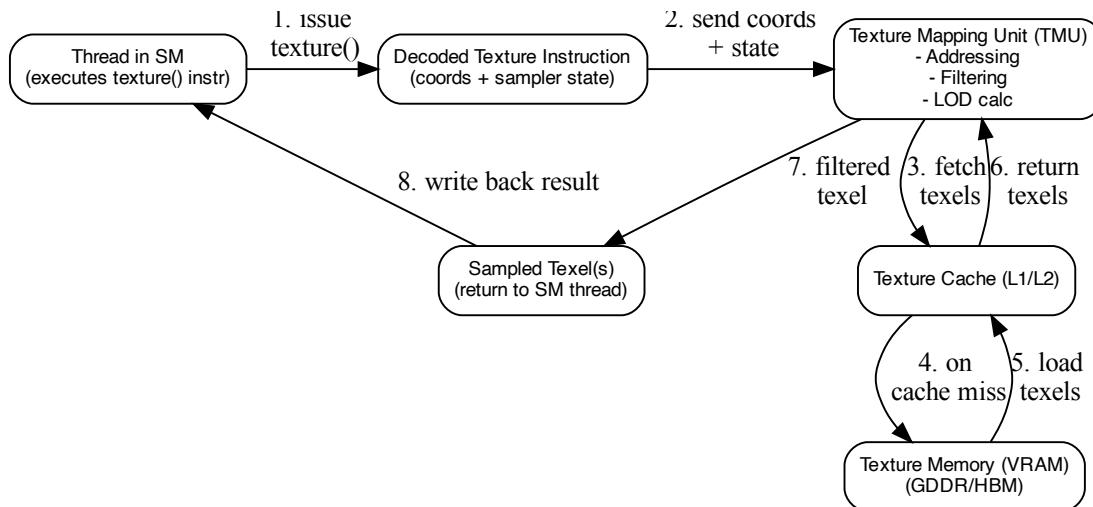


Fig. 14.63: The flow of issuing texture instruction from SM to TMU.

Pipeline Role

- In the **OpenGL / Direct3D graphics pipeline**, TMUs are mainly used in the *fragment shading stage*, where textured surfaces are shaded with data from 2D/3D textures.
- In **compute shaders**, TMUs are also used for image load/store operations and texture sampling.

Key Responsibilities

1. Texture Addressing
 - Compute the correct texture coordinate for a given fragment or pixel.

- Handle the following wrapping modes are shown as Fig. 14.64 and as Fig. 14.65:

Texture coordinates usually range from (0,0) to (1,1) but what happens if we specify coordinates outside this range? OpenGL provides the following wrapping modes for outside this range.

- Clamp-to-border (GL_CLAMP_TO_BORDER)
 - When a texture coordinate falls outside the [0,1] range, the GPU does not sample the nearest texel.
 - Instead, it returns a user-defined border color for that texture.
 - This is useful for effects like shadow maps, where sampling outside the valid area should produce a consistent value.
- Repeat (GL_REPEAT): Wraps coordinates around (tiles the texture).
- Clamp-to-edge (GL_CLAMP_TO_EDGE): Uses the edge texel when coordinates are out of range.
- Mirrored repeat (GL_MIRRORED_REPEAT): Mirrors the texture each repetition.
 - For the middle row ($t(V)$) in the range 0.0 to 1.0), the mirroring operation applies only a left-right swap. For the top and bottom rows, the mirroring includes both left-right and up-down swaps.

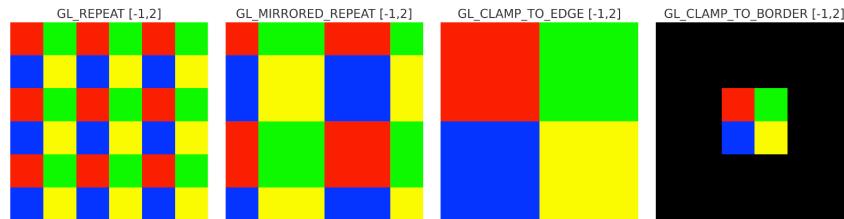


Fig. 14.64: Texture Warpping



Fig. 14.65: Texture Warpping^{Page 731, 71}

- Convert normalized texture coordinates into actual memory addresses.

2. Texture Fetching

- Retrieve texels (texture elements) from texture memory (L1 texture cache, then L2/VRAM on miss).
- Handle different texture layouts: - 1D, 2D, 3D textures - Cubemaps - Texture arrays
- Support compressed texture formats (e.g., DXT, ASTC, ETC2).

3. Texture Filtering

Given a Texture coordinates, OpenGL has to figure out which **texture pixel (also known as a texel)** to map the texture coordinate to.

- Perform *interpolation* between texels to produce smooth visual results.
- Filtering requires multiple texel reads + weighted average calculations.

- Common filtering modes as the following are shown as Fig. 14.68:
 - Nearest-neighbor (point sampling) (GL_NEAREST)
 - * When set to GL_NEAREST, OpenGL selects the color of the texel that center is closest to the texture coordinate shown as the example in Fig. 14.66. ‘+’ is the coordinates of texel. ‘Returns’ is the color of result.

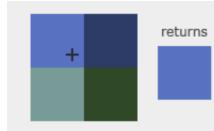


Fig. 14.66: GL_NEAREST^{Page 731, 71}

- Bilinear (GL_LINEAR)
 - * The return color is the mix of four neighboring pixels. The smaller the distance from the texture coordinate to a texel’s center, the more that texel’s color contributes to the sampled color shown as the example in Fig. 14.67.

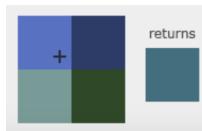


Fig. 14.67: GL_LINEAR^{Page 731, 71}

- Trilinear (with mipmaps)
- Anisotropic filtering (for angled surfaces)
- Let’s see how these methods work when using a texture with a low resolution on a large object (texture is therefore scaled upwards and individual texels are noticeable). The GL_NEAREST and GL_LINEAR as the following Fig. 14.68. As result, GL_LINEAR produces a more blurred color and smooth edge’s output.

4. Mipmap Level of Detail (LOD) Selection

- Choose the correct mipmap level based on screen-space derivatives of texture coordinates.
- Prevent aliasing and improve cache efficiency.
- Optionally blend between mip levels for trilinear filtering.

5. Texture Caching

- TMUs have a **dedicated texture cache** optimized for 2D/3D spatial locality.
- Neighboring threads in a Warp often fetch adjacent texels, improving cache hits.
- Caches reduce memory latency and improve bandwidth utilization.

6. Specialized Operations

- Texture gather: fetch 4 neighboring texels around a coordinate.
- Shadow mapping: compare fetched depth texel against reference value.
- Multisample textures: fetch per-sample data for MSAA.
- Border color application for out-of-bounds accesses.



Fig. 14.68: Texture Filter: GL_NEAREST has sharp color and jagged edge Page 731, 71

Microarchitecture Aspects

- Each **Streaming Multiprocessor (SM)** or **Compute Unit (CU)** is paired with several TMUs.
- The number of TMUs is a key spec in GPU datasheets (e.g., “64 TMUs”).
- TMU throughput is often measured in **texels per clock cycle**.
- Modern GPUs balance **TMUs per ALU** to ensure shading and texture workloads are not bottlenecked.

Performance Considerations

- **Bandwidth-limited:** TMUs rely heavily on memory bandwidth. Mipmapping and caches reduce this pressure.
- **Latency hiding:** texture fetches may take hundreds of cycles, so GPUs rely on massive multithreading to hide stalls.
- **Workload dependent:** texture-heavy games or rendering pipelines are often limited by TMU throughput.

Summary

TMUs are highly specialized GPU units that:

- Translate texture coordinates into addresses.
- Fetch texels efficiently with dedicated caches.
- Perform filtering and LOD computations in hardware.
- Deliver high throughput for texture operations that are essential in realistic rendering.

Without TMUs, all these operations would fall on general-purpose ALUs, resulting in drastically lower performance and efficiency.

Render Output Units (ROPs)¹¹⁰

Function:

Processed Fragments → Pixels

Overview

Render Output Units (ROPs), also known as Raster Operations Pipelines, are the final stage in the GPU graphics pipeline before pixel data is written to the framebuffer. ROPs handle pixel-level operations such as blending, depth and stencil testing, multisample resolve, and writing to memory. They are crucial for assembling the final image that appears on screen.

Pipeline Responsibilities

- **Fragment Reception:** Accepts shaded fragments from the pixel shader.
- **Depth and Stencil Testing:** Compares fragment depth/stencil values against buffers.
- **Blending:** Combines fragment color with existing framebuffer data.
- **Multisample Resolve:** Merges multiple samples into a final pixel (for MSAA).
- **Framebuffer Write:** Commits final pixel data to memory for display.

The pipeline flow is shown as Fig. 14.69.

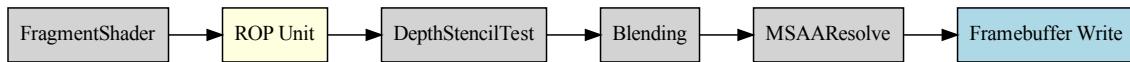


Fig. 14.69: The pipeline for Render Output Units (ROPs)

Performance Considerations

- **ROP Count:** More ROPs can increase pixel throughput, especially at high resolutions.
- **Memory Bandwidth:** ROPs are tightly coupled with memory controllers; bandwidth limits can bottleneck performance.
- **Antialiasing Support:** Hardware MSAA and resolve operations are often implemented in ROPs.
- **Compression:** Some GPUs use framebuffer compression to reduce bandwidth usage.

Vendor-Specific Notes

- **NVIDIA:** Refers to these units as ROPs; tightly integrated with memory partitions.
- **AMD:** Calls them Render Backends (RBs); RDNA architecture decouples ROPs from shader engines.
- **Intel & ARM:** Implement simplified ROPs for power-efficient mobile rendering.

References

- [Render Output Unit - Wikipedia](#)
- [What is a ROP on a GPU? - CORSAIR](#)
- [TechPowerUp Forums: ROPs and TMUs](#)

¹¹⁰ copilot: Please provide detailed information about the Render Output Units (ROPs) and its pipeline, including a dot graph and relevant website references in reStructuredText (reST) format.

14.3.6 System Features —Buffers

CPU and GPU provides different Buffers to speedup OpenGL pipeline rendering⁴⁷.

Table 14.10: Graphics Buffers

Buffer Type	Access	Location	API/Usage	Function	Description
Vertex Buffer (VBO)	Read	GPU	OpenGL, Vulkan	Store vertex attributes	Holds data like position, normal, and texture coords for drawing geometry.
Index Buffer (IBO/EBO)	Read	GPU	OpenGL, Vulkan	Reuse vertex data	Stores indices into the vertex buffer to avoid duplication.
Uniform Buffer (UBO)	Read	GPU or Shared	OpenGL, Vulkan	Constant input data	Shares transformation matrices, lighting, or material data across shaders.
Shader Storage Buffer (SSBO)	Read/Writ	GPU or Shared	OpenGL, Vulkan	General data exchange	Flexible, large buffers accessible for structured shader I/O.
Constant Buffer	Read	GPU or Shared	DirectX, Vulkan	Fast uniform access	Optimized for fast access to frequently read small data.
Image / Texture Buffer	Read/Writ	GPU	OpenGL, Vulkan	Sample/store pixels	Stores image data for sampling or read/write image operations in shaders.
Color Buffer	Write	GPU	OpenGL, Vulkan	Store final pixel color	Stores output of fragment shaders; used for display or post-processing.
Depth Buffer (Z-Buffer)	Write/Rea	GPU	OpenGL, Vulkan	Visibility testing	Stores per-pixel depth values for hidden surface removal.
Frame Buffer	Write	GPU	OpenGL, Vulkan	Store render output	Holds final color, depth, or other rendered output.
Stencil Buffer	Read/Writ	GPU	OpenGL, Vulkan	Pixel masking	Used to conditionally discard or preserve pixels in the pipeline.

- Color buffer

They contain the RGB or sRGB color data and may also contain alpha values for each pixel in the framebuffer. There may be multiple color buffers in a framebuffer. You've already used double buffering for animation. Double buffering is done by making the main color buffer have two parts: a front buffer that's displayed in your window; and a back buffer, which is where you render the new image⁸³.

- Depth buffer (Z buffer)

⁴⁷ Page 155 - 185 of book “OpenGL Programming Guide 9th Edition”[Page 692, 36](#).

⁸³ Page 155 of book “OpenGL Programming Guide 9th Edition”[Page 692, 36](#).

Depth is measured in terms of distance to the eye, so pixels with larger depth-buffer values are overwritten by pixels with smaller values⁸⁴⁸⁶⁸⁷.

- Frame Buffer

OpenGL offers: the color, depth and stencil buffers. This combination of buffers is known as the default framebuffer and as you've seen, a framebuffer is an area in memory that can be rendered to⁸⁹.

- Stencil Buffer

In the simplest case, the stencil buffer is used to limit the area of rendering (stenciling)⁸⁸⁸⁷.

Table 14.11: Compute Buffers

Buffer Type	Access	Location	API/Usage	Function	Description
Compute Buffer	Read/Writ	GPU or Shared	OpenCL, Vulkan, CUDA	Parallel compute data	Buffers used in compute kernels or shaders for general processing.
Atomic Buffer	Read/Writ (Atomic)	GPU	OpenGL, Vulkan	Shared counters/data	Used with atomic ops for synchronization or accumulation.
Acceleration Structure Buffer	Read	GPU	Vulkan RT, DXR	Ray tracing acceleration	Holds spatial hierarchy (BVH) for ray traversal efficiency.
Indirect Draw Buffer	Read	GPU	Vulkan, DirectX	GPU-issued draw	Stores draw/dispatch args to issue commands without CPU.

- DXR: DirectX Raytracing —a D3D12 extension for real-time ray tracing using GPU acceleration.
- Indirect Draw Buffer: A GPU-side buffer holding draw parameters so that GPU (not CPU) can issue rendering work dynamically.

Table 14.12: System-Level and Utility Buffers

Buffer Type	Access	Location	API/Usage	Function	Description
Command Buffer	Write (CPU) / Read (GPU)	Host → GPU	Vulkan, DirectX12	Submit work	Encapsulates commands like draw, dispatch, and memory ops.
Parking / Staging Buffer	Read/Writ	Host-visible	Vulkan, CUDA	Temporary transfer	Temporary CPU-visible buffer for uploading/downloading GPU data.

⁸⁴ Page 156 of book “OpenGL Programming Guide 9th Edition”⁸⁵ Page 692, 36.

⁸⁶ <https://en.wikipedia.org/wiki/Z-buffering>

⁸⁷ <https://open.gl/depthstencils>

⁸⁸ <https://open.gl/framebuffers>

⁸⁹ https://en.wikipedia.org/wiki/Stencil_buffer

14.4 Software Structure

As the previous section illustrated, GPU is a SIMD (SIMD) for data parallel application. This section introduces the GPU evolved from Graphics GPU to the General purpose GPU (GPGPU) and the software architecture of GPUs and explores AI software frameworks designed for GPUs, NPUs, and CPUs.

14.4.1 General purpose GPU

Since GLSL shaders provide a general way for writing C code in them, if applying a software frame work instead of OpenGL API, then the system can run some data parallel computation on GPU for speeding up and even get CPU and GPU executing simultaneously. Furthermore, any language that allows the code running on the CPU to poll a GPU shader for return values, can create a GPGPU framework⁹⁰.

Mapping data in GPU

As described in the previous section on GPUs, the subset of the array calculation $y[] = a * x[] + y[]$ is shown as follows:

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- name<<<dimGrid, dimBlock>>>(...parameter list ...):
 - dimGrid: Number of Blocks in Grid
 - dimBlock: 256 Threads in Block

Assembly code of PTX (from page 300 of Quantitative book)

```
// code to set VLR, Vector Length Register, to (n % 256)
// ...
//
shl.u32 R8, blockIdx, 9          ; Thread Block ID * Block size (512)
add.u32 R8, R8, threadIdx        ; R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3               ; byte offset
setp.neq.s32 P1, RD8, RD3       ; RD3 = n, P1 is predicate register 1
ld.global.f64 RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4           ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2            ; SuminRD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0        ; Y[i] = sum (X[i]*a + Y[i])
```

- Need to set VLR if PTX has this instruction. Otherwise, set lane-mask in the similar way of the code below.

```
__device__
void lane-mask-ex( double *X, double *Y, double *Z) {
```

(continues on next page)

⁹⁰ https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

(continued from previous page)

```

if (X[i] != 0)
    X[i] = X[i] - Y[i];
else X[i] = Z[i];
}

```

Assembly code of Vector Processor

```

LV V1,Rx      ;load vector X into V1
LV V2,Ry      ;load vector Y
L.D F0,#0     ;load FP zero into F0
SNEVS.D V1,F0 ;sets VM(i) to 1 if V1(i) !=F0
SUBVV.D V1,V1,V2 ;subtract under vector mask
SV V1,Rx      ;store the result in X

```

Assembly code of PTX (modified code from referring page 208 - 302 of Quantitative book)

```

ld.global.f64 RD0, [X+R9]      ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0       ; P1 is predicate register 1
@!P1, bra ELSE1, *Push        ; Push old mask, set new mask bits
                                ; if P1 false, go to ELSE1
ld.global.f64 RD2, [Y+R8]      ; RD2 = Y[i]
sub.f64 RD0, RD0, RD2         ; Difference in RD0
st.global.f64 [X+R8], RD0      ; X[i]=RD0
ELSE1:
ld.global.f64 RD0, [Z+R8]      ; RD0 = Z[i]
st.global.f64 [X+R8], RD0      ; X[i] = RD0
ENDIF1:
ret, *Pop                      ; pop to restore old mask

```

- For Lane Mask, refer to¹⁰³¹⁰⁴.

The following table explains how the elements of *saxpy()* are mapped to the Lanes of a SIMD Thread (Warp), which belongs to a Thread Block (Core) within a Grid.

Table 14.13: Mapping *saxpy* code to Fig. 14.52.

saxpy()	Instance in Fig. 14.52	Description
block-Dim.x	The index of Thread Block	blockDim: in this example configured as Fig. 14.52 is 16(Thread Blocks) * 16(SIMD Threads) = 256
block-Idx.x	The index of SIMD Thread	blockIdx: the index of Thread Block within the Grid
thread-dIdx.x	The index of elements	threadIdx: the index of the SIMD Thread within its Thread Block

- With Fermi, each 32-wide thread of SIMD instructions is mapped to 16 physical SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete.
- You could say that it has 16 Lanes, the vector length would be 32, and the Chime is 2 clock cycles.

¹⁰³ subsection Vector Mask Registers: Handling IF Statements in Vector Loops of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

¹⁰⁴ Code written by referring page 208 - 302 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

- The mape of $y[0..31] = a * x[0..31] * y[0..31]$ to $\langle\text{Core, Warp, Cuda Thread}\rangle$ of GPU as the following table. $x[0..31]$ map to 32 Cuda Threads; **two Cuda Threads map to one SIMD Lane** as Fig. 14.51..

 Table 14.14: Map $\langle\text{Core, Warp}\rangle$ to saxpy

	Warp-0	Warp-1	...	Warp-15
Core-0	$y[0..31] = a * x[0..31] * y[0..31]$	$y[32..63] = a * x[32..63] + y[32..63]$...	$y[480..511] = a * x[480..511] + y[480..511]$
...
Core-15	$y[7680..7711] = a * \dots$	$y[8160..8191] = a * x[8160..8191] + y[8160..8191]$

- Each Cuda Thread runs the GPU function code *saxpy*. Fermi has a register file of size 32768 x 32-bit. As shown in Fig. 14.43, the number of registers in a Thread Block is: 16 (SM) * 32 (Cuda Threads) * 64 (TLR, Thread Level Register) = 32768 x 32-bit (Register file).
- When mapping to fragments/pixels in a graphics GPU, $x[0..15]$ corresponds to a two-dimensional tile of fragments/pixels at $pixel[0..3]/[0..3]$, since images use tile-based grouping to cluster similar colors together.

Work between CPU and GPU in Cuda

The previous *daxpy()* GPU code did not include the host (CPU) side code that triggers the GPU function.

The following example shows the host (CPU) side of a CUDA program that calls *saxpy* on the GPU⁹³:

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float * x, float * y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    ...
    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
    ...
    saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
    ...
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
    ...
}
```

The *main()* function runs on the CPU, while *saxpy()* runs on the GPU. The CPU copies data from *x* and *y* to the corresponding device arrays *d_x* and *d_y* using *cudaMemcpy*.

The *saxpy* kernel is launched with the following statement:

```
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
```

⁹³ <https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/>

This launches the kernel with Thread Blocks containing 256 threads, and uses integer arithmetic to determine the number of Thread Blocks needed to process all N elements in the arrays. The expression $(N+255)/256$ ensures full coverage of the input data.

Using *cudaMemcpyHostToDevice* and *cudaMemcpyDeviceToHost*, the CPU can pass data in x and y to the GPU, and retrieve the results back to y .

Since both memory transfers are handled by DMA and do not require CPU operation, the performance can be improved by running CPU and GPU independently, each accessing their own cache.

After the DMA copy from CPU memory to GPU memory, the GPU performs the full matrix operation loop for $y[j] = a * x[j] + y[j]$; using a single Grid of threads.

DMA *memcpy* maps the data in CPU memory to each L1 cache of a core on GPU memory.

Many GPUs support scatter and gather operations to access DRAM efficiently for stream processing tasks¹⁰⁵[Page 771, 90](#)¹⁰⁶.

When the GPU function is dense computation in array such as MPEG4 encoder or deep learning for tuning weights, it may get much speed up¹⁰⁷. However when GPU function is matrix addition and CPU will idle for waiting GPU's result. It may slow down than doing matrix addition by CPU only. Arithmetic intensity is defined as the number of operations performed per word of memory transferred. It is important for GPGPU applications to have high arithmetic intensity else the memory access latency will limit computational speedup^{Page 771, 90}.

Wiki here¹⁰⁸ includes GPU-accelerated applications for speedup as follows:

General Purpose Computing on GPU, has found its way into fields as diverse as machine learning, oil exploration, scientific image processing, linear algebra, statistics, 3D reconstruction and even stock options pricing determination. In addition, section “GPU accelerated video decoding and encoding” for video compressing¹⁰⁸ gives the more applications for GPU acceleration.

Table 14.15: The differences for speedup in architecture of CPU and GPU

Item	CPU	GPU
Application	Non-data parallel	Data parallel
Architecture	SISD, small vector (eg.4*32bits)	Large SIMD (eg.16*32bits)
Cache	Smaller and faster	Larger and slower (ref. The following Note)
ILP	Pipeline	Pipeline
•	Superscalar, SMT	SIMT
•	Super-pipeline	
Core	Smaller threads for SMT (2 or 4)	Larger threads (16 or 32)
Branch	Conditional-instructions	Mask & conditional-instructions

Note

GPU-Cache

In theory, for data-parallel applications using GPU's SMT, the GPU can schedule more threads and aims for throughput rather than speedup of a single thread, as seen in SISD on CPUs.

However, in practice, GPUs provide only a small L1 cache, similar to CPUs, and handle cache misses by scheduling another thread.

¹⁰⁵ Reference “Gather-Scatter: Handling Sparse Matrices in Vector Architectures”: section 4.2 Vector Architecture of A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

¹⁰⁶ The whole chip shares a single L2 cache, but the different units will have individual L1 caches. <https://computergraphics.stackexchange.com/questions/355/how-does-texture-cache-work-considering-multiple-shader-units>

¹⁰⁷ <https://www.manchestervideo.com/2016/06/11/speed-h-264-encoding-budget-gpu/>

¹⁰⁸ https://en.wikipedia.org/wiki/Graphics_processing_unit

As a result, GPUs often lack L2 and L3 caches, which are common in CPUs with deeper cache hierarchies.

14.4.2 OpenCL, Vulkan and spir-v

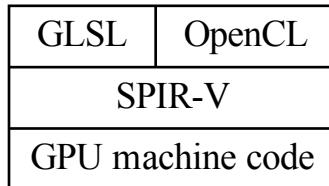


Fig. 14.70: OpenCL and GLSL(OpenGL)

Table 14.16: OpenCL and OpenGL SW system

Name of SW	GPU language	Level of GPU language
OpenCL	OpenCL	C99 dialect (with C pointer, ...)
OpenGL	GLSL	C-like (no C pointer, ...)
Vulkan	SPIR-V	IR

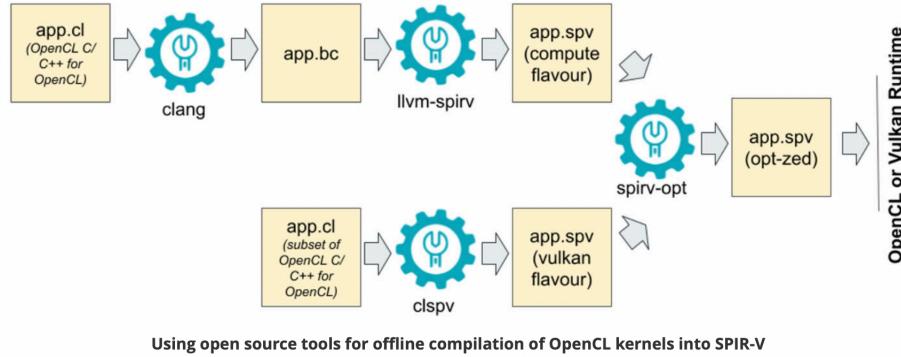


Fig. 14.71: Offline Compilation of OpenCL Kernels into SPIR-V Using Open Source Tooling¹¹²

- clang: Compile OpenCL to spirv for runtime+driver. Or compile OpenCL to llvm, then “SPIR-V LLVM Translator” translate llvm to spirv for runtime+driver.
- clspv: Compile OpenCL to spirv directly.

The flow and relationships among GLSL, OpenCL, SPIR-V (Vulkan/OpenCL), LLVM IR, and the GPU compiler are shown in the Fig. 14.70, Fig. 14.71 and Fig. 14.72. As shown in Fig. 14.72, OpenCL-C to SPIR-V (OpenCL) can be compiled using **clang + llvm-spirv** tools or a proprietary converter.

¹¹² <https://www.khronos.org/blog/offline-compilation-of-opencl-kernels-into-spir-v-using-open-source-tooling>



Fig. 14.72: GPU Compiler Components and Flow

As shown in Fig. 14.72, both GLSL and OpenCL use frontend tools to generate SPIR-V. The driver can invoke either the GLSL or OpenCL compiler based on metadata fields in the SPIR-V, as illustrated in Fig. 14.73 and the following figures, which describe offline compilation from GLSL/OpenCL to SPIR-V and online execution using the generated SPIR-V files.

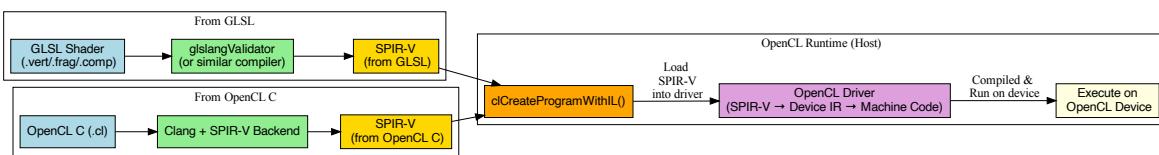


Fig. 14.73: Compiling and Deploying GPU Code from GLSL, Vulkan, and OpenCL

Based on the flows above, the public standards OpenGL and OpenCL provide tools for transferring these data format, as illustrated in Fig. 14.74. The corresponding LLVM IR and SPIR-V formats are listed below.

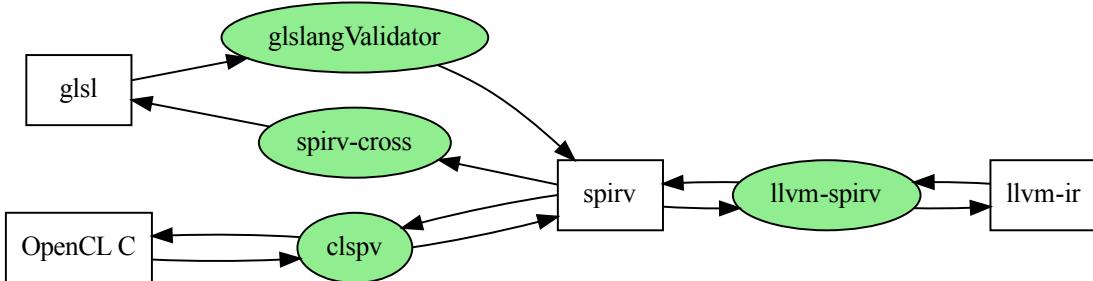


Fig. 14.74: Conversion between GLSL, OpenCL C, SPIRV-V and LLVM-IR

References/add-matrix.ll

```

; ModuleID = 'add-matrix.ll'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
                   ↪f32:32:32-f64:64:64-v16:16:16-v24:32:32-v32:32:32-v48:64:64-v64:64:64-v96:128:128-
                   ↪v128:128:128-v192:256:256-v256:256:256-v512:512:512-v1024:1024:1024-G1"
target triple = "spir64-unknown-unknown"

; Function Attrs: nounwind

```

(continues on next page)

(continued from previous page)

```
define spir_func <4 x i32> @add_mat(<4 x i32> %a, <4 x i32> %b) #0 {
entry:
    %sum = add <4 x i32> %a, %b
    ret <4 x i32> %sum
}

attributes #0 = { nounwind }

!spirv.MemoryModel = !{!0}
!opencl.enable.FP_CONTRACT = !{}
!spirv.Source = !{!1}
!opencl.spir.version = !{!2}
!opencl.used.extensions = !{!3}
!opencl.used.optional.core.features = !{!3}
!spirv.Generator = !{!4}

!0 = !{i32 2, i32 2}
!1 = !{i32 0, i32 0}
!2 = !{i32 1, i32 2}
!3 = !{}
!4 = !{i16 6, i16 14}
```

References/add-matrix.spvasm

```
; SPIR-V
; Version: 1.0
; Generator: Khronos LLVM/SPIR-V Translator; 14
; Bound: 10
; Schema: 0
        OpCapability Addresses
        OpCapability Linkage
        OpCapability Kernel
%1 = OpExtInstImport "OpenCL.std"
        OpMemoryModel Physical64 OpenCL
        OpSource Unknown 0
        OpName %add_mat "add_mat"
        OpName %a "a"
        OpName %b "b"
        OpName %entry "entry"
        OpName %sum "sum"
        OpDecorate %add_mat LinkageAttributes "add_mat" Export
%uint = OpTypeInt 32 0
%v4uint = OpTypeVector %uint 4
%4 = OpTypeFunction %v4uint %v4uint %v4uint
%add_mat = OpFunction %v4uint None %4
        %a = OpFunctionParameter %v4uint
        %b = OpFunctionParameter %v4uint
%entry = OpLabel
        %sum = OpIAdd %v4uint %a %b
            OpReturnValue %sum
        OpFunctionEnd
```

Convert between spirv and llvm-ir

```
% pwd
$HOME/git/lbd/References
% llvm-as -o add-matrix.bc add-matrix.ll
% llvm-spirv -o add-matrix.spv add-matrix.bc
% spirv-dis -o add-matrix.spvasm add-matrix.spv
// Convert spirv to llvm-ir again and check the converted llvm-ir is same
// with origin.
% llvm-spirv -r add-matrix.spv -o add-matrix.spv.bc
% llvm-dis add-matrix.spv.bc -o add-matrix.spv.bc.ll
% diff add-matrix.ll add-matrix.spv.bc.ll
1c1
< ; ModuleID = 'add-matrix.ll'
---
> ; ModuleID = 'add-matrix.spv.bc'
```

Install llvm-spirv and llvm with Brew-install

```
% brew install spirv-llvm-translator
% brew install llvm
```

The following explains how the driver identifies whether the SPIR-V source is from GLSL or OpenCL.

SPIR-V binaries contain metadata that can help identify whether they were generated from OpenCL, GLSL, or another language.

- Execution Model

Defined by the *OpEntryPoint* instruction. It is a strong indicator of the source language.

ExecutionModel	Typical Source	Notes
Kernel	OpenCL	Used only by OpenCL C
GLCompute	GLSL or HLSL	Used in compute shaders
Fragment	GLSL or HLSL	For pixel shaders
Vertex	GLSL or HLSL	For vertex shaders

- Capabilities

Declared using *OpCapability*. They provide clues about the SPIR-V's execution model and source.

Capability	Likely Source
Kernel	OpenCL
Addresses	OpenCL
Linkage	OpenCL
Shader	GLSL or HLSL

- Extensions

Declared using *OpExtension*. Some are tied to specific compilers or languages.

Extension	Likely Source
SPV_KHR_no_integer_wrap_decoration	OpenCL
SPV_INTEL_unified_shared_memory	OpenCL (Intel)
SPV_AMD_shader_ballot	GLSL (graphics)

- Memory Model

Defined by *OpMemoryModel*.

- *OpenCL* → OpenCL source
- *GLSL450* → GLSL or HLSL source

- How to Inspect

Use the *spirv-dis* tool to disassemble SPIR-V to human-readable form:

```
spirv-dis kernel.spirv -o kernel.spvasm
```

Look for these at the top of the file:

Example (GLSL):

```
OpCapability Shader
OpMemoryModel Logical GLSL450
OpEntryPoint GLCompute %main "main"
```

Example (OpenCL):

```
OpCapability Kernel
OpCapability Addresses
OpMemoryModel Logical OpenCL
OpEntryPoint Kernel %foo "foo"
```

Summary

Feature	Indicates
OpEntryPoint Kernel	OpenCL
OpCapability Shader	GLSL or HLSL
OpMemoryModel OpenCL	OpenCL
OpMemoryModel GLSL450	GLSL or HLSL

- Comparison for OpenCL and OpenGL's compute shader.

- Same:

Both are for General Computing of GPU.

- Difference:

OpenCL include GPU and other accelerate device/processor. OpenCL is C language on Device and C++ on Host based on OpenCL runtime. Compute shader is GLSL shader language run on OpenGL graphic environment and integrate and access data of OpenGL API easily¹¹¹.

- OpenGL/GLSL vs Vulkan/spir-v.

¹¹¹ <https://stackoverflow.com/questions/15868498/what-is-the-difference-between-opencl-and-opengl-compute-shader>

- High level of API and shader: OpenGL, GLSL.
- Low level of API and shader: Vulkan, spir-v.

Though OpenGL api existed in higher level with many advantages from sections above, sometimes it cannot compete in efficiency with direct3D providing lower levels api for operating memory by user program¹¹³. Vulkan api is lower level's C/C++ api to fill the gap allowing user program to do these things in OpenGL to compete against Microsoft direct3D. Here is an example¹¹⁴. Meanwhile glsl is C-like language. The vulkan infrastructure provides tool, glslangValidator¹¹⁵, to compile glsl into an Intermediate Representation Form (IR) called spir-v off-line. As a result, it saves part of compilation time from glsl to gpu instructions on-line since spir-v is an IR of level closing to llvm IR¹¹⁶. In addition, vulkan api reduces gpu drivers efforts in optimization and code generation¹¹³. These standards provide user programmer option to using vulkan/spir-v instead of OpenGL/glsl, and allow them pre-compiling glsl into spir-v off-line to saving part of on-line compilation time.

With vulkan and spir-v standard, the gpu can be used in OpenCL for Parallel Programming of Heterogeneous Systems¹¹⁷¹¹⁸. Similar with Cuda, a OpenCL example for fast Fourier transform (FFT) is here¹¹⁹. Once OpenCL grows into a popular standard when more computer languages or framework supporting OpenCL language, GPU will take more jobs from CPU¹²⁰.

Most GPUs have 16 or 32 Lanes in a SIMD processor (Warp), vulkan provides Subgroup operations to data parallel programming on Lanes of SIMD processor¹²¹.

Subgroup operations provide a fast way for moving data between Lanes intra Warp. Assuming each Warp has eight Lanes. The following table lists result of reduce, inclusive and exclusive operations.

Table 14.17: Lists each Lane's value after **Reduce**, **Inclusive** and **Exclusive** operations repectively

Lane	0	1	2	3
Initial value	a	b	c	d
Reduce	OP(abcd)	OP(abcd)	OP(abcd)	OP(abcd)
Inclusive	OP(a)	OP(ab)	OP(abc)	OP(abcd)
Exclusive	not define	OP(a)	OP(ab)	OP(abc)

- Reduce: e.g. subgroupAdd. Inclusive: e.g. subgroupInclusiveAdd. Exclusive: e.g. subgroupExclusiveAdd.
- For examples:
 - ADD operation: $OP(abcd) = a+b+c+d$.
 - MAX operation: $OP(abc) = \text{MAX}(a,b,c)$.
- When Lane i is inactive, its value is none.
 - For instance of Lane 0 is inactive, then MUL operation: $OP(abcd) = b*c*d$.

The following is a code example.

¹¹³ Vulkan offers lower overhead, more direct control over the GPU, and lower CPU usage...By allowing shader pre-compilation, application initialization speed is improved...A Vulkan driver only needs to do GPU specific optimization and code generation, resulting in easier driver maintenance ...Page 677, 10 https://en.wikipedia.org/wiki/Vulkan#OpenGL_vs._Vulkan

¹¹⁴ <https://github.com/SaschaWillems/Vulkan/blob/master/examples/triangle/triangle.cpp>

¹¹⁵ glslangValidator is the tool used to compile GLSL shaders into SPIR-V, Vulkan's shader format. https://vulkan.lunarg.com/doc/sdk/latest/windows/spirv_toolchain.html

¹¹⁶ SPIR 2.0: LLVM IR version 3.4. SPIR-V 1.X: 100% Khronos defined Round-trip lossless conversion to llvm. https://en.wikipedia.org/wiki/Standard_Portable_Intermediate_Representation

¹¹⁷ <https://www.khronos.org/opencl/>

¹¹⁸ https://en.wikipedia.org/wiki/Compute_kernel

¹¹⁹ <https://en.wikipedia.org/wiki/OpenCL>

¹²⁰ The OpenCL standard defines host APIs for C and C++; third-party APIs exist for other programming languages and platforms such as Python,[15] Java, Perl[15] and .NET.[11]:15 <https://en.wikipedia.org/wiki/OpenCL>

¹²¹ <https://www.khronos.org/blog/vulkan-subgroup-tutorial>

An example of subgroup operations in glsl for vulkan

```

vec4 sum = vec4(0, 0, 0, 0);
if (gl_SubgroupInvocationID < 16u) {
    sum += subgroupAdd(in[gl_SubgroupInvocationID]);
}
else {
    sum += subgroupInclusiveMul(in[gl_SubgroupInvocationID]);
}
subgroupMemoryBarrier();

```

- Nvidia's GPU provides `__syncWarp()` for `subgroupMemoryBarrier()` or compiler to sync for the Lanes in the same Warp.

In order to let Lanes in the same SIMD processor work efficiently, data uniformity analysis will provide many optimization opportunities in register allocation, transformation and code generation¹²².

LLVM IR expansion from CPU to GPU is becoming increasingly influential. In fact, LLVM IR has been expanding steadily from version 3.1 until now, as I have observed.

14.4.3 Unified IR Conversion Flows

Graphics and OpenCL Compilation

This section outlines the intermediate representation (IR) flows for graphics (Microsoft DirectX, OpenGL) and OpenCL compilation across major GPU vendors: NVIDIA, AMD, ARM, Imagination Technologies, and Apple.

Graphics Compilation Flow (Microsoft DirectX & OpenGL)

Graphics shaders are compiled from high-level languages (HLSL, GLSL) into vendor-specific GPU binaries via intermediate representations like DXIL and SPIR-V.

Each node in the graph is color-coded to indicate its category or role within the structure.



- **Vendor Driver** will call **Vendor Compiler** for on-line compilation.
- OpenCL C is the device side code in C language while Host side code is C/C++.
- OpenCL C is compiled to SPIR-V in later versions of OpenCL, while earlier versions used SPIR. SPIR-V has now largely replaced SPIR as the standard intermediate representation.

¹²² <https://llvm.org/docs/ConvergenceAndUniformity.html>

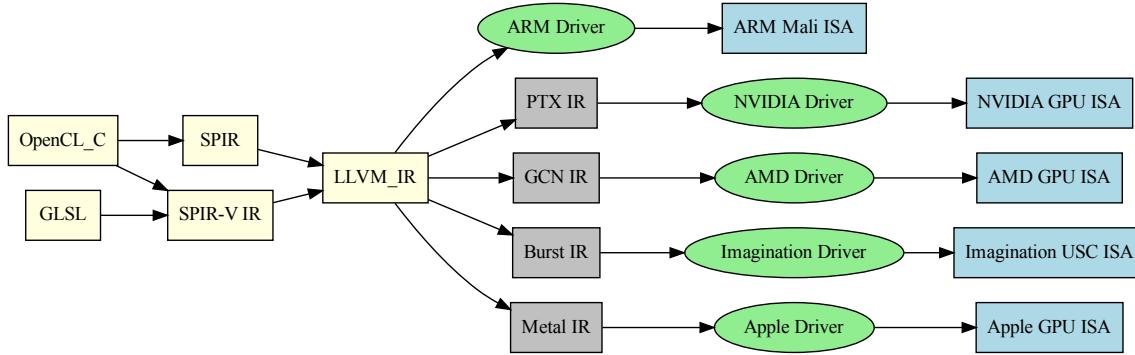


Fig. 14.75: Graphics and OpenCL Compiler IR Conversion Flow

Table 14.18: Comparison of PTX, GCN IR, Burst IR, and Metal IR

IR Layer	Abstraction Level	Register Model
PTX (NVIDIA)	Virtual ISA; portable across GPU generations; hides hardware scheduling	Virtual registers (%r, %f); mapped to physical registers during SASS lowering
GCN IR (AMD)	Machine IR; tightly coupled to GCN/RDNA architecture; exposes Wavefront semantics	Explicit vector (vN) and scalar (sN) registers; register pressure affects occupancy. AMD's compiler backend can lower vector operations to scalar instructions on low-end GPUs, while preserving vector operations on high-end architectures.
Burst IR (Imagination)	Power-aware IR; optimized for burst-core scheduling and latency hiding	Operand staging model; abstracted register usage; mapped late to USC ISA
Metal IR (Apple)	LLVM-inspired IR; abstracted from developers; tuned for tile shading and threadgroup fusion	Region-based register allocation; dynamic renaming; not exposed as physical register model

▀ NVIDIA, AMD, ARM and Imagination all have exposed LLVM IR and convert SPIR-V IR to LLVM IR.

- SPIR:
 - For OpenCL development, the IR started from SPIR (LLVM-based IR).
 - SPIR-V's Limitation: tightly coupled to specific LLVM versions, making it brittle across.
- SPIR-V:
 - A complete redesign: binary format, not tied to LLVM.
 - Designed for Vulkan, but also supports OpenCL and OpenGL.
 - Enables cross-vendor portability, shader reflection, and custom extensions.
 - Used in graphics and compute pipelines, including ML workloads via Vulkan compute.
 - A Vulkan shader written in GLSL is compiled to SPIR-V, then passed to the GPU driver.
 - An OpenCL kernel written in C can be compiled to SPIR-V, then lowered to LLVM IR internally by vendors like AMD or NVIDIA.

△ Apple

- Uses LLVM IR Partially. Apple supports SPIR-V in Metal and OpenCL, but LLVM IR is not always exposed.
- Metal shaders are compiled via MetalIR, which is LLVM-inspired but not standard LLVM IR. Metal IR is not standard LLVM IR and is not exposed to developers.
- Apple's ML compiler stack may use LLVM IR internally, but it's abstracted from developers.
- Apple is not a vendor of GPU IP, so it does not expose LLVM IR in its ML or graphics APIs for the reasons below:
 - Security: opaque compilation prevents tampering
 - Performance tuning: Apple controls the entire stack for optimal hardware use
 - Developer simplicity: high-level APIs reduce friction

Notes:

- **HLSL → DXIL → DirectX** is Microsoft's graphics pipeline, used on Windows and Xbox.
- **GLSL → SPIR-V → OpenGL/Vulkan** is cross-platform and supported by all vendors.
- Final GPU ISA varies by vendor:
 - NVIDIA: PTX → SASS
 - AMD: LLVM IR → GCN/RDNA
 - ARM: Mali ISA
 - Imagination: USC ISA
 - Apple: Metal GPU ISA

Notes:

- **OpenCL C → SPIR → Vendor Driver → GPU ISA** is the standard compilation path.
- Some vendors (e.g., AMD, NVIDIA) may bypass SPIR and compile directly to LLVM IR or PTX.
- Apple deprecated OpenCL in favor of Metal, but legacy support remains.

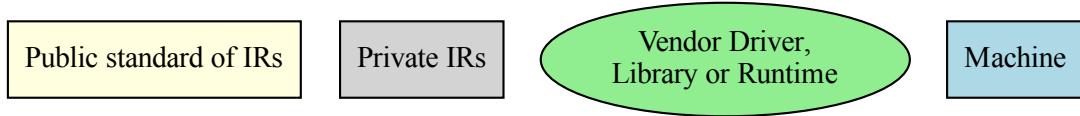
References

- OpenCL Specification: <https://www.khronos.org/opencl/>
- SPIR-V Specification: <https://www.khronos.org/spir>
- DirectX Shader Compiler: <https://github.com/microsoft/DirectXShaderCompiler>
- Imagination E-Series GPU: <https://www.imaginationtech.com/>
- Apple Metal API: <https://developer.apple.com/metal/>

ML and GPU Compilation

This section outlines the intermediate representation (IR) flows used by NVIDIA, AMD, and ARM in machine learning and GPU compilation pipelines. It includes both inference engines and compiler toolchains.

Each node in the graph is color-coded to indicate its category or role within the structure. In AI, usually use runtime instead of driver for graphics.



NVIDIA IR Conversion Flow

NVIDIA supports both TensorRT-based inference and MLIR-based compilation targeting CUDA GPUs is shown as Fig. 14.76.

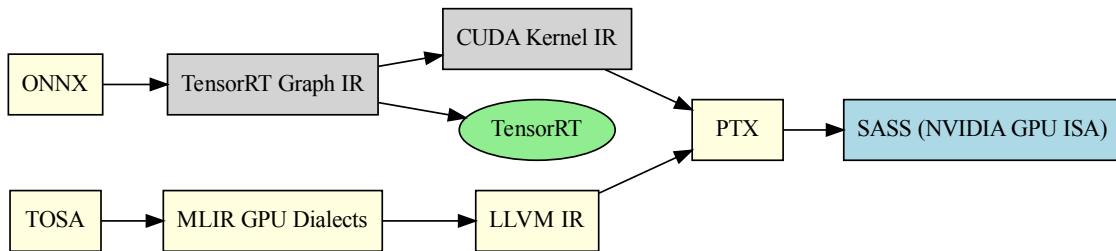


Fig. 14.76: NVIDIA IR Conversion Flow

- SASS stands for Streaming ASSEMBLER, and it represents the native instruction set architecture (ISA) for NVIDIA GPUs.
- TensorRT is a C++ library and runtime developed by NVIDIA for deep learning inference—the phase where trained models make predictions.
 - It works with models trained in frameworks like TensorFlow, PyTorch, and ONNX, converting them into highly optimized engines for execution on NVIDIA hardware¹²³¹²⁴.
- CUDA Kernel IR is a bridge between LLVM IR and PTX/SASS, or a direct output from TensorRT.
- LLVM IR is foundational in many flows, but **TensorRT may skip it** and directly emit CUDA kernels.
- Although MLIR dialects may be publicly available, they are typically hardware-dependent and tailored to specific vendors' GPU architectures. As a result, their applicability is limited to the corresponding hardware platforms.
- MLIR GPU Dialects is public but it is for Nvidia's GPU.

¹²³ <https://resources.nvidia.com/en-us-inference-resources/nvidia-tensorrt>

¹²⁴ <https://www.geeksforgeeks.org/deep-learning/what-is-tensorrt/>

AMD IR Conversion Flow

AMD uses ROCm and MIOpen for ML workloads, with LLVM-based compilation targeting GCN or RDNA architectures is shown as Fig. 14.77.

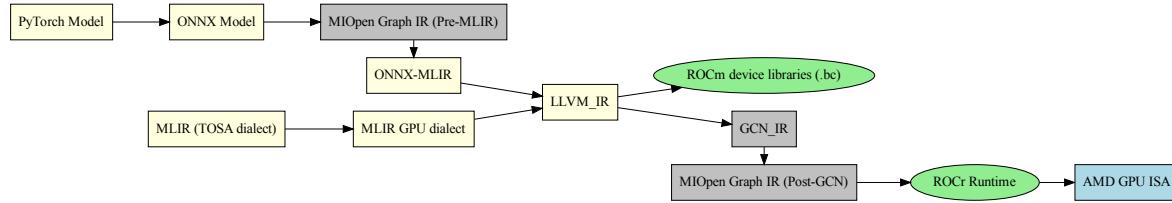


Fig. 14.77: AMD IR Conversion Flow

- **ROCm** is not just a compiler or driver —it includes a full runtime stack that enables AMD GPUs to execute compute workloads across HIP (Heterogeneous-compute Interface for Portability), OpenCL, and ML frameworks. It’s **analogous to NVIDIA’s CUDA runtime** but built around open standards like HSA (Heterogeneous System Architecture)¹²⁵ and LLVM.
- **MIOpen Graph IR** includes different form and structure. (**Pre-MLIR**) and (**Post-GCN**) are different.
 - Developers interact with MIOpen via high-level APIs (e.g., miopenConvolutionForward) —not via direct IR manipulation.
 - While MIOpen itself is open source (GitHub repo), its graph IR format and transformation passes are internal.

ARM IR Conversion Flow

ARM supports both CPU/NPU deployment (e.g., Ethos-U/N) and GPU execution (e.g., Mali via Vulkan). The IR flow diverges depending on the target is shown as Fig. 14.78.

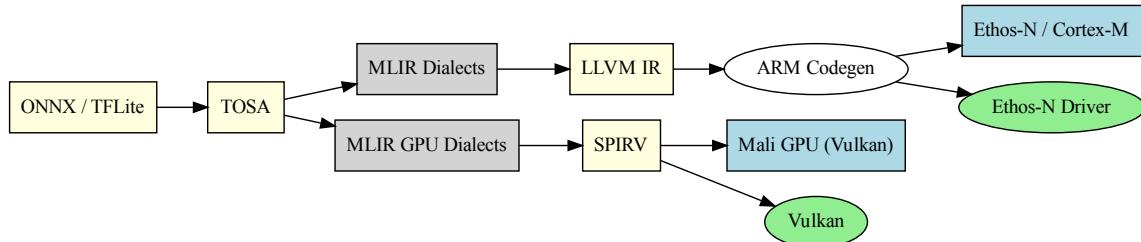


Fig. 14.78: ARM IR Conversion Flow

- Node “**Mali GPU (Vulkan)**” is the SPIR-V compilation flow that illustrated in the previous section.

¹²⁵ HSA is an open standard developed to simplify programming across heterogeneous systems —especially those combining CPUs and GPUs. It defines:

- Agents: CPUs, GPUs, and other compute units treated uniformly
- Queues: Asynchronous command queues for dispatching kernels
- Memory model: Shared virtual memory across agents
- Signals: Lightweight synchronization primitives

- Ethos-N is ARM's NPU. Cortex-M is ARM's CPU.

ARM Codegen generally emits instructions for CPU/NPU execution, but for certain NN operations (especially those requiring vendor-specific acceleration), it may generate function calls into the Ethos-N driver, which then orchestrates execution on the NPU.

✓ Common Case: Direct NPU/CPU Instruction Generation

- For operations that are well-supported by the NPU or CPU, the codegen backend emits hardware-specific instructions or IR directly.
- These are scheduled for execution on the CPU or passed to the Ethos-N via its driver stack.

✗ Special Case: Function Calls to Ethos-N Driver

- For complex or fused neural network operations (e.g., custom activation functions, quantized convolutions, or optimized memory layouts), the codegen may emit calls (**LLVM-IR `call`**) to precompiled driver functions.
- These function calls act as entry points into the Ethos-N runtime, which handles:
 - Memory management
 - Scheduling
 - Firmware-level execution
 - Hardware-specific optimizations

Imagination Technologies IR Conversion Flow

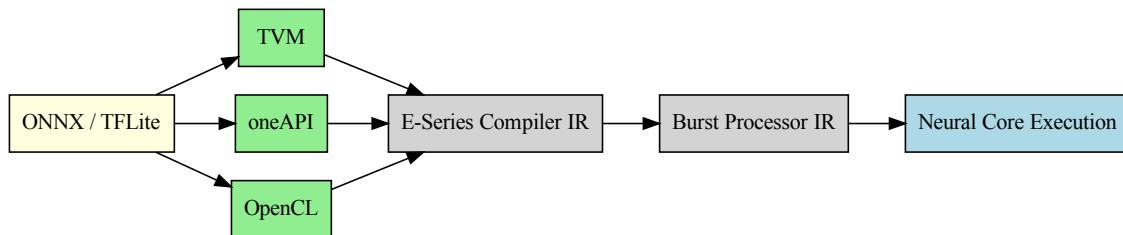


Fig. 14.79: Imagination Technologies IR Conversion Flow

Notes:

- E-Series GPUs support up to **200 TOPS INT8/FP8** for edge AI workloads [B](https://www.techpowerup.com/336545/imagination-announces-e-series-gpu-ip-with-burst-processors-and-up-to-200-tops?copilot_analytics_metadata=eyJldmVudEluZm9fY29udmVyc2F0aW9uSWQiOjJMSINQWjRaWjY5Y2ZuN0VnWnJEVzEiLCJldmVudEluZm9fbV3D%3D&citationMarker=9F742443-6C92-4C44-BF58-8F5A7C53B6F1).
- The architecture is **programmable**, supporting **graphics and AI** workloads simultaneously.
- Developers can target the GPU using **OpenCL**, **Apache TVM**, or **oneAPI**.
- The **Burst Processor IR** optimizes power efficiency and memory locality.
- Final execution occurs on **Neural Cores**, deeply integrated into the GPU.

Comparison Summary

Vendor	High-Level IR	Mid-Level IR	Low-Level IR	Libraries / Runtimes
NVIDIA	ONNX, TensorRT IR	MLIR GPU Dialects	PTX → SASS	TensorRT
AMD	ONNX, MIOpen IR	MLIR Dialects	LLVM IR → GCN ISA	MIOpen, ROCm
ARM	ONNX, TFLite	TOSA, MLIR Dialects	LLVM IR / SPIR-V	Ethos-N Driver, Vulkan
Imagination	ONNX, TFLite	E-Series Compiler IR	Burst IR → Neural Core	OpenCL, TVM, oneAPI

References

- NVIDIA TensorRT: <https://developer.nvidia.com/tensorrt>
- AMD ROCm: <https://rocm.docs.amd.com/>
- ARM ML Toolchain: <https://developer.arm.com/solutions/machine-learning>
- Imagination:
 - Imagination E-Series GPU IP: <https://www.imaginationtech.com/>
 - TechPowerUp E-Series Launch: <https://www.techpowerup.com/336545/imagination-announces-e-series-gpu-ip-with-burst-processors-and-up-to-200> [A] (
- MLIR Project: <https://mlir.llvm.org/>
- Vulkan API: <https://www.khronos.org/vulkan/>
- Apache TVM: <https://tvm.apache.org/>
- oneAPI: <https://www.oneapi.io/>

14.4.4 Accelerate ML/DL on OpenCL/SYCL

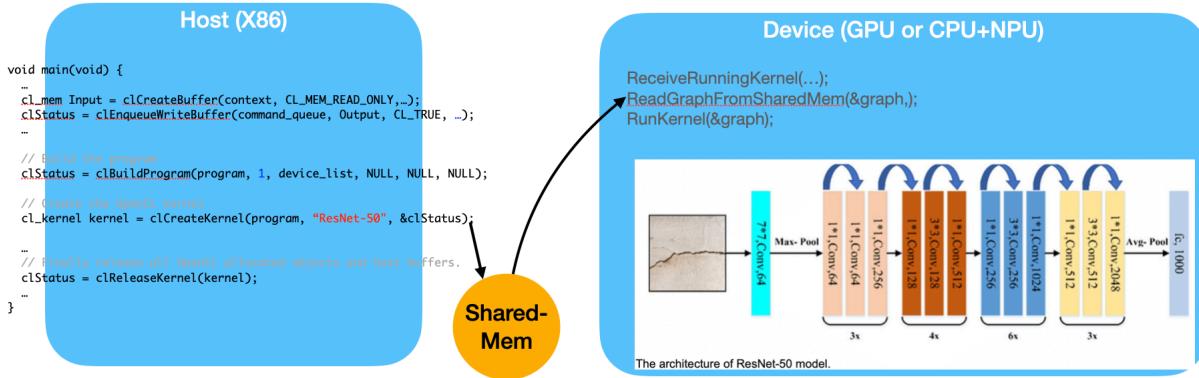


Fig. 14.80: Implement ML graph scheduler both on compiler and runtime

As shown in Fig. 14.80, the Device, such as a GPU or a CPU+NPU, is capable of running the entire ML graph. However, if the Device has only an NPU, then operations like Avg-Pool, which require CPU support, must run on the Host side. This introduces communication overhead between the Host and the Device.

Similar to OpenGL shaders, the “kernel” function may be compiled either on-line or off-line and then sent to the GPU as a programmable function.

In order to run ML (Machine Learning) efficiently, all platforms for ML on GPU/NPU implement scheduling SW both on graph compiler and runtime. **If OpenCL can extend to support ML graph, then graph compiler such as TVM or Runtime from Open Source have chance to leverage the effort of scheduling SW from programmers¹²⁶**. Cuda graph is an idea like this¹²⁷¹²⁸.

- SYCL: Using C++ templates to optimize and generate code for OpenCL and Cuda. Provides a consistent language, APIs, and ecosystem in which to write and tune code for different accelerator architecture, CPUs, GPUs, and FPGAs¹²⁹.
 - SYCL uses generic programming with templates and generic lambda functions to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code across an extensive range of acceleration backend APIs, such as OpenCL and CUDA¹³⁰.

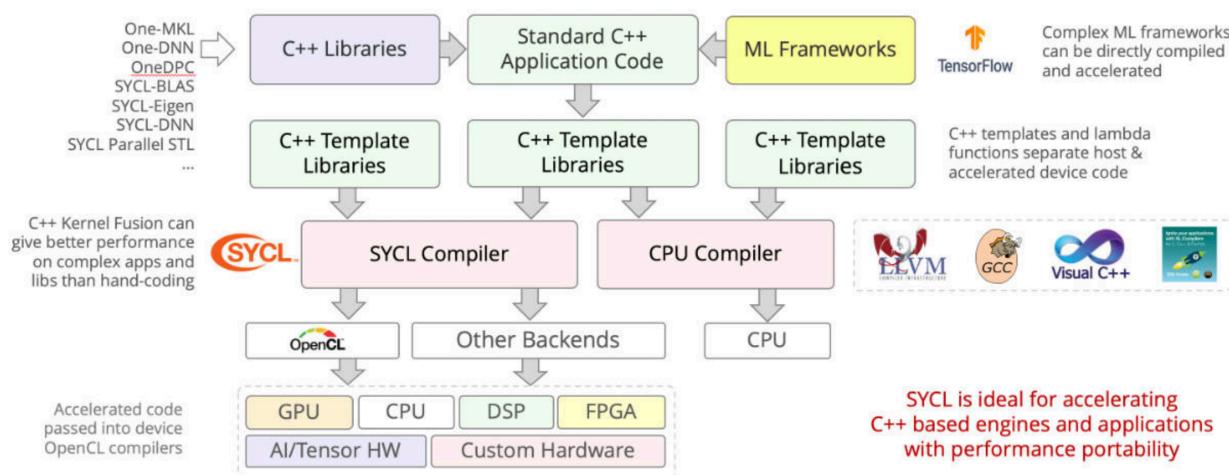


Fig. 14.81: SYCL = C++ template and compiler for Data Parallel Applications on AI on CPUs, GPUs and HPGAs.

- DPC++ (OneDPC) compiler: Based on SYCL, DPC++ can compile the DPC++ language for both CPU host and GPU device. DPC++ (Data Parallel C++) is a language developed by Intel and may be adopted into standard C++. The GPU-side (kernel code) is written in C++ but does not support exception handling¹³¹¹³².

– Features of Kernel Code:

- * Not supported:

Dynamic polymorphism, dynamic memory allocations (therefore no object management using new or delete operators), static variables, function pointers, runtime type information (RTTI), and **exception handling**. No virtual member functions, and no variadic functions, are allowed to be called from kernel code. Recursion is not allowed within kernel code.

¹²⁶ <https://easychair.org/publications/preprint/GjhX>

¹²⁷ <https://developer.nvidia.com/blog/cuda-graphs/>

¹²⁸ <https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/>

¹²⁹ <https://www.khronos.org/sycl/>

¹³⁰ <https://github.com/codeplaysoftware/sycl-for-cuda/blob/cuda/sycl/doc/GetStartedWithSYCLCompiler.md>

¹³¹ <https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html#gs.cxolyy>

¹³² <https://link.springer.com/book/10.1007/978-1-4842-5574-2>

* Supported:

Lambdas, operator overloading, templates, classes, and static polymorphism¹³³.

14.5 Open Sources

- <https://registry.khronos.org/OpenGL-Refpages/>
- <https://www.mesa3d.org>
- <https://www.opengl.org/sdk/>, <https://www.opengl.org/sdk/libs/>

¹³³ Page 14 of DPC++ book.

APPENDIX A: GETTING STARTED: INSTALLING LLVM AND THE CPU0 EXAMPLE CODE

- *Build Steps*
- *Setting Up Your Mac*
 - *Install Icarus Verilog Tool on iMac*
 - *Install Other Tools on iMac*
- *Setting Up Your Linux Machine*
 - *Install Icarus Verilog tool on Linux*
 - *Install Other Tools on Linux*
- *Toolchain*

The Cpu0 example code, *lbdex*, can be found at the lower left section of this website, or directly via this link: <http://jonathan2251.github.io/lbd/lbdex.tar.gz>.

For details on using `cmake` to build LLVM, refer to “Building LLVM with CMake”¹ documentation.

We will install two LLVM directories in this chapter. One is the directory `~/llvm/debug/`, which contains the `clang` and `clang++` compilers used to translate C/C++ source files into LLVM IR. The other is `~/llvm/test/`, which contains our Cpu0 backend program and another `clang` build.

15.1 Build Steps

On Linux, using multi-threading (`-DLLVM_PARALLEL_COMPILE_JOBS=4`) requires more than 16GB of memory. I created a 64GB swap file to avoid link failures²³. iMac systems typically do not encounter this issue.

```
$ cat /etc/fstab
# <file system> <mount point> <type> <options> <dump> <pass>
...
/swapfile swap swap default 0 0
```

After installing necessary packages (via `brew` on iMac), follow the build instructions here: <https://github.com/Jonathan2251/lbd/blob/master/README.md>.

¹ <http://llvm.org/docs/CMake.html?highlight=cmake>

² <https://bogdancornianu.com/change-swap-size-in-ubuntu/>

³ <https://linuxize.com/post/how-to-add-swap-space-on-ubuntu-18-04/>

15.2 Setting Up Your Mac

Install Homebrew first⁴:

```
% /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

After installation, add the `brew` command to your shell's PATH, as shown in the final message of the install script. The command typically looks like the following:

```
% echo 'eval "$( /opt/homebrew/bin/brew shellenv )"' >> /Users/cschen/.zprofile
% eval "$( /opt/homebrew/bin/brew shellenv )"
```

For installing Homebrew in China, use the following install script instead⁵.

```
% /bin/zsh -c "$(curl -fsSL https://gitee.com/cunkai/HomebrewCN/raw/master/Homebrew.sh)"
...
% source /Users/cschen/.zprofile
% brew --version
Homebrew 3.6.7-28-g560f571
fatal: detected dubious ownership in repository at '/usr/local/Homebrew/Library/Taps/homebrew/homebrew-core'
To add an exception for this directory, call:

      git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/homebrew/homebrew-core
Homebrew/homebrew-core (no Git repository)
fatal: detected dubious ownership in repository at '/usr/local/Homebrew/Library/Taps/homebrew/homebrew-cask'
To add an exception for this directory, call:

      git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/homebrew/homebrew-cask

% git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/homebrew/homebrew-core
% git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/homebrew/homebrew-cask
% brew install cmake
...
==> Running `brew cleanup cmake`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).

% brew install ninja
```

⁴ <https://brew.sh/>

⁵ https://blog.csdn.net/weixin_45571585/article/details/126977413

15.2.1 Install Icarus Verilog Tool on iMac

Install Icarus Verilog using the command `brew install icarus-verilog` as shown below:

```
% brew install icarus-verilog
==> Downloading ftp://icarus.com/pub/eda/verilog/v0.9/verilog-0.9.5.tar.gz
# ######################################################################## 100.0%
# ######################################################################## 100.0%
==> ./configure --prefix=/usr/local/Cellar/icarus-verilog/0.9.5
==> make
==> make installdirs
==> make install
/usr/local/Cellar/icarus-verilog/0.9.5: 39 files, 12M, built in 55 seconds
```

15.2.2 Install Other Tools on iMac

Install CMake and Ninja with the following command:

```
brew install cmake ninja
```

Install Graphviz for displaying LLVM IR nodes during debugging⁷.

```
% brew install graphviz
```

Information about using Graphviz with LLVM is available in the section “SelectionDAG Instruction Selection Process” of “The LLVM Target-Independent Code Generator”⁸, and in the section “Viewing graphs while debugging code” of the “LLVM Programmer’s Manual”⁹.

Install binutils with the following command:

```
// get brew by the following ruby command if you don't have installed brew
118-165-77-214:~ Jonathan$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)" < /dev/null 2> /dev/null
118-165-77-214:~ Jonathan$ brew install binutils
==> Downloading http://ftpmirror.gnu.org/binutils/binutils-2.22.tar.gz
# ######################################################################## 100.0%
==> ./configure --program-prefix=g --prefix=/usr/local/Cellar/binutils/2.22
--infodir=/usr/local
==> make
==> make install
/usr/local/Cellar/binutils/2.22: 90 files, 19M, built in 4.7 minutes
118-165-77-214:~ Jonathan$ ls /usr/local/Cellar/binutils/2.22
COPYING      README      lib
ChangeLog     bin        share
INSTALL_RECEIPT.json   include    x86_64-apple-darwin12.2.0
118-165-77-214:binutils-2.23 Jonathan$ ls /usr/local/Cellar/binutils/2.22/bin
gaddr2line  gc++filt  gnm  gobdump  greadelf  gstrings
gar  gelfedit  gobjcopy  granlib  gsize  gstrip
```

⁷ <https://graphviz.org/download/>

⁸ <http://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process>

⁹ <http://llvm.org/docs/ProgrammersManual.html#viewing-graphs-while-debugging-code>

15.3 Setting Up Your Linux Machine

15.3.1 Install Icarus Verilog tool on Linux

Download Icarus Verilog as follows¹⁰.

```
$ git clone http://iverilog.icarus.com/
```

Follow the README or INSTALL file guide to install it.

Install *sh autoconf.sh* dependencies and other dependencies as follows,

```
$ pwd
$ ~/git/iverilog
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
patchutils bc zlib1g-dev libexpat-dev
$ sh autoconf.sh
```

Then install Icarus Verilog using the following commands,

```
$ ./configure
// or below if you are in shared server
$ ./configure --prefix=$HOME/local
$ make
$ make check
```

15.3.2 Install Other Tools on Linux

Install CMake and Ninja as follows,

```
$ pwd
$ ~/local
$ wget -b https://github.com/Kitware/CMake/releases/download/v3.23.3/cmake-3.23.3-
linux-x86_64.sh
$ bash cmake-3.23.3-linux-x86_64.sh
Do you accept the license? [yn]:
Y
By default the CMake will be installed in:
"/u/jonathanchen/local/cmake-3.23.3-linux-x86_64"
Do you want to include the subdirectory cmake-3.23.3-linux-x86_64?
Saying no will install in: "/u/jonathanchen/local" [Yn]:
Y
...
Unpacking finished successfully
$ ls
bin cmake-3.23.3-linux-x86_64 ...
$ sudo apt install ninja-build
```

Download Graphviz from¹¹ according to your Linux distribution. The file comparison tool KDiff3 can be downloaded from the website⁶.

¹⁰ <http://iverilog.icarus.com/>

¹¹ <http://www.graphviz.org/Download.php>

⁶ <http://kdiff3.sourceforge.net>

```
$ sudo apt install graphviz  
$ dot -V  
dot - graphviz version 2.40.1 (20161225.0304)
```

Set `~/.profile` as follows,

`~/.profile`

```
~/.profile: executed by the command interpreter for login shells.  
...  
# set PATH so it includes user's private bin if it exists  
if [ -d "$HOME/local/bin" ] ; then  
    PATH="$HOME/local/bin:$PATH"  
fi  
# set PATH for cmake  
if [ -d "$HOME/local/cmake-3.23.3-linux-x86_64/bin" ] ; then  
    PATH="$HOME/local/cmake-3.23.3-linux-x86_64/bin:$PATH"  
fi  
...
```

15.4 Toolchain

List some GNU and LLVM tools as follows,

```
// Linux  
~/git/lbd/lbdex/input$ ~/llvm/debug/build/bin/clang -fpic hello.c  
~/git/lbd/lbdex/input$ man ldd  
ldd - print shared object dependencies  
~/git/lbd/lbdex/input$ ldd a.out  
    linux-vdso.so.1 (0x00007ffffd1fe5000)  
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2c92a82000)  
    /lib64/ld-linux-x86-64.so.2 (0x00007f2c92e73000)  
  
// MacOS  
% man otool  
otool-classic - object file displaying tool  
...  
-L      Display the names and version numbers of the shared libraries that  
        the object file uses, as well as the shared library ID if the file  
        is a shared library.  
% otool -L a.out  
a.out:  
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current  
    version 1292.100.5)  
  
// Linux  
~/git/lbd/lbdex/input$ man objcopy  
objcopy - copy and translate object files  
...  
[-O bfdname|--output-target=bfdname]  
~/git/lbd/lbdex/input$ objcopy -O verilog a.out a.hex  
~/git/lbd/lbdex/input$ vi a.hex
```

(continues on next page)

(continued from previous page)

```
@00400238  
2F 6C 69 62 36 34 2F 6C 64 2D 6C 69 6E 75 78 2D  
78 38 36 2D 36 34 2E 73 6F 2E 32 00  
@00400254  
04 00 00 00 10 00 00 00 01 00 00 00 47 4E 55 00  
...
```

APPENDIX B: CPU0 DOCUMENT AND TEST

- *github*
 - *Create gh-pages*
- *reST format*
 - *Six Levels of Chapter, section, subsections*
 - *Symbols*
 - *Math*
 - *Figures*
 - *Graphviz*
- *Cpu0 document*
 - *Install sphinx*
 - *Note of Sphinx*
 - *Generate Cpu0 document*
 - *About Cpu0 Document*
- *Cpu0 Regression Test*

16.1 github

16.1.1 Create gh-pages

1. Create a new orphan branch for GitHub Pages and clean the directory.

```
% git branch
* gh-pages
master

% git checkout --orphan gh-pages
% git rm -rf .
```

2. GitHub Pages build error
 - Cause: GitHub tries to build Jekyll by default

- Fix: Add a .nojekyll file to the root of gh-pages:

```
% git branch
* gh-pages
master

% touch .nojekyll
% git add .nojekyll
% git commit -m "Disable Jekyll"
% git push
```

16.2 reST format

The reStructure format (Sphinx).

16.2.1 Six Levels of Chapter, section, subsections

Six Levels of Chapter, section, subsections

```
1. Chapter -- Level 1
=====
Level 1 Title
=====

2. Section -- Level 2
-----
Level 2 Title
-----

3. Subsection -- Level3
*****
Level 3 Title
*****

4. Subsection -- Level4
^^^^^^^^^^^^^^^^^
Level 4 Title
^^^^^^^^^^^^^

5. Subsection -- Level5
#####
Level 5 Title
#####

6. Subsection -- Level6
~~~~~
Level 6 Title
~~~~~
```

- Do not use more than 90 characters per line in `code-block:: console` or any other code block, otherwise errors may occur during *make latexpdf*.
- Use `\clearpage` (see `gpu.rst`) to force page breaks in pdf and avoid content splitting across two pages.

- For instructions on inserting a reference before any paragraph that is not a section heading, see the word `cfg-warps` in `gpu.rst`.

16.2.2 Symbols

Pdf (xelatex) recognizes:

4. Arrows

- $\leftarrow \rightarrow \uparrow \downarrow$
- $\leftrightarrow \updownarrow$
- $\mapsto \leftarrow\rightarrow$
- $\Leftarrow \Rightarrow \uparrow \Downarrow$
- \Leftrightarrow
- Use math: :math:`\backslash Leftarrow` or :math:`\backslash Rightarrow`; $\Leftarrow \Rightarrow$; reference *Cross Product*.

\Leftarrow, \Rightarrow

16.2.3 Math

- Sphinx supports math symbols; see references⁴ and⁵ for more details.

16.2.4 Figures

- Figures side by side: refer to sm:left in gpu.rst.

16.2.5 Graphviz

- To draw nodes inside a parent node, use “compound=true;” and subgraph cluster_name⁷. To create edges from/to parent nodes, use the attributes “ltail” and “lhead” set to the parent node.
- See graphviz documents here⁶.

16.3 Cpu0 document

This section illustrates how to generate Cpu0 backend document.

16.3.1 Install sphinx

LLVM and this book use Sphinx to generate HTML documents. This book uses Sphinx to generate PDF and EPUB formats as well. See the installation guide here¹.

Sphinx uses reStructuredText format; see⁸,⁹, and¹⁰. For code-block formatting in this document, see¹¹ and¹².

On iMac you can install as follows:

```
brew install sphinx-doc
echo 'export PATH="/opt/homebrew/opt/sphinx-doc/bin:$PATH"' >> ~/.zshrc
% source ~/.zshrc
```

⁴ <https://sphinx-rtd-trial.readthedocs.io/en/latest/ext/math.html#module-sphinx.ext.mathbase>

⁵ <https://mirrors.mit.edu/CTAN/info/short-math-guide/short-math-guide.pdf>

⁷ Ex. lbd/Fig/gpu/opengl-flow.gv. If the name of the subgraph begins with cluster, Graphviz notes the subgraph as a special cluster subgraph. If supported, the layout engine will do the layout so that the nodes belonging to the cluster are drawn together, with the entire drawing of the cluster contained within a bounding rectangle. <https://graphviz.org/doc/info/lang.html>.

⁶ <https://www.graphviz.org/pdf/dotguide.pdf>

¹ <https://www.sphinx-doc.org/en/master/usage/installation.html>

⁸ <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>

⁹ <http://docutils.sourceforge.net/docs/ref/rst/directives.html>

¹⁰ <http://docutils.sourceforge.net/rst.html>

¹¹ <http://llvm.org/docs/SphinxQuickstartTemplate.html> If you need to show LLVM IR use the llvm code block. <https://llvm.org/docs/SphinxQuickstartTemplate.html#code-blocks>

¹² <http://pygments.org/docs/lexers/>

On Linux, install with:

```
sudo apt-get install python3-sphinx
```

The above installs Sphinx for making HTML documents but not PDF.

For PDF/LaTeX generation, install the following.

On iMac, install MacTeX.pkg from here² and restart your computer.

On Linux, install texlive as follows:

```
sudo apt-get install texlive texlive-latex-extra latexmk
```

or

```
sudo yum install texlive texlive-latex-extra latexmk
```

On Fedora 17, the texlive-latex-extra package is missing. Instead, install the package that includes pdflatex. For example, install pdfjam on Fedora 17 as follows,

```
[root@localhost lbd]$ yum list pdfjam
Loaded plugins: langpacks, presto, refresh-packagekit
Installed Packages
pdfjam.noarch                               2.08-3.fc17                                     @fedora
[root@localhost lbd]$
```

Do

```
sudo apt-get update -y
sudo apt-get install -y latexmk
```

if latexmk gives error during ‘make latexpdf’³.

After upgrading to iMac OS X 10.11.1, the pdflatex link may be missing. Fix it by adding the following to your .profile:

```
114-37-153-62:1bd Jonathan$ ls /usr/local/texlive/2012/bin/universal-darwin/pdflatex
/usr/local/texlive/2012/bin/universal-darwin/pdflatex
114-37-153-62:1bd Jonathan$ cat ~/.profile
export PATH=$PATH:...:/usr/local/texlive/2012/bin/universal-darwin
```

16.3.2 Note of Sphinx

```
$HOME/git/lbd$ make latexpdf

Latexmk: applying rule 'pdflatex'...
Rule 'pdflatex': File changes, etc:
    Changed files, or newly in use since previous run(s):
        LLVMToolchainCpu0.aux
        LLVMToolchainCpu0.toc
Rule 'pdflatex': The following rules & subrules became out-of-date:
    pdflatex
Latexmk: Maximum runs of pdflatex reached without getting stable files
```

(continues on next page)

² <http://www.tug.org/mactex/>

³ <https://zoomadmin.com/HowToInstall/UbuntuPackage/latexmk>

(continued from previous page)

```
Latexmk: All targets (LLVMToolchainCpu0.pdf) are up-to-date
-----
This message may duplicate earlier message.
Latexmk: Failure in processing file 'LLVMToolchainCpu0.tex':
  'pdflatex' needed too many passes
-----
Latexmk: If appropriate, the -f option can be used to get latexmk
  to try to force complete processing.
make[1]: *** [LLVMToolchainCpu0.pdf] Error 12
make: *** [latexpdf] Error 2
make latexpdf 14.57s user 1.11s system 99% cpu 15.790 total
```

16.3.3 Generate Cpu0 document

The Cpu0 example code is added chapter by chapter. It can be configured to a specific chapter by changing the CH definition in Cpu0SetChapter.h. For example, the following definition configures it to chapter 2.

Ibdex/Cpu0/Cpu0SetChapter.h

```
#define CH          CH2
```

To help readers understand the backend structure step by step, the Cpu0 example code can be generated chapter by chapter using the following commands:

```
118-165-12-177:lbd Jonathan$ pwd
/home/Jonathan/test/lbd
118-165-12-177:lbd Jonathan$ make genexample
...
118-165-12-177:lbd Jonathan$ ls lbdex/chapters/
Chapter10_1  Chapter2      Chapter3_4  Chapter5_1  Chapter8_2
Chapter11_1  Chapter3_1    Chapter3_5  Chapter6_1  Chapter9_1
Chapter11_2  Chapter3_2    Chapter4_1  Chapter7_1  Chapter9_2
Chapter12_1  Chapter3_3    Chapter4_2  Chapter8_1  Chapter9_3
```

Besides the example code in each chapter, the above HTML and PDF Cpu0 documents also include the *.ll and *.s files located in the lbd/lbdex/output directory.

```
JonathantekeiMac:lbd Jonathan$ ls lbdex/output/
ch12_eh.cpu0.s                      ch12_thread_var.cpu0.pic.s      ch12_thread_var.
↪11
ch12_eh.ll                           ch12_thread_var.cpu0.static.s   ch4_math.s
```

Then, the HTML and PDF versions of this book can be generated by running the following commands.

```
118-165-12-177:lbd Jonathan$ pwd
/home/Jonathan/test/lbd
118-165-12-177:lbd Jonathan$ make html
...
118-165-12-177:lbd Jonathan$ make latexpdf
...
```

16.3.4 About Cpu0 Document

Since LLVM has a new release about every 6 months, and names of files, functions, classes, variables, etc. may change, maintaining the Cpu0 document is a continuous effort. The document adds code chapter by chapter.

To keep the document correct and easy to maintain, I use the `:start-after:` and `:end-before:` directives of reStructuredText to keep the document up to date.

For every new release, when the Cpu0 backend code changes, the document will reflect those changes in most of its content.

In the `lbdex/Cpu0` folder, text beginning with `//\@` and `#ifdef CH > CHxx` is referenced by document files `*.rst`.

In `lbdex/llvm/modify/llvm`, the `*.rst` files reference the code by copying it directly. Most references exist in `llvmstructure.rst` and `elf.rst`.

The example C/C++ code in `lbdex/input` comes from my own ideas and refers to the `clang/test/CodeGen` directory in the Clang source code release.

16.4 Cpu0 Regression Test

The last chapter can verify the Cpu0 backend's generated code by using a Verilog simulator for code without global variable access.

The chapter `lld` in the repository <https://github.com/Jonathan2251/lbt.git> includes an LLVM ELF linker implementation and can verify test items involving global variable access.

However, LLVM provides its own test cases (regression tests) for each backend to verify the backend compiler¹³ without needing a simulator or real hardware platform.

Cpu0 regression test items exist in the `lbdex.tar.gz` example code. Untar it into the `lbdex/` directory.

For both iMac and Linux, copy the folder:

```
lbdex/regression-test/Cpu0
```

To

```
~/llvm/test/llvm/test/CodeGen/Cpu0
```

Then run the tests as follows on iMac, for both single and all test cases:

```
$ llvm-lit -a ~/llvm/test/llvm/test/CodeGen/Cpu0
```

The option `-a` shows the executing commands for each test case.

```
1-160-130-77:Cpu0 Jonathan$ pwd
/Users/Jonathan/llvm/test/llvm/test/CodeGen/Cpu0
1-160-130-77:Cpu0 Jonathan$ ~/llvm/test/build/bin/llvm-lit seteq.ll
-- Testing: 1 tests, 1 threads --
PASS: LLVM :: CodeGen/Cpu0/seteq.ll (1 of 1)
Testing Time: 0.08s
  Expected Passes : 1
1-160-130-77:Cpu0 Jonathan$ ~/llvm/test/build/bin/llvm-lit .
...
PASS: LLVM :: CodeGen/Cpu0/zeroreg.ll
```

(continues on next page)

¹³ <http://llvm.org/docs/TestingGuide.html>

(continued from previous page)

```
PASS: LLVM :: CodeGen/Cpu0/tailcall.ll  
...
```

Run the following command to execute the test on Linux.

```
$ pwd  
/home/cschen/llvm/test/llvm/test/CodeGen/Cpu0  
$ ~/llvm/test/build/bin/llvm-lit seteq.ll  
-- Testing: 1 tests, 1 threads --  
PASS: LLVM :: CodeGen/Cpu0/seteq.ll (1 of 1)  
Testing Time: 0.08s  
    Expected Passes : 1  
$ ~/llvm/test/build/bin/llvm-lit .  
...  
PASS: LLVM :: CodeGen/Cpu0/zeroreg.ll  
PASS: LLVM :: CodeGen/Cpu0/tailcall.ll  
...
```

The chapters of this book and their related regression test items are listed as follows:

Table 16.1: Chapters

1	about
2	Cpu0 architecture and LLVM structure
3	Backend structure
4	Arithmetic and logic instructions
5	Generating object files
6	Global variables
7	Other data type
8	Control flow statements
9	Function call
10	ELF Support
11	Assembler
12	C++ support
13	Verify backend on verilog simulator

Table 16.2: Regression test items for Cpu0

File	v:pass x:fail	test ir, -> output asm	chapter
2008-06-05-Carry.ll	v		7
2008-07-15-InternalConstant.ll	v		6
2008-07-15-SmallSection.ll	v		6
2008-07-03-SRet.ll	v		9
2008-07-29-icmp.ll	v		8
2008-08-06-Alloca.ll	v		9
2008-08-01-AsmInline.ll	v		11
2008-08-08-ctlz.ll	v		7
2008-08-08-bswap.ll	v	bswap	12

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
2008-10-13-LegalizerBug.ll	v		8
2010-11-09-Mul.ll	v		4
2010-11-09-CountLeading.ll	v		7
2008-11-10-xint_to_fp.ll	v		7
addc.ll	v	64-bit add	7
addi.ll	v	32-bit add, sub	4
address-mode.ll	v	br, -> BB0_2:	8
alloca.ll	v	alloca i8, i32 %size, dynamic allocation	9
analyzebranch.ll	v	br, -> bne, beq	8
and1.ll	v	and	4
asm-large-immediate.ll	v	inline asm	11
atomic-1.ll	v	atomic	12
atomic-2.ll	v	atomic	12
atomics.ll	v	atomic	12
atomics-index.ll	v	atomic	12
atomics-fence.ll	v	atomic	12
br-jmp.ll	v	br, -> jmp	8
blockaddress.ll	v	blockaddress, -> lui, ori	8
cmove.ll	v	select, -> movn, movz	8
cprestore.ll	v	-> .cprestore	9
div.ll	v	sdiv, -> div, mflo	4
divrem.ll	v	sdiv, srem, udiv, urem, -> div, divu	4
div_rem.ll	v	sdiv, srem, -> div, mflo, mfhi	4
divu.ll	v	udiv, -> divu, mflo	4
divu_rem.ll	v	udiv, urem -> div, mflo, mfhi	4
double2int.ll	v	double to int, -> %call16(__fixdfsi)	7
eh-dwraf-cfa.ll	v		9
eh-return32.ll	v	Spill and reload all registers used for exception	9
eh.ll	v	c++ exception handling	12
ex2.ll	v	c++ exception handling	12
fastcc.ll	v	No effect in fastcc but can pass	9
fneg.ll	v	verify Cpu0 don't uses hard float instruction	7
fp-spill-reload.ll	v	-> st \$fp, ld \$fp	9
frame-address.ll	v	addu \$2, \$zero, \$fp	9
global-address.ll	v	global address, global variable	6
global-pointer.ll	v	global register load and restore, -> .cpload, .cprestore	9
gprestore.ll	v	global register restore, -> .cprestore	9

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
helloworld.ll	v	global register load and restore, -> .cupload, .cprestore	9
hf16_1.ll	v	function call in PIC, -> ld, jalr	9
i32k.ll	v	argument of constant int passing in register	9
i64arg.ll	v	argument of constant 64-bit passing in register	9
imm.ll	v	return constant 32-bit in register	9
indirectcall.ll	v	indirect function call	9
init-array.ll	v	check .init	6
inlineasm_constraint.ll	v	inline asm	11
inlineasm-cnstrnt-reg.ll	v	•	11
inlineasmmemop.ll	v	•	11
inlineasm-operand-code.ll	v	•	11
internalfunc.ll	v	internal function	9
jstat.ll	v	switch, -> JTI	8
lb1.ll	v	load i8*, sext i8, -> lb	7
lbu1.ll	v	load i8*, zext i8, -> lbu	7
lh1.ll	v	load i16*, sext i16, -> lh	7
lhu1.ll	v	load i16*, zext i16, -> lhu	7
llcarry.ll	v	64-bit add sub	7
longbranch.ll	v		8
machineverifier.ll	v	delay slot, (comment in machineverifier.ll)	8
mipslopat.ll	v	no check output (comment in mipslopat.ll)	6
misha.ll	v	miss alignment half word access	7
module-asm.ll	v	module asm	11
module-asm-cpu032II.ll	v	module asm	11
mul.ll	v	mul	4
mulll.ll	v	64-bit mul	4
mulull.ll	v	64-bit mul	4
not1.ll	v	not 1	4
null.ll	v	ret i32 0, -> ret \$lr	3
o32_cc_byval.ll	v	by value	9
o32_cc_vararg.ll	v	variable argument	9
private.ll	v	private function call	9
rem.ll	v	srem, -> div, mfhi	4
remat-immed-load.ll	v	immediate load	3
remul.ll	v	urem, -> div, mfhi	4
return-vector-float4.ll	v	return vector, -> lui lui ...	3
return-vector.ll	v	return vector, -> ld ld ..., st	3
		st ...	
return_address.ll	v	llvm.returnaddress, -> addu \$2, \$zero, \$lr	9
rotate.ll	v	rotl, rotr, -> rolv, rol, rorv	4
sb1.ll	v	store i8, sb	7

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
select.ll	v	select, -> movn, movz	8
seleq.ll	v	following for br with different condition	8
seleqk.ll	v	•	8
selgek.ll	v	•	8
selgt.ll	v	•	8
selle.ll	v	•	8
selltk.ll	v	•	8
selne.ll	v	•	8
selnek.ll	v	•	8
seteq.ll	v	•	8
seteqz.ll	v	•	8
setge.ll	v	•	8
setgek.ll	v	•	8
setle.ll	v	•	8
setlt.ll	v	•	8
setltk.ll	v	•	8
setne.ll	v	•	8
setuge.ll	v	•	8
setugt.ll	v	•	8
setule.ll	v	•	8
setult.ll	v	•	8
setultk.ll	v	•	8
sext_inreg.ll	v	sext i1, -> shl, sra	4
shift-parts.ll	v	64-bit shl, lshr, ash, -> call function	9
shl1.ll	v	shl, -> shl	4
shl2.ll	v	shl, -> shlv	4
shr1.ll	v	shr, -> shr	4
shr2.ll	v	shr, -> shrv	4
sitofp-selectcc-opt.ll	v	comment in sitofp-selectcc-opt.ll	7
small-section-reserve-gp.ll	v	Cpu0 option -cpu0-use-small-section=true	6
sra1.ll	v	ashr, -> sra	4
sra2.ll	v	ashr, -> srav	4
stacksave-restore.ll	v		9
stacksize.ll	v	comment in stacksize.ll	9
stchar.ll	v	load and store i16, i8	7
stldst.ll	v	register sp spill	9
sub1.ll	v	sub, -> addiu	4
sub2.ll	v	sub, -> sub	4
tailcall.ll	v	tail call	9
tls.ll	v	ir thread_local global is for c++ “__thread int b;”	12
tls-alias.ll	v	thread_local global and thread local alias	12
tls-models.ll	v	ir external/internal thread_local global	12

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
uitofp.ll	v	integer2float, uitofp, -> jsub __floatunisf	9
uli.ll	v	unalignment init, -> sb sb	6
		...	
unalignedload.ll	v	unalignment init, -> sb sb	6
		...	
vector-setcc.ll	v		7
weak.ll	v	extern_weak function, -> .weak	9
xor1.ll	v	xor, -> xor	4
zeroreg.ll	v	check register \$zero	4

These supported test cases are located in `lbdex/regression-test/Cpu0`, which can be extracted from `tar -xf lbdex.tar.gz`.

The regression test is useful for two major reasons. First, it provides the LLVM input, assembly output, and the corresponding command and options within the same sample input file. This makes it a well-documented reference for both end users and developers.

Second, when developers make changes to their backend compiler—especially for optimization—these tests help detect side effects or bugs caused by the modifications. This is the core purpose of “regression testing.”

The following file includes the assembly output patterns for two subtargets of the Cpu0 backend. In addition to checking opcodes, it can also verify register numbers. For example, the destination register of the “andi” instruction must match the first source register of the following “xori” instruction. This is ensured when both are specified as register T1 in the corresponding assembly output.

lbdex/regression-test/Cpu0/setule.ll

```
; RUN: llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -O3 %s -o - | FileCheck
↪ %s -check-prefix=cpu032I
; RUN: llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic -O3 %s -o - | FileCheck
↪ %s -check-prefix=cpu032II

@j = global i32 5, align 4
@k = global i32 10, align 4
@l = global i32 20, align 4
@m = global i32 10, align 4
@r1 = common global i32 0, align 4
@r2 = common global i32 0, align 4
@r3 = common global i32 0, align 4

define void @test() nounwind {
entry:
    %0 = load i32, i32* @j, align 4
    %1 = load i32, i32* @k, align 4
    %cmp = icmp ule i32 %0, %1
    %conv = zext i1 %cmp to i32
    store i32 %conv, i32* @r1, align 4
; cpu032I: cmp $sw, ${{[0-9]+|t9}}, ${{[0-9]+|t9}}
; cpu032I: andi      $[[T1:[0-9]+|t9]], $sw, 1
; cpu032I: xori      ${{[0-9]+|t9}}, $[[T1]], 1
```

(continues on next page)

(continued from previous page)

```
; cpu032II: sltu      ${[T0:[0-9]+|t9]}, ${{[0-9]+|t9}}, ${{{[0-9]+|t9}}}
; cpu032II: xori      ${{{[0-9]+|t9}}}, ${[[T0]]}, 1
%2 = load i32, i32* @m, align 4
%cmp1 = icmp ule i32 %2, %1
%conv2 = zext i1 %cmp1 to i32
store i32 %conv2, i32* @r2, align 4
ret void
}
```

Running regression tests must occur after building LLVM. However, the following README.rst and changes in related config files allow you to set up *llvm-lit* for a pre-built or pre-installed LLVM environment.

This setup enables running *llvm-lit* without rebuilding LLVM, making the regression testing more efficient when testing changes to the backend only.

Ibdex/set-llvm-lit/README.txt

llvm-lit can be changed to support the installed clang/llvm. Then *llvm-lit* can be run directly without build the clang/llvm.

```
## Run llvm-lit without building llvm-project

$ pwd
$ $HOME/test/llvm/llvm/test/CodeGen/Cpu0
// change the following to dir for your llvm-project
$ export LLVM_DIR=$HOME/test/llvm
$ export LLVM_INSTALLED_DIR=$HOME/llvm-installed
$ ~/llvm-installed/bin/llvm-lit addc.ll -a
$ ~/llvm-installed/bin/llvm-lit .
~/test/llvm/clang/test/CodeGen/Cpu0$ ~/riscv/riscv_newlib/bin/llvm-lit . -a
```

set-llvm-lit % diff -r origin modify &> set-llvm-lit.diff

Ibdex/set-llvm-lit/set-llvm-lit.diff

```
diff -r origin/bin/llvm-lit modify/bin/llvm-lit
8c8
< install_dir = r'/usr/local/riscv/andes/riscv_newlib'
---
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
diff -r origin/test/Unit/lit.site.cfg.py modify/test/Unit/lit.site.cfg.py
3a4,9
> import os
>
> llvm_project_dir = os.environ["PHOENIX_LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
```

(continues on next page)

(continued from previous page)

```
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,14c19,21
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build./bin")
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"./bin")
17c24
< config.shlibdir = path(r"/Users/cschen/llvm/test/build./lib")
---
> config.shlibdir = path(install_dir)+path(r"./lib")
23a31
>     config.shlibdir = config.shlibdir % lit_config.params
diff -r origin/test/lit.site.cfg.py modify/test/lit.site.cfg.py
1c1
< # Autogenerated from /Users/cschen/llvm/test/llvm/test/lit.site.cfg.py.in
---
> # Autogenerated from /u/jonathanchen/andes/riscv/Phoenix/llvm-project/llvm/test/lit.
-> site.cfg.py.in
3a4,9
> import os
>
> llvm_project_dir = os.environ["PHOENIX_LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = r'/usr/local/riscv/andes/riscv_newlib'
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,19c19,27
< config.host_triple = "arm64-apple-darwin20.6.0"
< config.target_triple = "cpu0-unknown-elf"
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build./bin")
< config.llvm_lib_dir = path(r"/Users/cschen/llvm/test/build./lib")
< config.llvm_shlib_dir = path(r"/Users/cschen/llvm/test/build./lib")
< config.llvm_shlib_ext = ".dylib"
---
> config.host_triple = "x86_64-unknown-linux-gnu"
> config.target_triple = "riscv64-unknown-elf"
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"./bin")
```

(continues on next page)

(continued from previous page)

```

> config.llvm_lib_dir = path(install_dir)+path(r"../lib")
> config.llvm_shlib_dir = path(install_dir)+path(r"../lib")
> config.llvm_shlib_ext = ".so"
> config.llvm_plugin_ext = ".so"
22,24c30,34
< config.python_executable = "/opt/homebrew/Frameworks/Python.framework/Versions/3.9/
↪bin/python3.9"
< config.gold_executable = "/opt/homebrew/bin/riscv64-unknown-elf-ld"
< config.ld64_executable = "/Applications/Xcode.app/Contents/Developer/Toolchains/
↪XcodeDefault.xctoolchain/usr/bin/ld"
---

> config.errc_messages = "No such file or directory;Is a directory;Invalid argument;
↪Permission denied"
> config.python_executable = "/usr/bin/python3.6"
> config.gold_executable = "/usr/bin/ld.gold"
> config.ld64_executable = ""
> config.osx_sysroot = path(r"")
33,34c43,44
< config.targets_to_build = " Cpu0"
< config.native_target = "AArch64"
---

> config.targets_to_build = " RISCV"
> config.native_target = "X86"
36,38c46,48
< config.host_os = "Darwin"
< config.host_cc = "/opt/homebrew/opt/llvm/bin/clang "
< config.host_cxx = "/opt/homebrew/opt/llvm/bin/clang++ "
---

> config.host_os = "Linux"
> config.host_cc = "/usr/bin/cc "
> config.host_cxx = "/usr/bin/c++ "
44,45c54,55
< config.have_libxar = 1
< config.have_libxml2 = 1
---

> config.have_libxar = 0
> config.have_libxml2 = 0
52,53c62,63
< config.llvm_host_triple = 'arm64-apple-darwin20.6.0'
< config.host_arch = "arm64"
---

> config.llvm_host_triple = 'x86_64-unknown-linux-gnu'
> config.host_arch = "x86_64"
65a76
>     config.llvm_lib_dir = config.llvm_lib_dir % lit_config.params
diff -r origin/tools/clang/test/Unit/lit.site.cfg.py modify/tools/clang/test/Unit/lit.
↪site.cfg.py
3a4,9
> import os
>
> llvm_project_dir = os.environ["LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)

```

(continues on next page)

(continued from previous page)

```
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,15c19,22
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build./bin")
< config.llvm_libs_dir = path(r"/Users/cschen/llvm/test/build./lib")
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"./bin")
> config.llvm_libs_dir = path(install_dir)+path(r"./lib")
17c24
< config.clang_obj_root = path(r"/Users/cschen/llvm/test/build/tools/clang")
---
> config.clang_obj_root = path(install_dir)+path(r"/tools/clang")
19,20c26,27
< config.shlibdir = path(r"/Users/cschen/llvm/test/build./lib")
< config.target_triple = "cpu0-unknown-elf"
---
> config.shlibdir = path(install_dir)+path(r"./lib")
> config.target_triple = "riscv64-unknown-elf"
28a36
>     config.shlibdir = config.shlibdir % lit_config.params
36c44
<     config, os.path.join(path(r"/Users/cschen/llvm/test/clang"), "test/Unit/lit.cfg.
˓→py"))
---
>     config, os.path.join(path(llvm_project_dir)+path(r"/clang"), "test/Unit/lit.cfg.
˓→py"))
diff -r origin/tools/clang/test/lit.site.cfg.py modify/tools/clang/test/lit.site.cfg.
˓→py
3a4,9
> import os
>
> llvm_project_dir = os.environ["LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,17c19,24
```

(continues on next page)

(continued from previous page)

```

< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build/.bin")
< config.llvm_libs_dir = path(r"/Users/cschen/llvm/test/build/.lib")
< config.llvm_shlib_dir = path(r"/Users/cschen/llvm/test/build/.lib")
< config.llvm_plugin_ext = ".dylib"
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"/.bin")
> config.llvm_libs_dir = path(install_dir)+path(r"/.lib")
> config.llvm_shlib_dir = path(install_dir)+path(r"/.lib")
> config.llvm_plugin_ext = ".so"
19,24c26,35
< config.clang_obj_root = path(r"/Users/cschen/llvm/test/build/tools/clang")
< config.clang_src_dir = path(r"/Users/cschen/llvm/test/clang")
< config.clang_tools_dir = path(r"/Users/cschen/llvm/test/build/.bin")
< config.host_triple = "arm64-apple-darwin20.6.0"
< config.target_triple = "cpu0-unknown-elf"
< config.host_cxx = "/opt/homebrew/opt/llvm/bin/clang++"
---
> config.errc_messages = "No such file or directory; Is a directory; Invalid argument;
  ↪Permission denied"
> config.clang_lit_site_cfg = __file__
> config.clang_obj_root = path(install_dir)+path(r"/tools/clang")
> config.clang_src_dir = path(llvm_project_dir)+path(r"/clang")
> config.clang_tools_dir = path(install_dir)+path(r"/.bin")
> config.clang_lib_dir = path(install_dir)+path(r"/lib")
> config.host_triple = "x86_64-unknown-linux-gnu"
> config.target_triple = "riscv64-unknown-elf"
> config.host_cc = "/usr/bin/cc"
> config.host_cxx = "/usr/bin/c++"
34c45
< config.enable_experimental_new_pass_manager = 0
---
> config.enable_experimental_new_pass_manager = 1
36,37c47,48
< config.host_arch = "arm64"
< config.python_executable = "/opt/homebrew/Frameworks/Python.framework/Versions/3.9/
  ↪bin/python3.9"
---
> config.host_arch = "x86_64"
> config.python_executable = "/usr/bin/python3.6"
40a52
> config.llvm_external_lit = path(r"""
diff -r origin/utils/lit/lit.site.cfg modify/utils/lit/lit.site.cfg
3a4,9
> import os
>
> llvm_project_dir = os.environ["PHOENIX_LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]

```

(continues on next page)

(continued from previous page)

```
>
4a11
>     from pathlib import Path
7c14
<         return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
13,15c20,22
< config.llvm_src_root = "/Users/cschen/llvm/test/llvm"
< config.llvm_obj_root = "/Users/cschen/llvm/test/build"
< config.llvm_tools_dir = "/Users/cschen/llvm/test/build./bin"
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"./bin")
30c37
< lit_config.load_config(config, "/Users/cschen/llvm/test/build/utils/lit/tests/lit.
˓→cfg")
---
> lit_config.load_config(config, os.path.join(config.llvm_obj_root, "utils/lit/tests/
˓→lit.cfg"))
```

- Only *tools/clang/test/lit.site.cfg.py* and *test/lit.site.cfg.py* need to be modified.
- The other files, *tools/clang/test/Unit/lit.site.cfg.py*, *test/Unit/lit.site.cfg.py*, and *utils/lit/tests/lit.site.cfg.in*, are empty and not used. However, I modified them as well for completeness.

CHAPTER
SEVENTEEN

**APPENDIX C: THE CONCEPT OF NPU (NEURAL PROCESSOR UNIT)
COMPILER**

- *Deep Learning Theory*
 - *Deep Learning*
 - *CNN*
 - *Book*
- *Deep learning compiler*
 - *Survey*
 - *GPU*
 - *NNEF*
 - *OpenVX*
 - * *SYCL*
 - *Tools*
- *NPU compiler*
 - *Abstract*
 - *MLIR and IREE*
 - *Tensorflow*
 - *mlir to onnx*
 - *Support tensorflow*
 - *ONNC*
- *LLVM based NPU compiler*
 - *llvm IR for NPU compiler*
 - *Open source project*
- *ONNX*
 - *viewer*
 - *install*
 - *onnx api*

17.1 Deep Learning Theory

17.1.1 Deep Learning

Hung-Yi Lee's video¹.

17.1.2 CNN

CNN: They have applications in image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, and financial time series⁴.

¹ https://www.youtube.com/watch?v=CXgbekl66jc&list=PLJV_el3uVTsPy9oCRY30oBPNLCo89yu49

⁴ https://en.wikipedia.org/wiki/Convolutional_neural_network

Concept about how to apply Convolution and MaxPool to getting features from image². Conv+MaxPool -> get features map and downsize image, more Conv+MaxPool can filter image and higher level of features and downsize more image. CNN model used in image recognition.

Concept and data applying in Deep Learning for different models of CNN³.

17.1.3 Book

<http://www.deeplearningbook.org/> NTU: Hung-yi Lee MLDS (Machine Learning Deep + Structure approaches), the trend for future implementation.

17.2 Deep learning compiler

17.2.1 Survey

The current open source compilers for deep learning⁵ as follows,

TVM compiler open source infrastructure supports most of DL Frameworks such as TensorFlow, PyTorch..., and generating CUDA/OpenCL/OpenGL for gpu and LLVM for cpu.

17.2.2 GPU

The NVIDIA CUDA Toolkit provides a development environment for creating high-performance GPU-accelerated applications. GPU-accelerated CUDA libraries enable acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics⁶.

OpenCL runs on AMD GPUs and provides partial support for TensorFlow and PyTorch. If you want to develop new networks some details might be missing, which could prevent you from implementing the features you need⁶. For instance, if ARM GPU doesn't implement operation "Cosh" on TVM while a DL model created from PyTorch generate "Cosh" operation, then it will fail to run the DL model though TVM compile PyTorch model into OpenCL. Once "Cosh" is implemented with kernel function "Cosh" in OpenCL by calling GPU's instructions, it can be fixed.

17.2.3 NNEF

Neural Network Exchange Format.

NPU or GPU can apply NNEF to support all AI models.

The goal of NNEF is to enable data scientists and engineers to easily transfer trained networks from their chosen training framework into a wide variety of inference engines. A stable, flexible and extensible standard that equipment manufacturers can rely on is critical for the widespread deployment of neural networks onto edge devices, and so NNEF encapsulates a complete description of the structure, operations and parameters of a trained neural network, independent of the training tools used to produce it and the inference engine used to execute it⁷.

ONNX and NNEF are Complementary ONNX moves quickly to track authoring framework updates NNEF provides a stable bridge from training into edge inferencing engines⁷.

ONNX import/export to NNEF, should edge NPU use NNEF⁷?

² <http://violin-tao.blogspot.com/2017/07/ml-convolutional-neural-network-cnn.html>

³ <https://github.com/onnx/models>

⁵ <https://arxiv.org/pdf/2002.03794.pdf>

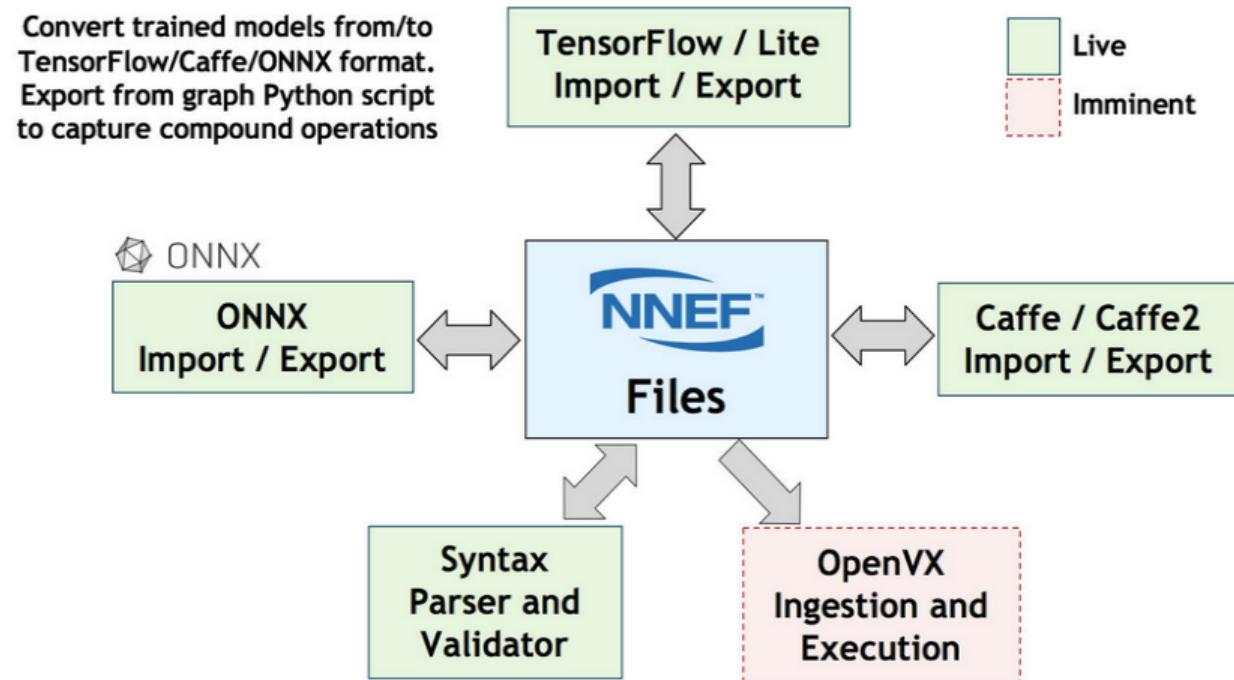
⁶ <https://missinglink.ai/guides/tensorflow/tensorflow-support-opencl/>

⁷ <https://www.khronos.org/nnef>

Table 1. The detailed comparison of popular DL compilers.

	TVM	TC	Glow	nGraph+PlaidML	XLA
Core/Programming Language					
Core Programming	C++	C++	C++	C++	C++
Supported Hardware Targets					
CPU	✓	✓	✓	✓	✓
NVIDIA-GPU	✓	✓	✓	✓	✓
AMD-GPU	✓	✗	✓	✓	✓
FPGA	✓	✗	✗	✗	✗
TPU	✗	✗	✗	✗	✓
NNP	✗	✗	✗	✓	✗
Customed	✓	✗	✓	✓	✓
Supported DL Frameworks					
TensorFlow	✓	✗	✗	✓	✓
PyTorch	✓	✓	✓	✗	✓
MXNet	✓	✗	✗	✗ not active	✗
Caffe2	✓	✓	✓	✗	✗
ONNX	✓	✗	✓	✓	✗
CoreML	✓	✗	✗	✗	✗
Keras	✓	✗	✗	✓	✓
PaddlePaddle	✗	✗	✗	✓	✗
DarkNet	✓	✗	✗	✗	✗
Supported Generating Languages					
CUDA	✓	✓	✗	✗	✓
OpenCL	✓	✗	✓	✓	✓
Metal	✓	✗	✗	✓	✗
LLVM	✓	✓	✓	✓	✓
OpenGL	✓	✗	✗	✓	✗
Supported Features/Strategies					
AOT	✓	✗	✓	✓ official release	✓
JIT	✓	✓	✓	✓	✓
Training	—	✓	✓	✓	✓
Quantization	—	✗	✓	✓	✗
Automatic Differentiation	—	✓	✓	✓	✗
Dynamic Shape	✓	✗	✗	✓	✗
Auto-tuning	✓	✓	✗	✓ only tiling	✗

NNEF Tools Ecosystem



NNEF open source projects hosted on Khronos

[NNEF GitHub repository](#) under Apache 2.0

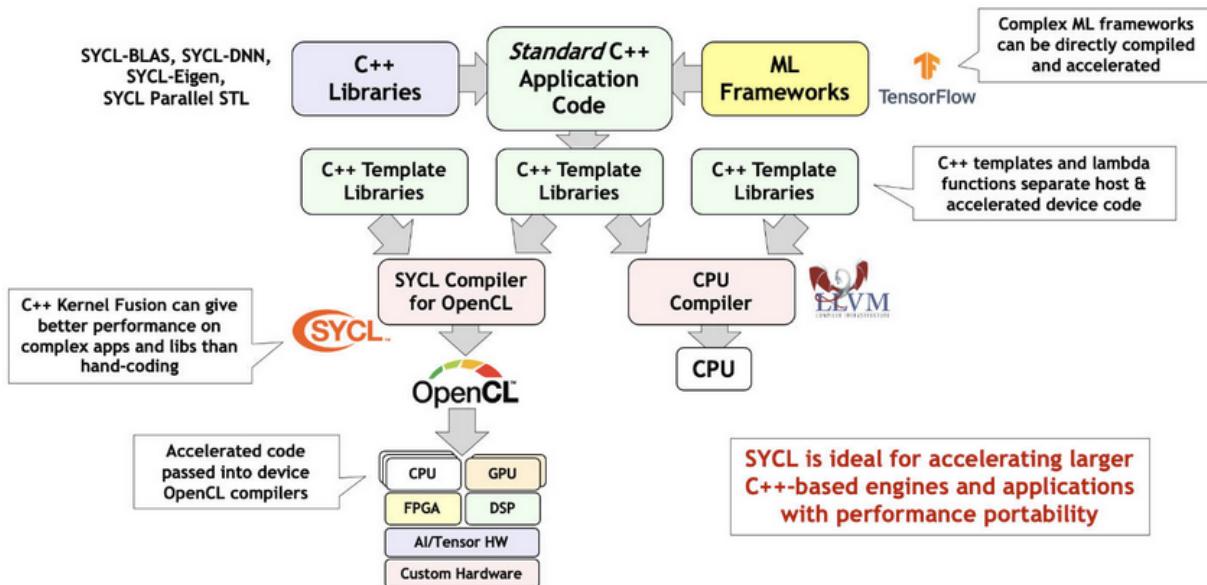
17.2.4 OpenVX

OpenVX enables the graph to be extended to include hardware architectures that don't support programmable APIs⁸.

It is designed by the Khronos Group to facilitate portable, optimized and power-efficient processing of methods for vision algorithms⁹. Nothing about NPU I think.

SYCL

User use SYCL or DSL compile domain language into SYCL and run on OpenCL hardwares¹⁰. An example here¹¹.



https://github.com/Jonathan2251/nc/OpenCL_SYCL

17.2.5 Tools

Create onnx test file¹².

17.3 NPU compiler

17.3.1 Abstract

Tensorflow support unknown shape¹⁴. Though our npu support kernel call where kernel call is a set of commands to npu to deal shape at run time, it is unefficiency. As I remember mlt supports binding shape for unknown at compile-time but not always work. Luckily, we can customilze by redefining model to binding shape statically [20200412]

⁸ <https://www.khronos.org/openvx>

⁹ <https://en.wikipedia.org/wiki/OpenVX>

¹⁰ <https://www.khronos.org/sycl>

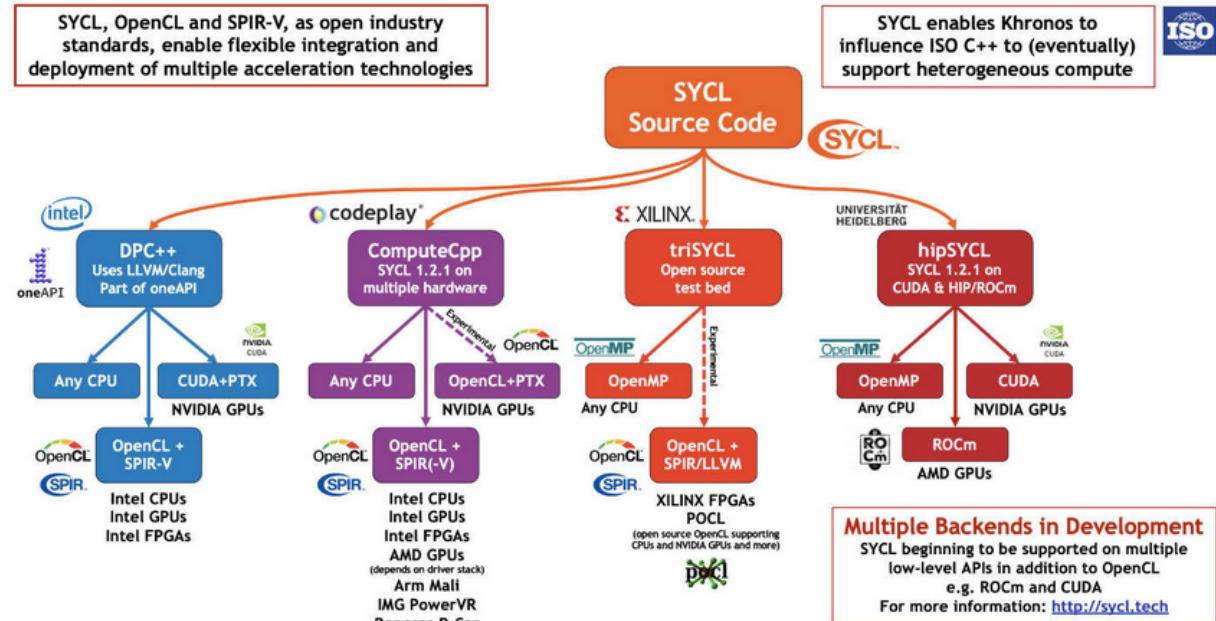
¹¹ <https://en.wikipedia.org/wiki/SYCL>

¹² <https://www.youtube.com/watch?v=QZQwmZTfLmI>

¹⁴ <https://pgaleone.eu/tensorflow/2018/07/28/understanding-tensorflow-tensors-shape-static-dynamic/>

SYCL Implementations

SYCL implementations are available from an increasing number of vendors, including adding support for diverse acceleration API back-ends in addition to OpenCL.



17.3.2 MLIR and IREE

IREE (Intermediate Representation Execution Environment, pronounced as “eerie”) is an MLIR-based end-to-end compiler that lowers ML models to a unified IR optimized for real-time mobile/edge inference against heterogeneous hardware accelerators. IREE also provides flexible deployment solutions for the compiled ML models¹³ as the following figure.

- HAL IR: Vulkan-like allocation and execution model encoding -> on-line first-time compilation and save in cache. Executable compilation via architecture specific backend compiler plugins.
- VM IR: Dynamic module linkage definitions (imports, exports, globals, etc)¹⁵.

The purpose of mlir is:

- Connect cpu with mlir-to-llvm-ir.

17.3.3 Tensorflow

The mechanism of Mlir and iree applied on tensorflow as the figure above section is not fitted for off-line edge npu that stand alone without server-connection for tuning weight of face detection’s purpose. It is designed for on-line server-connected npu. The gpu of supporting spirv is best candidate until this date 2020/5/12.

At beginning, tensorflow rely on api without fixed format such as ONNX¹⁶. As a result ONNX emerged and adopted for most of npu in their private backend compiler. Google does not like to hire onnx as the format for npu backend compiler onnx-mlir project¹⁷ which convert onnx to mlir dialect is sponsored by Google I guess¹⁸ for encouraging new npu compiler

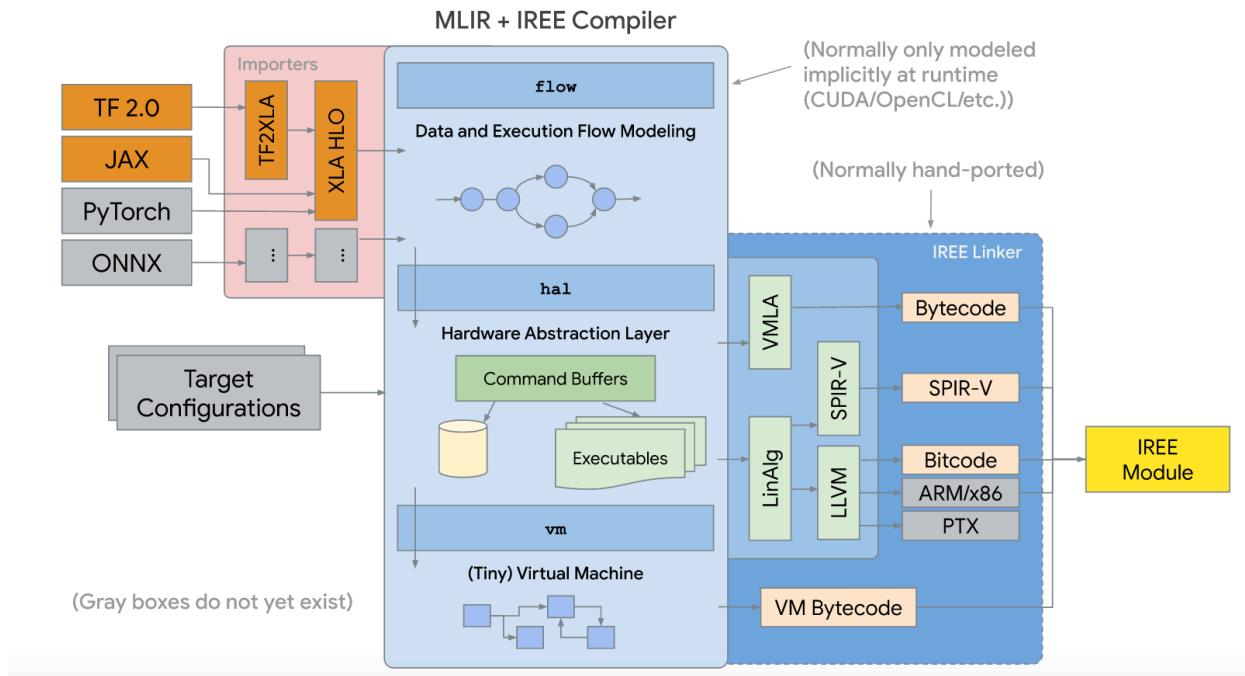
¹³ <https://github.com/google/iree>

¹⁵ Page 15 of https://docs.google.com/presentation/d/1RCQ4ZPQFK9cVgu3IH1e5xbrBcqy7d_cEZ578j84OvYI/edit#slide=id.g6e31674683_0_23101

¹⁶ Actually onnx format based on IO api with protobuf. It has real binary format but may change from version to version. Tensorflow api has no real binary format.

¹⁷ <https://github.com/onnx/onnx-mlir>

¹⁸ <https://groups.google.com/a/tensorflow.org/forum/#topic/mlir/2FT4sD8kqTY>



development hiring mlir as their compiler input (convert onnx to mlir then handling mlir input).

With mlir and iree appear on tensorflow as a series of fixed formats in tensorflow as section above. The hardware vendors for cloud server AI machine with heterogeneous hardware accelerators will run tensorflow system by supporting mlir/iree input format in their compilers more and more. So, it is unavoidable that tensorflow system's npu vendors have to support mlir/iree input format beyond onnx. Or open source software or vendor software appear to do transfer from mlir/iree to onnx. (python in tensorflow api allow unknown type and shape size, so it cannot transer python api to onnx fully).

If lucky, google may hire onnx. Because onnx format is older than mlir in history. In addition in aspect of format, mlir has multi-level multi-dialect and more complicate while onnx is easy and better to understand (P.S. I don't dig into mlir yet). Many AI models has supported onnx file format. For some AI model's formats that run on tensorflow without supporting onnx, apply tensorflow-onnx open source project¹⁹ can convert tensorflow to onnx partly.

Onnx alliance may release some programs for transferring mlir to onnx for fighting agiant mlir-iree growing in npu compiler but not at this moment.

For off-line edge npu that stand alone without server-connection for tunning weight of face detection's purpose, suprting mlir-iree compiler may not necessary.

17.3.4 mlir to onnx

<https://www.tensorflow.org/mlir>

<https://mlir.llvm.org/talks/>

https://llvm.org/devmtg/2019-04/talks.html#Tutorial_1

- 3 ppt in llvm tutorials

<https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf>

build mlir: https://mlir.llvm.org/getting_started/

¹⁹ <https://github.com/onnx/tensorflow-onnx>

```
~/llvm/1/llvm-project/build$ cmake -G Ninja .. llvm \
>     -DLLVM_ENABLE_PROJECTS=mlir \
>     -DLLVM_BUILD_EXAMPLES=ON \
>     -DLLVM_TARGETS_TO_BUILD="X86;NVPTX;AMDGPU" \
>     -DCMAKE_BUILD_TYPE=Release \
>     -DLLVM_ENABLE_ASSERTIONS=ON

~/llvm/1/llvm-project/build$ cmake --build . --target check-mlir
[200/1919] Generating VCSRevision.h
-- Found Git: /usr/bin/git (found version "2.17.1")
[1604/1919] Building CXX object tools/mlir/tools/mlir-linalg-ods-gen/CMakeFiles/mlir-
↳ linalg-ods-gen.dir/mlir-linalg-ods-gen.cpp.o
/home/cschen/llvm/1/llvm-project/mlir/tools/mlir-linalg-ods-gen/mlir-linalg-ods-gen.
↳ cpp:935:6: warning: 'bool {anonymous}::Expression::operator==(const {anonymous}
↳ ::Expression&) const' defined but not used [-Wunused-function]
bool Expression::operator==(const Expression &e) const {
    ~~~~~
[1918/1919] Running the MLIR regression tests

Testing Time: 9.88s
Unsupported Tests: 16
Expected Passes : 465
```

run: <https://mlir.llvm.org/docs/Tutorials/Toy>

```
~/llvm/1/llvm-project/mlir/test/Examples/Toy/Ch1$ ~/llvm/1/llvm-project/build/bin/
↳ toyc-ch1 ast.toy -emit=ast
...
~/llvm/1/llvm-project/mlir/test/Examples/Toy/Ch1$ ~/llvm/1/llvm-project/build/bin/
↳ toyc-ch1 ast.toy -emit=ast 2>&1 | ~/llvm/1/llvm-project/build/bin/FileCheck ast.toy
~/llvm/1/llvm-project/mlir/test/Examples/Toy/Ch1$ ~/llvm/1/llvm-project/build/bin/
↳ llvm-lit ast.toy
-- Testing: 1 tests, 1 workers --
PASS: MLIR :: Examples/Toy/Ch1/ast.toy (1 of 1)

Testing Time: 0.11s
Expected Passes: 1
```

The result I run is based on git commit 455ccde1377b3ec32d321eb7c38808fecdf230a8 Date: Sun May 17 21:00:09 2020 -0400

17.3.5 Support tensorflow

Question:

Sean,

As I said, we can always redefine AI model to remove unknown type or dimension at ahead of time compilation to fit static compilation binding, and my AI input models are CNN without loop (it is DAG form). For this kind of models on tensorflow, can it be translated absolutely to mlir form based on what you know? If it can, then I can write converting program for mlir to my npu internal ir to support tensorflow.

Answer:

For programs with those restrictions, converting to MLIR xla_hlo dialect is always possible.

Note that it is always possible to convert a TensorFlow GraphDef into MLIR tensorflow dialect. MLIR is very flexible. But MLIR tensorflow dialect is too general for NPU and needs to be converted to MLIR xla_hlo dialect.

—Sean Silva

Sean,

Thank you! I am going to pass this information to my boss. We don't study mlir yet. I believe it will take effort and we only have few engineers on compiler taking a lot of works. There other resource such as tensorflow-onnx but only part of supporting tensorflow to onnx converting.

Jonathan

17.3.6 ONNC

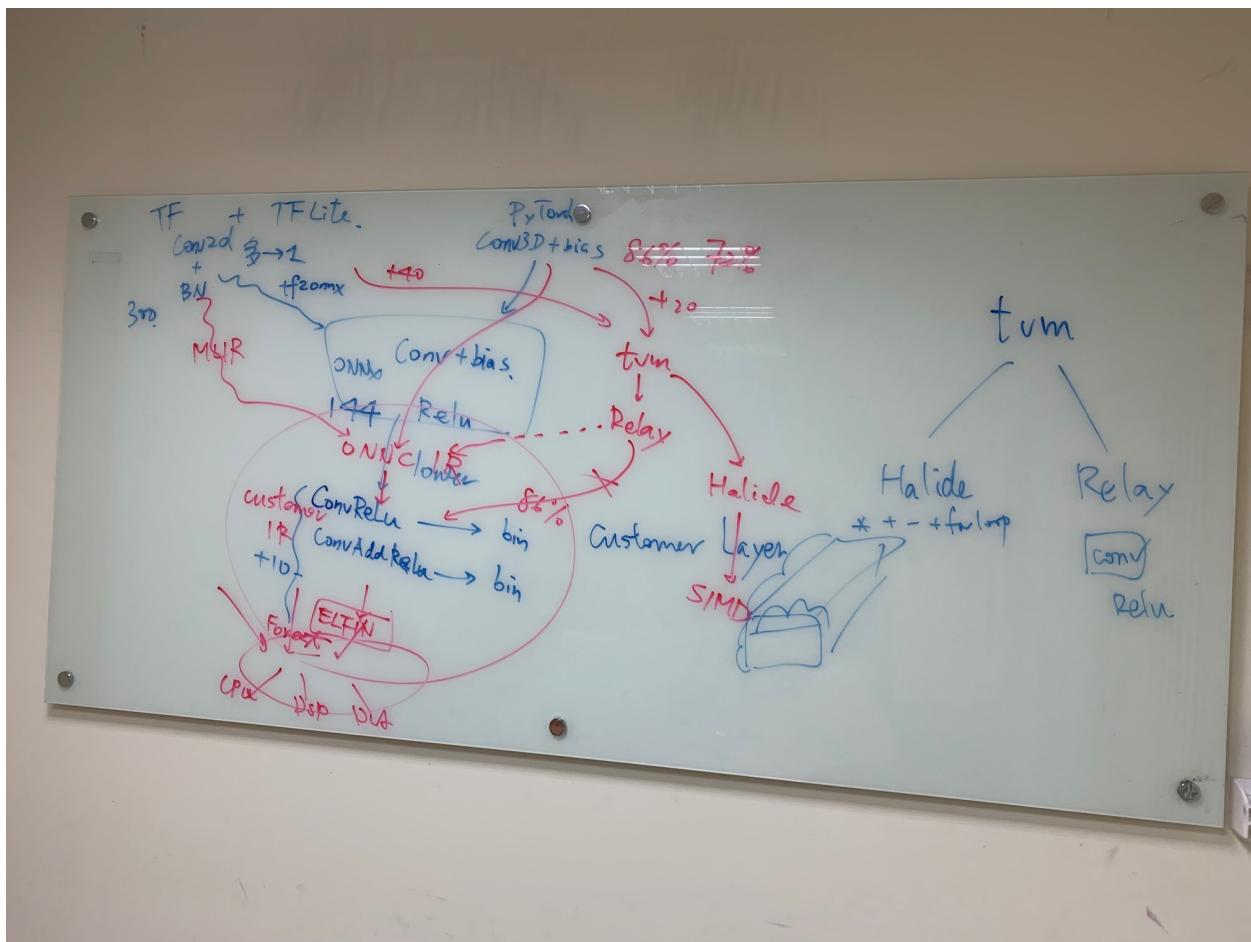


Fig. 17.1: ONNC

- Private IR is better than llvm intrinsic IR for non-VLIW (SIMD or MIMD). Stride, memory dependences, ..., llvm has not much advantages in this. Private IR is better.
- Support MLIR to private IR in Novemember. Open source tensorflow to onnx has limited operations support only, bad and not good.
- TVM support python interfaces but from TVM -> relay is not good according their experience. From MLIR is better.
- Support MLIR, pytorch, caffe are enough. Future has less AI model tools.

- Run time scheduling customer support.
- tf-onnx is not sufficient to support tf's operators and bad. So, translate tf through MLIR to ONNC to customer IR is must.

<https://onnc.ai/>

17.4 LLVM based NPU compiler

Use LLVM rather than GCC because TVM open source compiler generating llvm-ir for X86 and ARM and may extend to support other CPU in future. Though TVM may uses BYOC to generate C function calling NPU's builtin function, the AutoTVM layer allows doing more optimization code generation such as vectorization, CPU/NPU instructions interleaving, ..., etc²³²⁴. However in cloud of DL scenario, since the time of data transfer from global DRAM to PE's (Processor Entity) local memory SRAM is unknown until run time, applying BYOC for calling NPU's builtin function then using GCC instead of llvm is possilbe. NPU usually implements data parallel instructions such as matrix multiplication, convolution, relu, pool, ..., to speed up the Deep Learning Operations. For other operations not very data parallel such as global pool, concat, sort, ..., may leave to CPU finishing them. TVM output llvm-ir rather than GCC since GCC community never had desire to enable any tools besides compiler (Richard Stallman resisted attempts to make IR more reusable to prevent third-party commercial tools from reusing GCC's frontends)²⁵.

The way for supporting llvm based NPU compiler is to implement builtin functions in clang and the corresponding specific NPU's llvm-intrinsic functions in llvm backend. For instance, the matrix multiplication operations of clang/llvm support as the following table.

Table 17.1: Matrix Multiplication defined in clang, llvm and ASM

Component	Function/IR/Instruction
clang's builtin	__builtin_tensor_matmul(A, B, C)
llvm's intrinsic	@llvm.lt.matmul %A, %B, %C
NPU's ASM instruction	matmul \$A, \$B, %C

The detail steps to support clang's builtin and llvm's intrinsic function for backend are in my books²⁶²⁷.

17.4.1 llvm IR for NPU compiler

Though npu has no general purpose registers GPR, it is possible to apply llvm ir for npu to do codegen by llvm as follows,

```
@x1 = global [1 x [3 x [120 x [120 x float]]]], align 4
@w1 = global [64 x [3 x [7 x [7 x float]]]], align 4
@conv = @llvm.npu1.conv float* @x, float* @weight, ...
```

Conclusion:

1. No GPRs in NPU but can get advantage of code-gen by llvm-tblgen tool.
2. The vector size of llvm is power of 2 (1, 2, 4, 8, ...). But it can be achieved by modifying llvm kernel source data type.
- ref. code/llvm-ex1.c
3. Though NPU has no GPRs, the memory allocation can be done by adjust instructions order and split instructions (if over NPU's memory) in passes of LLVM IR level.

²³ <https://discuss.tvm.apache.org/t/how-to-see-different-ir-relay-te-tir-from-my-own-pytorch-model/13684>

²⁴ <https://discuss.tvm.apache.org/t/which-is-the-best-way-to-port-tvm-to-a-new-ai-accelerator/6905>

²⁵ <https://stackoverflow.com/questions/40799696/how-is-gcc-ir-different-from-llvm-ir/40802063>

²⁶ <http://jonathan2251.github.io/lbt/clang.html#builtin-functions>

²⁷ <http://jonathan2251.github.io/lbd/funccall.html#add-specific-backend-intrinsic-function>

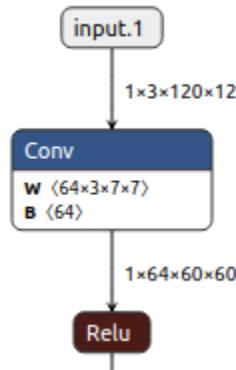


Fig. 17.2: Conv operation in onnx file

reference:

- section 5.2.2 Code Generation based on Low-Level IR. The low-level IR adopted by most DL compilers can be eventually lowered to LLVM IR, and benefits from LLVM’s mature optimizer and code generator^{Page 817, 5.}

17.4.2 Open source project

- onnx to mlir dialect: <https://github.com/onnx/onnx-mlir>
- tensorflow to onnx: <https://github.com/onnx/tensorflow-onnx>
- onnx to tensorflow: <https://github.com/onnx/onnx-tensorflow>

17.5 ONNX

17.5.1 viewer

- Web for opening onnx <https://lutzroeder.github.io/netron/>
- Application tool for opening onnx <https://github.com/lutzroeder/netron> // find “Browser: Start” in this page

Netron app in ubuntu.

17.5.2 install

\$ pip install onnx

17.5.3 onnx api

- copy onnx file: ex. nc/code/copy_onnx.py reference here²⁰.
- create onnx file: ex. nc/code/create1.py reference here²¹.
- Kneron onnx creating tool²².

²⁰ <https://github.com/onnx/onnx/issues/2052>

²¹ https://www.google.com/search?client=ubuntu&hs=bS9&channel=fs&sxsrf=ALeKk00IITG3Dj_IryeytZ_iTJE3PsMA%

3A1597217944046&ei=mJwzX8KZAuiLr7wPr_aq0AU&q=onnx+python+api&oq=onnx+python+api&gs_lcp=CgZwc3ktYWIQAzIECCMQJzIGCAAAQCBAsOggIBAB

sclient=psy-ab&ved=0ahUKEwjCxbvBlJXrAhXoxYsBHS-7CloQ4dUDCAs&uact=5

²² https://github.com/kneron/ONNX_Convertor/blob/master/optimizer_scripts/onnx2onnx.py

CHAPTER
EIGHTEEN

TODO LIST

RESOURCES

19.1 Build steps

<https://github.com/Jonathan2251/lbd/blob/master/README.md>

19.2 Book example code

The example code lbdex.tar.gz is available in:

<http://jonathan2251.github.io/lbd/lbdex.tar.gz>

19.3 Alternate formats

The book is also available in the following formats:

19.4 Presentation files

19.5 Search this website

- search