Write An LLVM Backend Tutorial For Cpu0

Release 3.1.1

Chen Chung-Shu gamma_chen@yahoo.com.tw
Anoushe Jamshidi ajamshidi@gmail.com

December 04, 2012

CONTENTS

1 About **1**
- 1.1 Authors . . . 1
- 1.2 Revision history . . . 1
- 1.3 Licensing . . . 1
- 1.4 Preface . . . 1
- 1.5 Prerequisites . . . 2

2 Install llvm and Cpu0 example code **3**
- 2.1 Install LLVM, Xcode and cmake on iMac . . . 3
- 2.2 Create LLVM.xcodeproj by cmake Graphic UI . . . 4
- 2.3 Build llvm by Xcode . . . 4
- 2.4 Create LLVM.xcodeproj of supporting cpu0 by terminal `cmake` command . . . 7
- 2.5 Install other tools on iMac . . . 7
- 2.6 Install LLVM 3.1 release build on Linux . . . 16
- 2.7 Install cpu0 debug build on Linux . . . 18

3 Cpu0 Instruction and LLVM Target Description **25**
- 3.1 CPU0 processor architecture . . . 25
- 3.2 LLVM structure . . . 27
- 3.3 Target Description td . . . 33
- 3.4 Write td (Target Description) . . . 33
- 3.5 Write cmake file . . . 42
- 3.6 Target Registration . . . 44
- 3.7 Build libraries and td . . . 44

4 Back end structure **47**
- 4.1 TargetMachine structure . . . 47
- 4.2 Add RegisterInfo . . . 53
- 4.3 Add AsmPrinter . . . 54
- 4.4 LLVM Code Generation Sequence . . . 56
- 4.5 DAG (Directed Acyclic Graph) . . . 58
- 4.6 Instruction Selection . . . 59
- 4.7 Add Cpu0DAGToDAGISel class . . . 61
- 4.8 Add Prologue/Epilogue functions . . . 62
- 4.9 Summary of Chapter 3 . . . 65

5 Todo List **67**

6 Alternate formats **69**

CHAPTER

ONE

ABOUT

1.1 Authors

陳鍾樞

Chen Chung-Shu gamma_chen@yahoo.com.tw

Anoushe Jamshidi ajamshidi@gmail.com

1.2 Revision history

Version 1, Released Chapter 1, 2, 3

Version 2, Released February 4, 2012 Added Chapter 0, Section 3.3 Correct some English & typing errors in book

Version 3, Released February 19, 2012 Shift Chapter 0..2 to Chapter 1..3; Move Section 3.1, 3.2 to 4.1, 4.2; Move Section 3.3 to 5.1 Added Section 5.2 to 5.6; Added Chapter 6; Added Section 7.1 to 7.4 Added first paragraph in Chapter 1; Added Section" 2.1 CPU0 processor architecture" and shift other sections in Chapter 2 Correct some English & typing errors

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 5.5 to 2.1 Correct some English & typing errors

1.3 Licensing

Todo

Add info about LLVM documentation licensing.

1.4 Preface

The LLVM Compiler Infrastructure provides a versatile structure for creating new backends. Creating a new backend should not be too difficult once you familiarize yourself with this structure. However, the available backend documentation is fairly high level and leaves out many details. This tutorial will provide step-by-step instructions to write a new backend for a new target architecture from scratch.

We will use the Cpu0 architecture as an example to build our new backend. Cpu0 is a simple RISC architecture that has been designed for educational purposes. More information about Cpu0, including its instruction set, is available here: http://ccckmit.wikidot.com/ocs:cpu0. The Cpu0 example code referenced in this book can be found in this shared folder on Dropbox. As you progress from one chapter to the next, you will incrementally build the backend's functionality.

This tutorial was written using the LLVM 3.1 Mips backend as a reference. Since Cpu0 is an educational architecture, it is missing some key pieces of documentation needed when developing a compiler, such as an Application Binary Interface (ABI). We implement our backend borrowing information from the Mips ABI as a guide. You may want to familiarize yourself with the relevant parts of the Mips ABI as you progress through this tutorial.

1.5 Prerequisites

Readers should be comfortable with the C++ language and Object-Oriented Programming concepts. LLVM has been developed and implemented in C++, and it is written in a modular way so that various classes can be adapted and reused as often as possible.

Already having conceptual knowledge of how compilers work is a plus, and if you already have implemented compilers in the past you will likely have no trouble following this tutorial. As this tutorial will build up an LLVM backend step-by-step, we will introduce important concepts as necessary.

This tutorial references the following materials. We highly recommend you read these documents to get a deeper understanding of what the tutorial is teaching:

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 (Search for it on Google)

CHAPTER

TWO

INSTALL LLVM AND CPU0 EXAMPLE CODE

Before start, I have one thing hope you to know. In fact, you can work with a backend program, even a new CPU, by porting from the similar existed llvm backend or from the existed llvm supported CPU which you familiar with. In that way, maybe it include redundant code. But it's OK since compiler is running on PC or Laptop which got a lots of memory. It's a style for real working in programming. It's a way to learn, I agree. The advantage of this approach came from the fact of most RISC CPUs have similar instruction set in concept and style. All you need to do is porting from an existing CPU into a similar instruction set of CPU with different OP code. According my experience, this solution has two shortage. First, this approach can make progress better in time at beginning, I agree. But the time will be offset in debug and maintenance. The backend program is a kind of system program which is a low level program and most of the functions is triggered by call back function or override function in C++. LLVM call these backend function at proper time. As you can imagine, it's not easy to track the program execution flow. It's harder to debug compare to the front end program which you can track the program flow easily because front end call llvm interface/function when it like to. And more harder and harder then the ordinary UI or high level application because backend is a system program. Second, by this approach you won't get a full knowledge in llvm backend programming. With full llvm backend knowledge equipment, you will find the backend programming is becoming easier and easier for your daily work, day by day. So, if you are a paid time llvm backend programmer, I suggest you learn llvm backend program from beginning when you got time. Of course, put the learning task on aside when you got scheduled job, and pike up the learning task back when your time is available. It's my opinion. Just do your choice.

I will show you how I install llvm in iMac and Linux in this chapter. Currently, I can build the llvm source code on iMac Xcode and do debug by lldb. I cannot do debug on IDE Xcode at this point, so if you know how to do debug LLVM on Xcode, please let me know by email. I really need that. In Linux, I can build and debug by gdb on Linux Fedora 17, and the same I don't know how to do debug on IDE like Eclips. About `cmake` for llvm please reference Building LLVM with CMake further. The Xcode version I use in this book is 4.5.1, cmake is 2.8.9.

This book is still a work in progress, so sections may be incomplete, but I hope you will enjoy it as it grows.

2.1 Install LLVM, Xcode and cmake on iMac

Please download llvm version 3.1 (llvm, clang, compiler-rf) from llvm release download. Then, `tar -zxvf llvm, clang, compiler-rt`, and change the llvm source code root directory into src. After that, move clang source code to src/tools/clang, move compiler-rt source to src/project/compiler-rt as Fig 1.1.

Next, copy llvm source to /Users/Jonathan/llvm/3.1/src by terminal command `cp -rf /Users/Jonathan/Documents/llvmSrc/src /Users/Jonathan/llvm/3.1/.`.

Install Xcode from App Store as well as cmake from http://www.cmake.org/cmake/resources/software.html. Before install cmake, check the "Anywhere" of "Allow applications download from" of "Security & Privacy" of "System Preferences" as Fig 1.2.

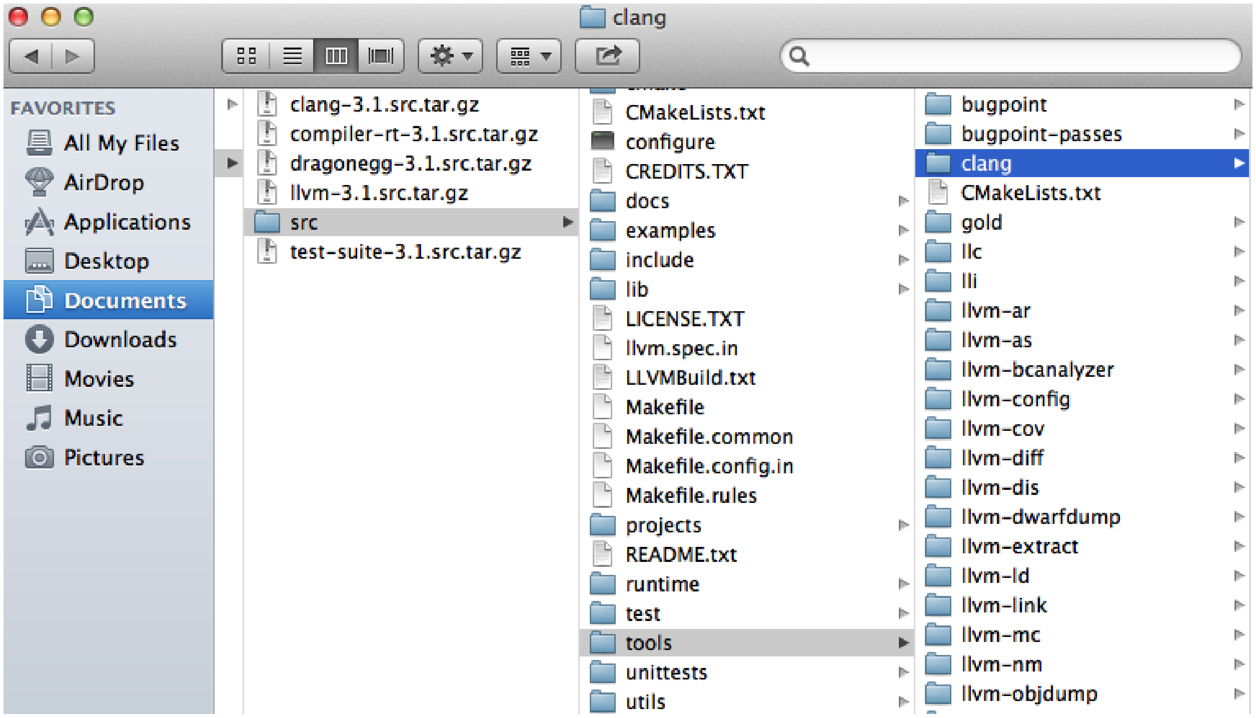


Figure 2.1: Fig 1.1 llvm, clang, compiler-rt source code position on iMac

2.2 Create LLVM.xcodeproj by cmake Graphic UI

Currently, I cannot do debug by lldb with cmake graphic UI operations depicted in this section, but I can do debug by lldb with section “1.4 Create LLVM.xcodeproj of support cpu0 by terminal cmake command”. Even though, let’s build LLVM project with cmake graphic UI now since this LLVM build is to build the release version for clang, llvm-as, llc, ..., execution command use, not for working backend program. First, create LLVM.xcodeproj as Fig 1.3, then click configure button to enter Fig 1.4, and then click Done button on Fig 1.4 to get Fig 1.5.

Click OK from Fig 1.5 and select Cmake 2.8-9.app for CMAKE_INSTALL_NAME_TOOL by click the right side button “...” of that row in Fig 1.5 to get Fig 1.6.

Click Configure button in Fig 1.6 to get Fig 1.7.

Check CLANG_BUILD_EXAMPLES, LLVM_BUILD_EXAMPLES, and uncheck LLVM_ENABLE_PIC as Fig 1.8.

Click Configure button again. If the output result message has no red color, then click Generate button to get Fig 1.9.

2.3 Build llvm by Xcode

Now, LLVM.xcodeproj is created. Open the cmake_debug_build/LLVM.xcodeproj by Xcode and click menu “Product – Build” as Fig 1.10.

After few minutes of build, the clang, llc, llvm-as, ..., can be found in cmake_debug_build/bin/Debug/ as Fig 1.11.

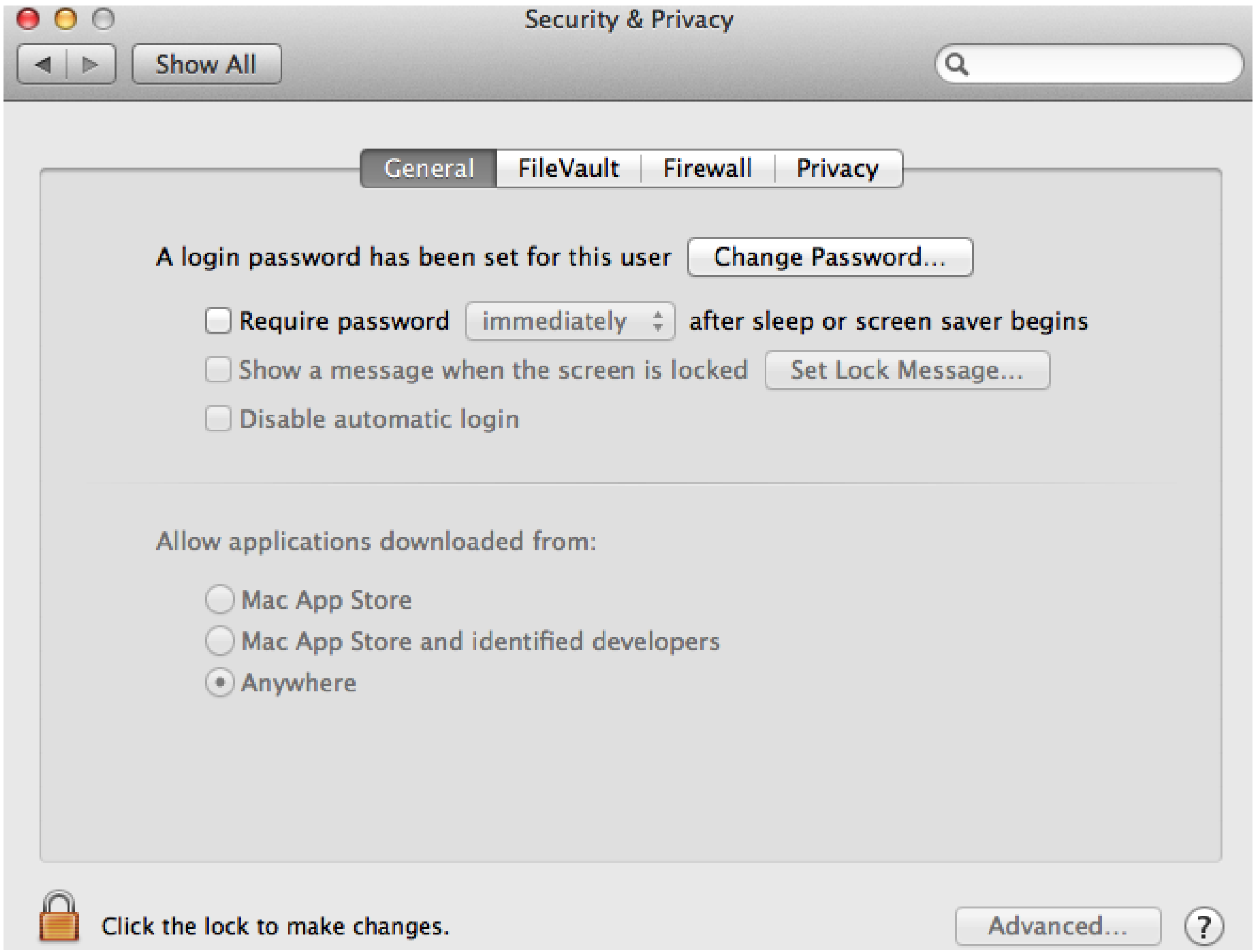


Figure 2.2: Fig 1.2 Adjust iMac to allow download cmake

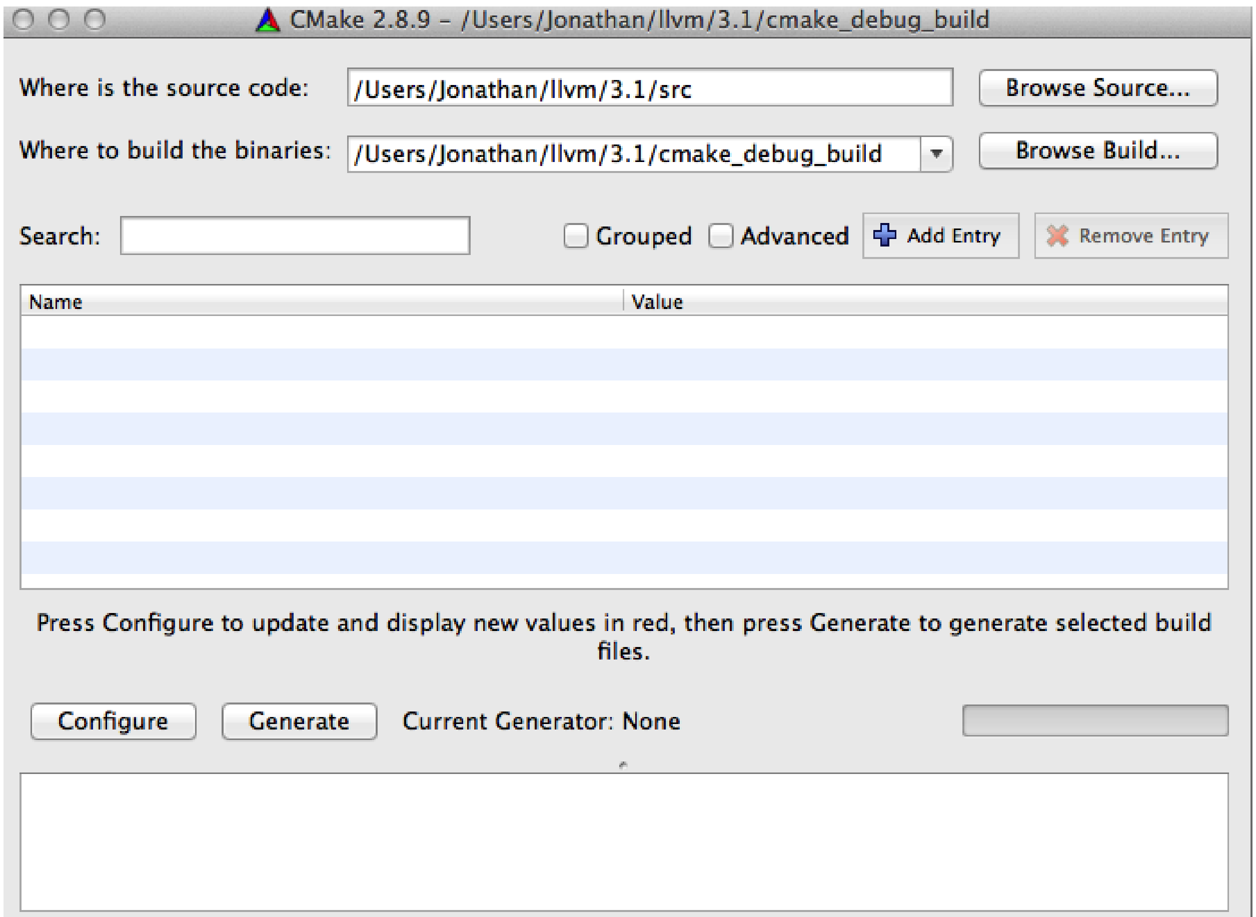


Figure 2.3: Fig 1.3 Start to create LLVM.xcodeproj by cmake

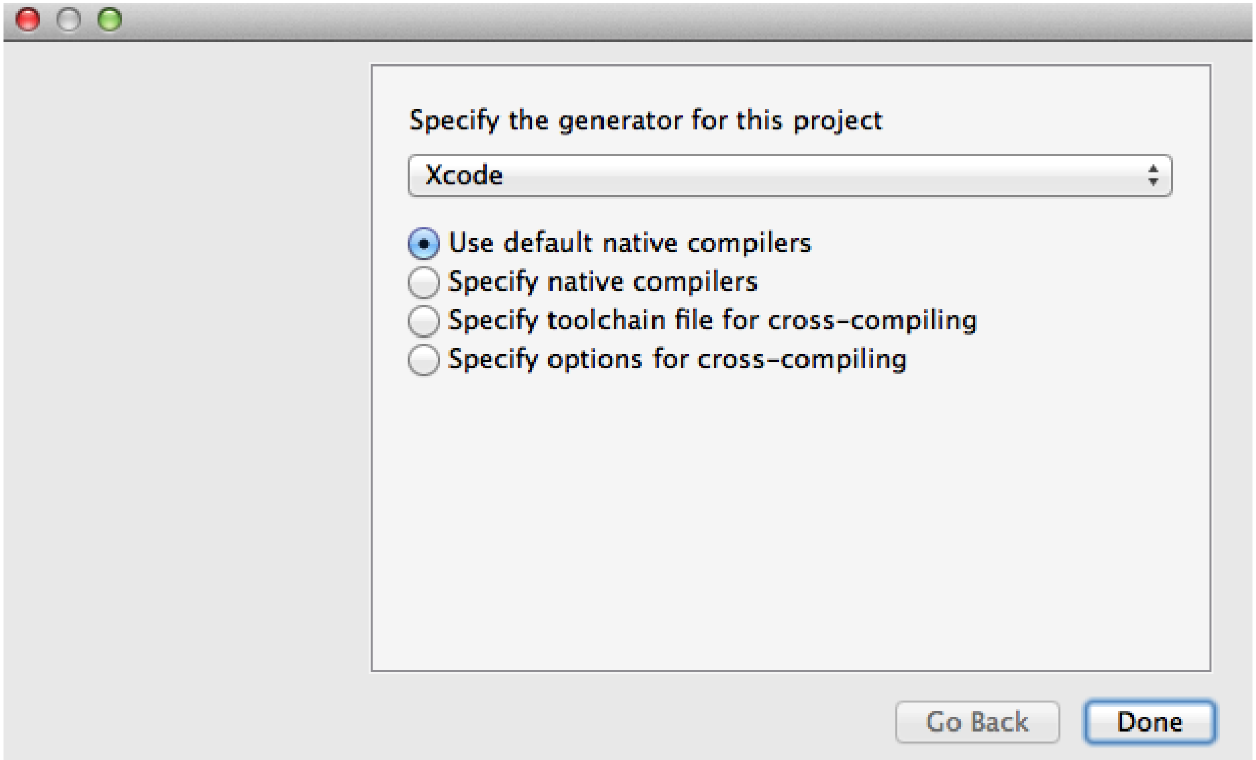


Figure 2.4: Fig 1.4 Create LLVM.xcodeproj by cmake – Set option to generate Xcode project

2.4 Create LLVM.xcodeproj of supporting cpu0 by terminal `cmake` command

In section 1.2, we create LLVM.xcodeproj by cmake graphic UI. We can create LLVM.xcodeproj by `cmake` command on terminal also. Now, let's repeat above steps to create llvm/3.1.test with cpu0 modified code as Fig 1.13.

/Users/Jonathan/Documents/Gamma_flash/LLVMBackendTutorial/src_files_modify/src/ contains the files I modified for cpu0 architecture. Copy it as Fig 1.13 to replace the original 3.1 source code for cpu0 backend support. After Fig 1.13, copy cpu0 example code from LLVMBackendTutorial/1/Cpu0 to src/lib/Target/ as Fig 1.14.

Please remove src/tools/clang since it will waste time to build clang for our working Cpu0 changes. Now, it's ready for building 1/Cpu0 code by command `cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -G "Xcode" ../src/` as Fig 1.15. Remind, currently, the `cmake` terminal command can work with lldb debug, but the section "1.2 cmake graphic UI steps" cannot.

Since Xcode use clang compiler and lldb instead of gcc and gdb, we can run lldb debug as Fig 1.16. About the lldb debug command, please reference http://lldb.llvm.org/lldb-gdb.html or lldb portal http://lldb.llvm.org/.

2.5 Install other tools on iMac

These tools mentioned in this section is for coding and debug. You can work even without these tools. Files compare tools Kdiff3 http://kdiff3.sourceforge.net. FileMerge is a part of Xcode, you can type FileMerge in Finder – Applications as Fig 1.17 and drag it into the Dock as Fig 1.18.

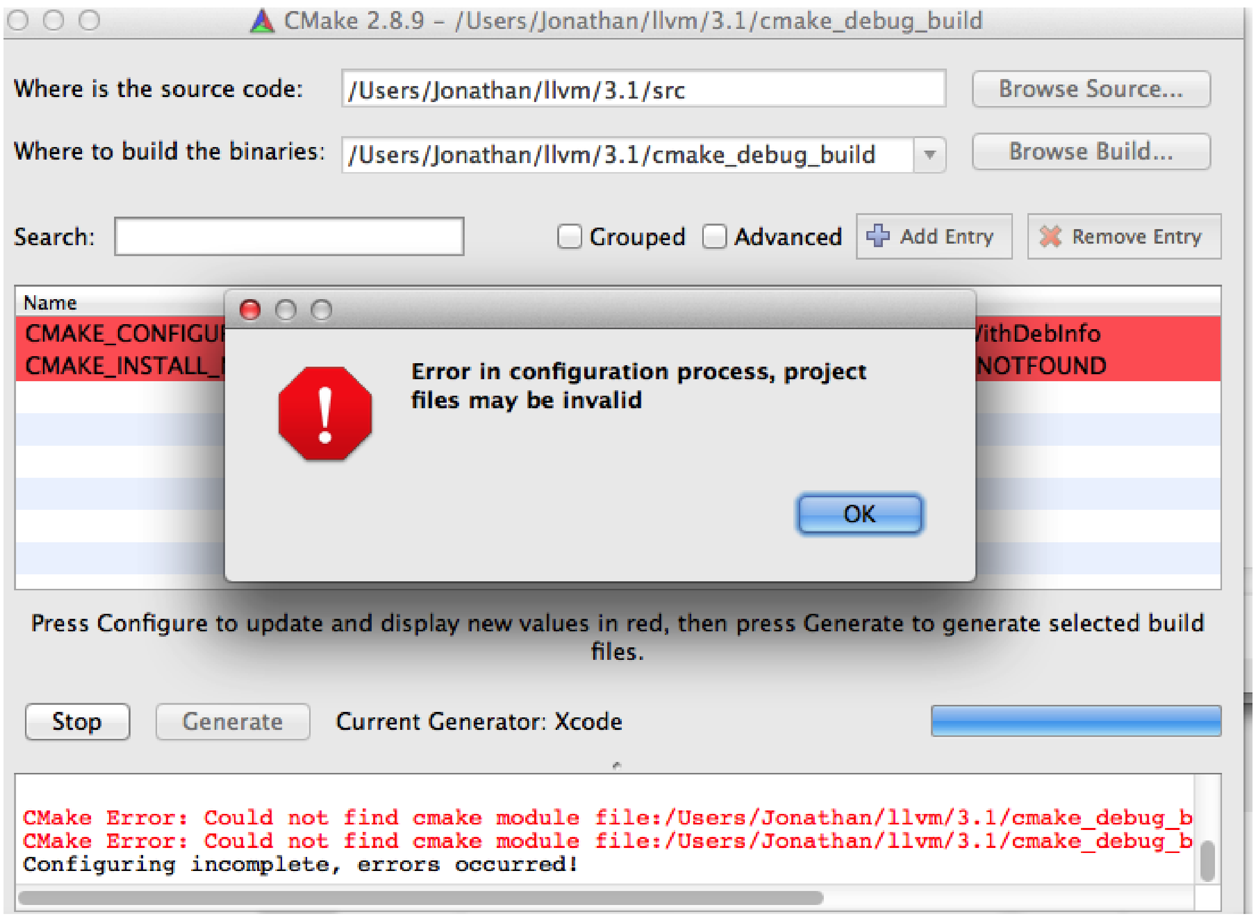


Figure 2.5: Fig 1.5 Create LLVM.xcodeproj by cmake – Before Adjust CMAKE_INSTALL_NAME_TOOL

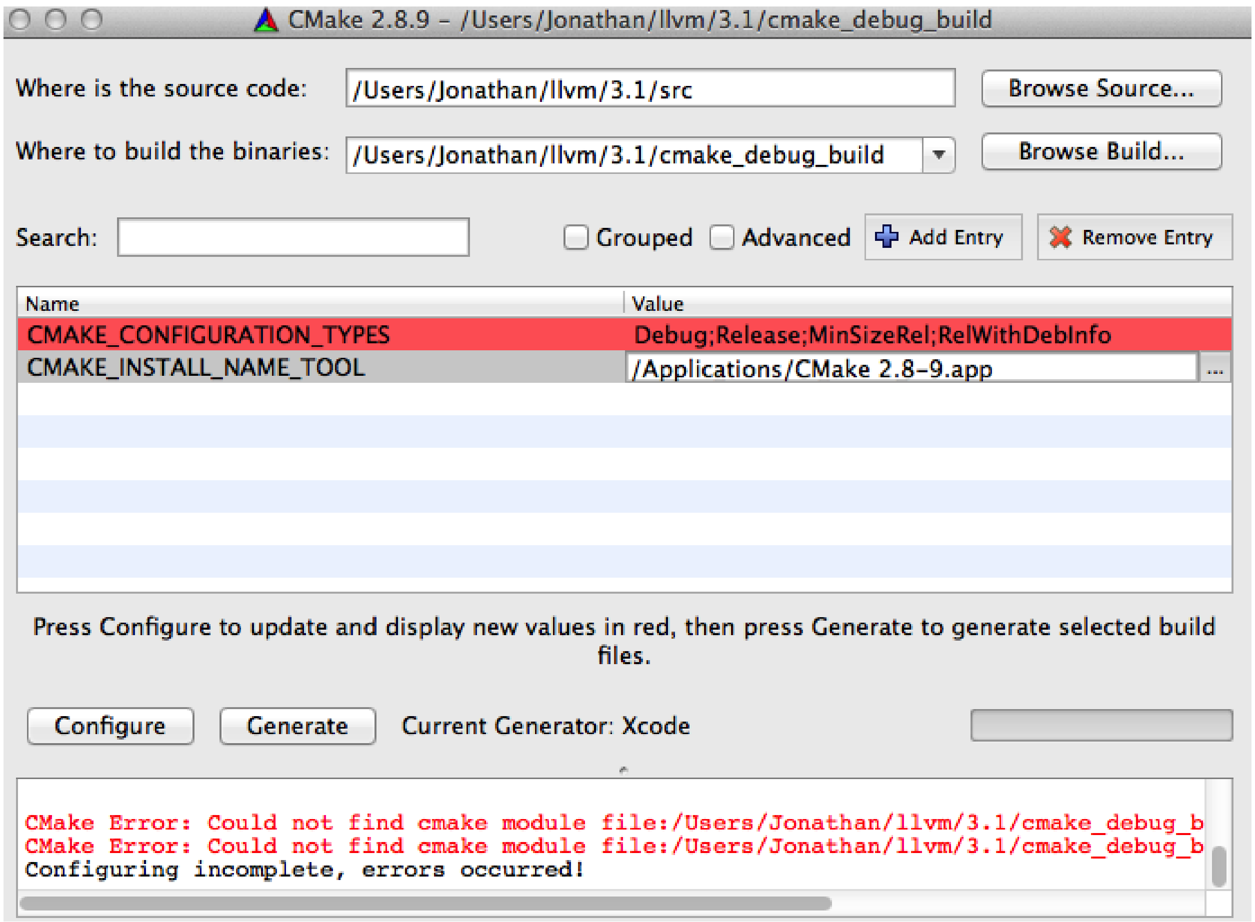


Figure 2.6: Fig 1.6 select Cmake 2.8-9.app

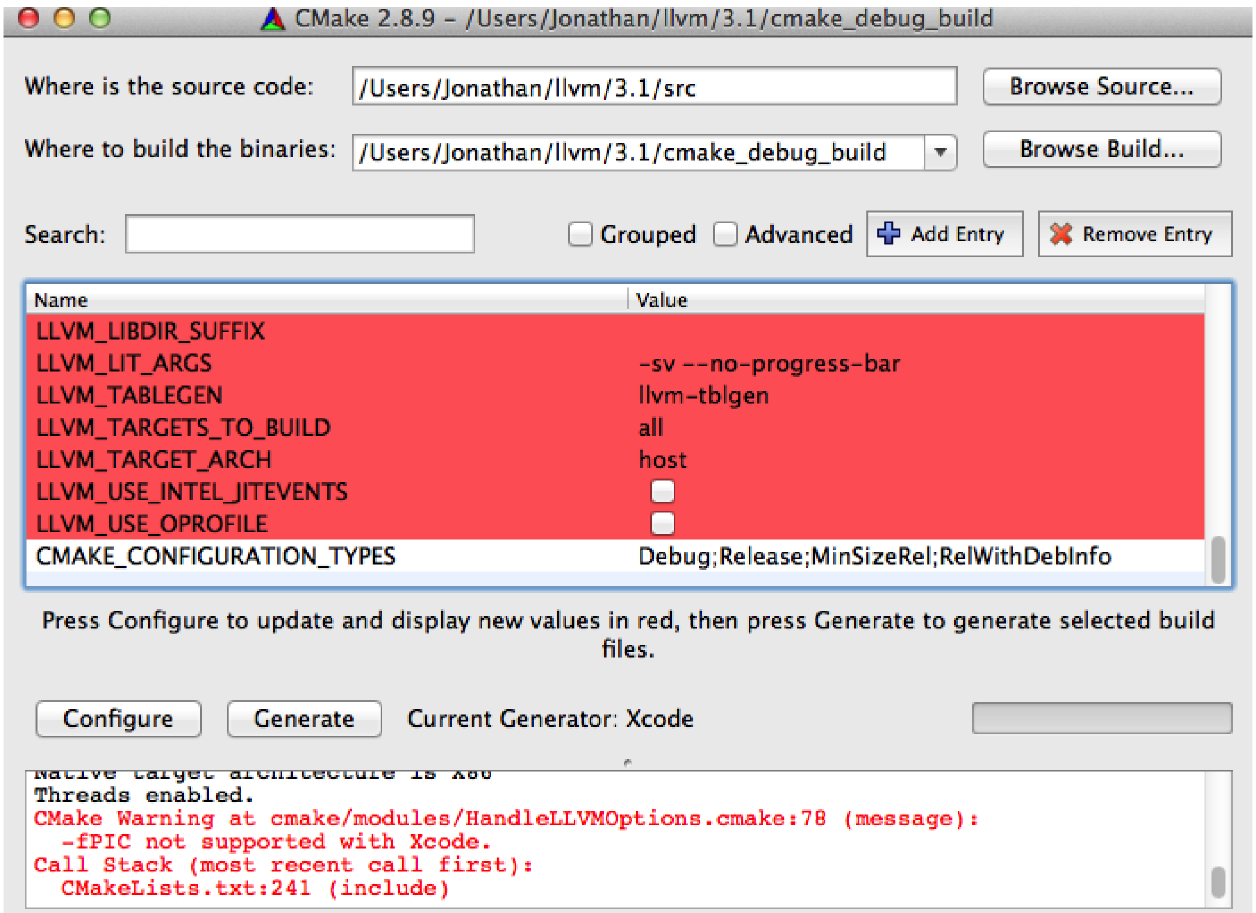


Figure 2.7: Fig 1.7 Click cmake Configure button first time

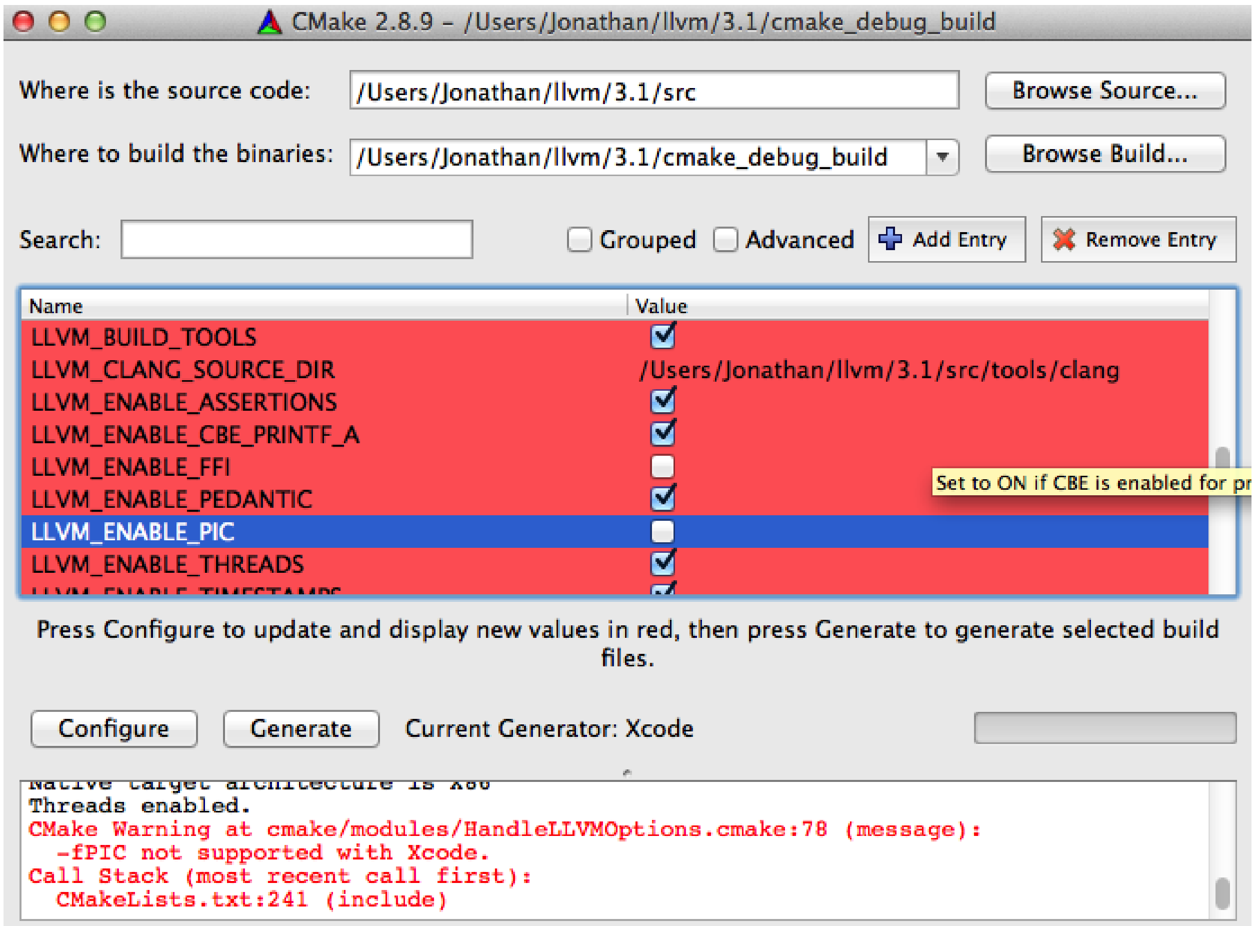


Figure 2.8: Fig 1.8 Check CLANG_BUILD_EXAMPLES, LLVM_BUILD_EXAMPLES, and uncheck LLVM_ENABLE_PIC in cmake

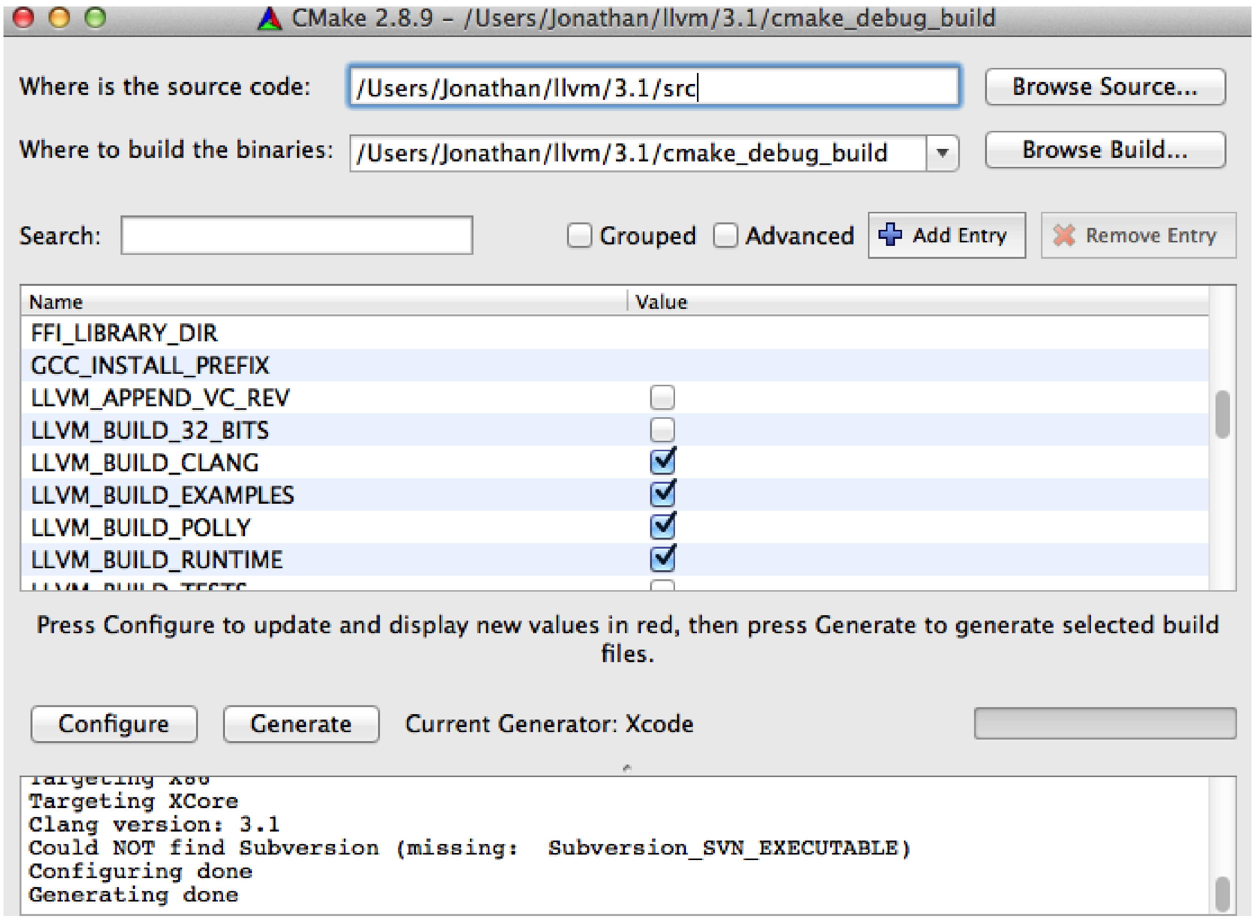


Figure 2.9: Fig 1.9 Click cmake Generate button second time

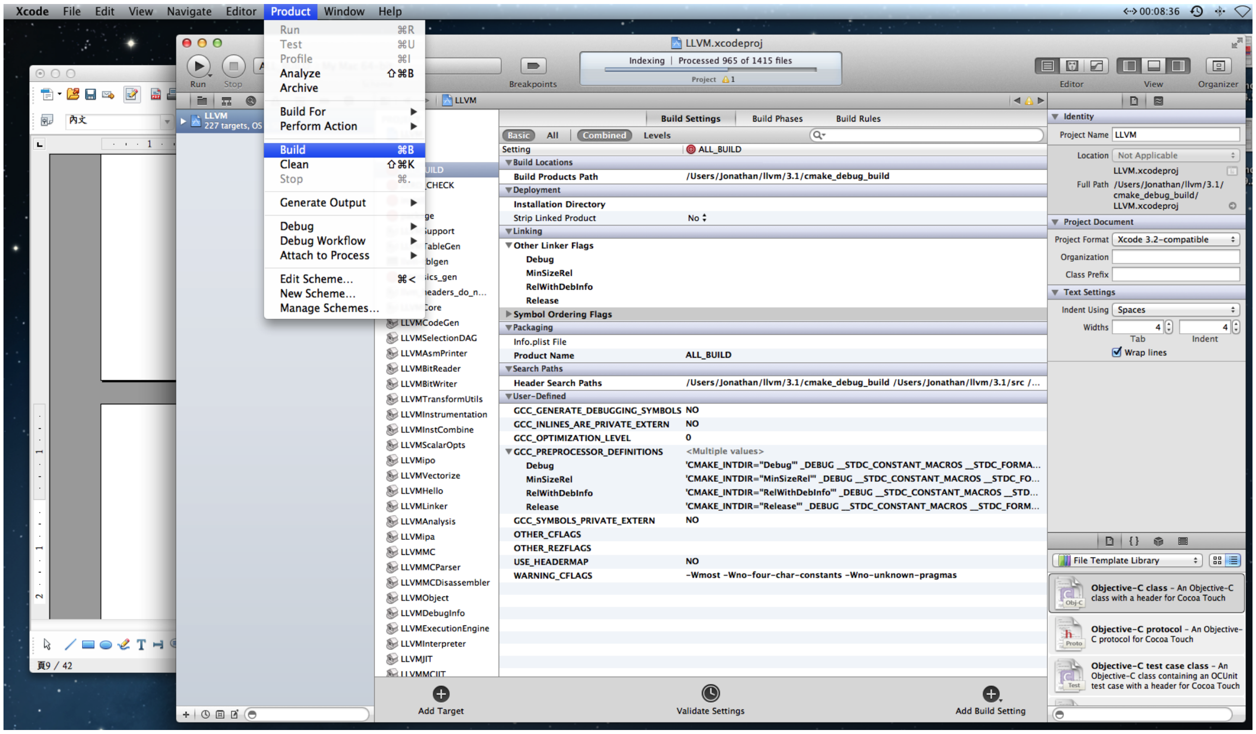


Figure 2.10: Fig 1.10 Click Build button to build LLVM.xcodeproj by Xcode

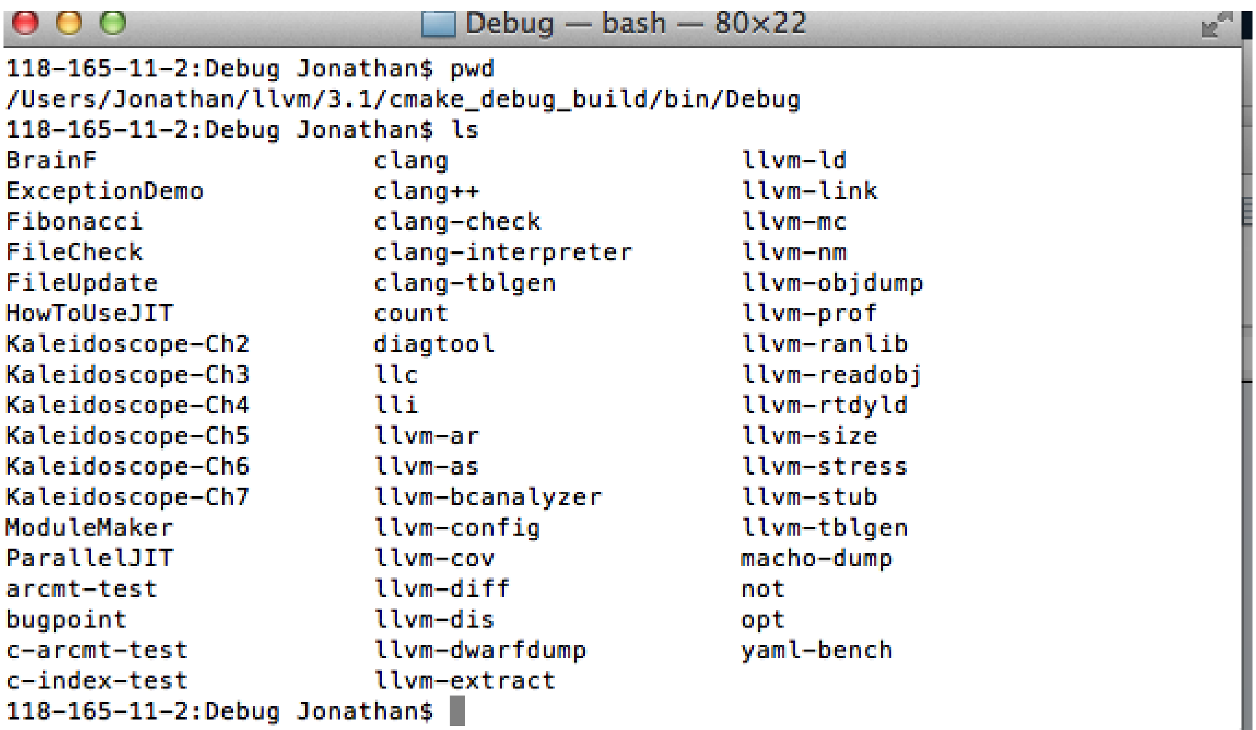


Figure 2.11: Fig 1.11 Execution files built by Xcode

To access those execution files, edit .profile (if you .profile not exists, please create file .profile), save .profile to /Users/Jonathan/, and enable $PATH by command `source .profile` as Fig1.12. Please add path /Applications//Xcode.app/Contents/Developer/usr/bin to .profile if you didn't add it after Xcode download.

```
Jonathan — bash — 156×9
118-165-11-2:~ Jonathan$ pwd
/Users/Jonathan
118-165-11-2:~ Jonathan$ cat .profile
export PATH=$PATH:/Applications//Xcode.app/Contents/Developer/usr/bin:/Users/Jonathan/llvm/3.1/cmake_debug_build/bin/Debug
118-165-11-2:~ Jonathan$ source .profile
118-165-11-2:~ Jonathan$ $PATH
-bash: /usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/Applications//Xcode.app/Contents/Developer/usr/bin:/Users/Jonathan/llvm/3.1/cmake_debug_build/bin/Debug
:/Applications//Xcode.app/Contents/Developer/usr/bin:/Users/Jonathan/llvm/3.1/cmake_debug_build/bin/Debug: No such file or directory
118-165-11-2:~ Jonathan$
```

Figure 2.12: Fig 1.12 Edit .profile and save .profile to /Users/Jonathan/

```
Target — bash — 100×29
118-165-11-2:llvm Jonathan$ cd 3.1.test/
118-165-11-2:3.1.test Jonathan$ mkdir cpu0
118-165-11-2:3.1.test Jonathan$ cd cpu0/
118-165-11-2:cpu0 Jonathan$ mkdir 1
118-165-11-2:cpu0 Jonathan$ cd 1
118-165-11-2:1 Jonathan$ mkdir src
118-165-11-2:1 Jonathan$ cd src
118-165-11-2:src Jonathan$ cp -rf /Users/Jonathan/llvm/3.1/src/ .
118-165-11-2:src Jonathan$ ls
CMakeLists.txt          Makefile.config.in      configure               projects
CREDITS.TXT             Makefile.rules          docs                    runtime
LICENSE.TXT             README.txt              examples                test
LLVMBuild.txt           autoconf                include                 tools
Makefile                bindings                lib                     unittests
Makefile.common         cmake                   llvm.spec.in            utils
118-165-11-2:src Jonathan$ cp -rf /Users/Jonathan/Documents/Gamma_flash/LLVMBackendTutorial/src_file
s_modify/src/* .
118-165-11-2:src Jonathan$ cd lib/Target/
118-165-11-2:Target Jonathan$ ls
ARM                             Mips                            TargetJITInfo.cpp
CMakeLists.txt                  PTX                             TargetLibraryInfo.cpp
CellSPU                         PowerPC                         TargetLoweringObjectFile.cpp
CppBackend                      README.txt                      TargetMachine.cpp
Hexagon                         Sparc                           TargetMachineC.cpp
LLVMBuild.txt                   Target.cpp                      TargetRegisterInfo.cpp
MBlaze                          TargetData.cpp                  TargetSubtargetInfo.cpp
MSP430                          TargetELFWriterInfo.cpp         X86
Makefile                        TargetInstrInfo.cpp             XCore
Mangler.cpp                     TargetIntrinsicInfo.cpp
```

Figure 2.13: Fig 1.13 create llvm/3.1.test with cpu0 modified code

```
Target — bash — 100×16
118-165-11-2:Target Jonathan$ pwd
/Users/Jonathan/llvm/3.1.test/cpu0/1/src/lib/Target
118-165-11-2:Target Jonathan$ cp -rf /Users/Jonathan/Documents/Gamma_flash/LLVMBackendTutorial/1/Cpu
0 .
118-165-11-2:Target Jonathan$ ls
ARM                             Mangler.cpp                     TargetIntrinsicInfo.cpp
CMakeLists.txt                  Mips                            TargetJITInfo.cpp
CellSPU                         PTX                             TargetLibraryInfo.cpp
CppBackend                      PowerPC                         TargetLoweringObjectFile.cpp
Cpu0                            README.txt                      TargetMachine.cpp
Hexagon                         Sparc                           TargetMachineC.cpp
LLVMBuild.txt                   Target.cpp                      TargetRegisterInfo.cpp
MBlaze                          TargetData.cpp                  TargetSubtargetInfo.cpp
MSP430                          TargetELFWriterInfo.cpp         X86
Makefile                        TargetInstrInfo.cpp             XCore
118-165-11-2:Target Jonathan$
```

Figure 2.14: Fig 1.14 copy cpu0 example code from 1/Cpu0 to src/lib/Target/

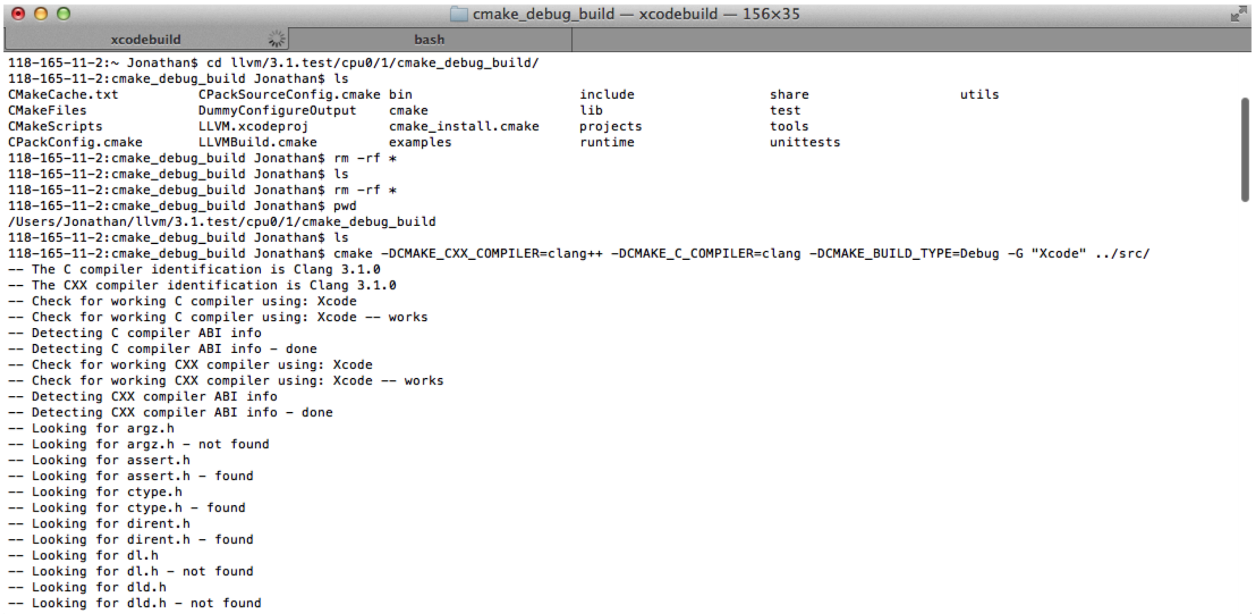


Figure 2.15: Fig 1.15 Build llvm debug cpu0 working project by `cmake` terminal command

Figure 2.16: Fig 1.16 Run lldb debug

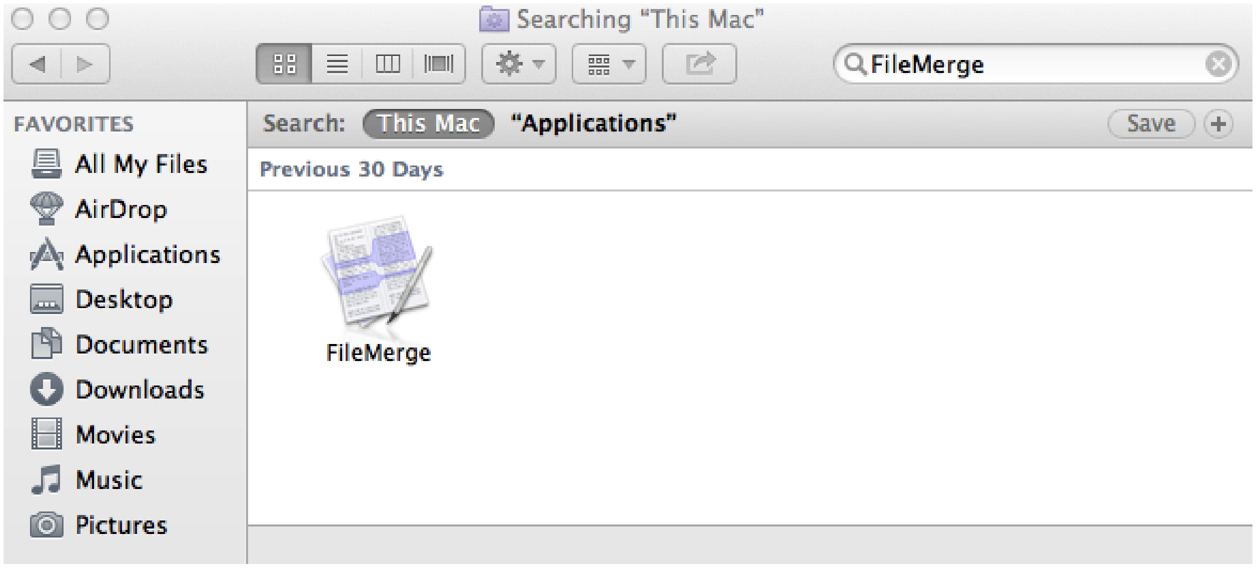


Figure 2.17: Fig 1.17 Type FileMerge in Finder – Applications

Figure 2.18: Fig 1.18 Drag FileMege into the Dock

Download tool Graphviz for display llvm IR nodes in debugging, http://www.graphviz.org/Download_macos.php. I choose mountainlion as Fig 1.19 since my iMac is Mountain Lion.

After install Graphviz, please set the path to .profile. For example, I install the Graphviz in directory /Applications/Graphviz.app/Contents/MacOS/, so add this path to /User/Jonathan/.profile as follows,

118-165-12-177:InputFiles Jonathan$ cat /Users/Jonathan/.profile export PATH=$PATH:/Applications/Xcode.app/Contents/Developer/u
lvm/3.1/cmake_debug_build/bin/Debug

The Graphviz information for llvm is in http://llvm.org/docs/CodeGenerator.html? highlight=graph%20view and http://llvm.org/docs/ProgrammersManual.html#ViewGraph. TextWrangler is for edit file with line number display and dump binary file like the obj file, *.o, that will be generated in chapter 2. You can download from App Store. To dump binary file, first, open the binary file, next, select menu “File – Hex Front Document” as Fig 1.20. Then select “Front document’s file” as Fig 1.21.

2.6 Install LLVM 3.1 release build on Linux

First, install the llvm release build by, 1) Untar llvm source, rename llvm source with src. 2) Untar clang and move it src/tools/clang. 3) Untar compiler-rt and move it to src/project/compiler-rt as Fig 1.22.

Next, build with cmake command, `cmake -DCMAKE_BUILD_TYPE=Release -DCLANG_BUILD_EXAMPLES=ON -DLLVM_BUILD_EXAMPLES=ON -G "Unix Makefiles" ../src/`, shown in Fig 1.23.

After cmake, run command `make`, then you can get clang, llc, llvm-as, ..., in cmake_release_build/bin/ after a few tens minutes of build. Next, edit /home/Gamma/.bash_profile with adding /usr/local/llvm/3.1/cmake_release_build/bin to PATH to enable the clang, llc, ..., command search path, as shown in Fig 1.24.

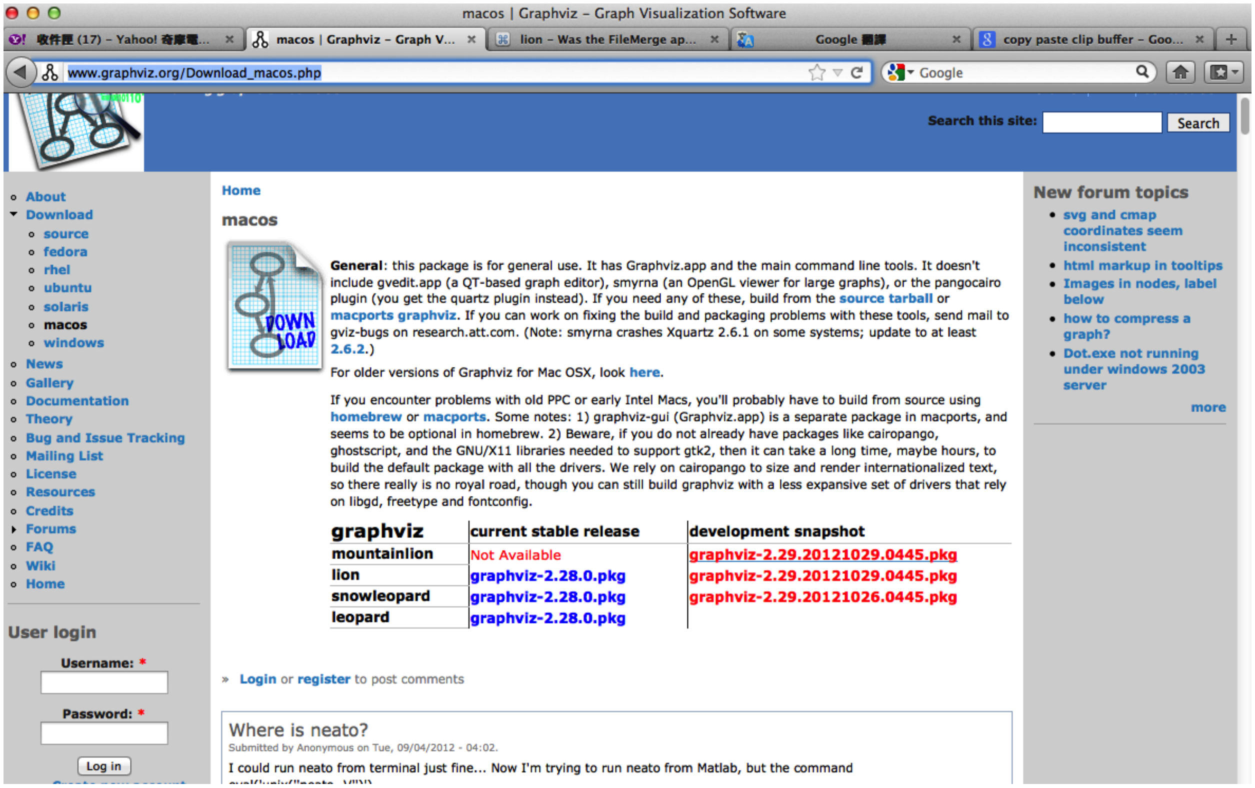


Figure 2.19: Fig 1.19 Download graphviz for llvm IR node display

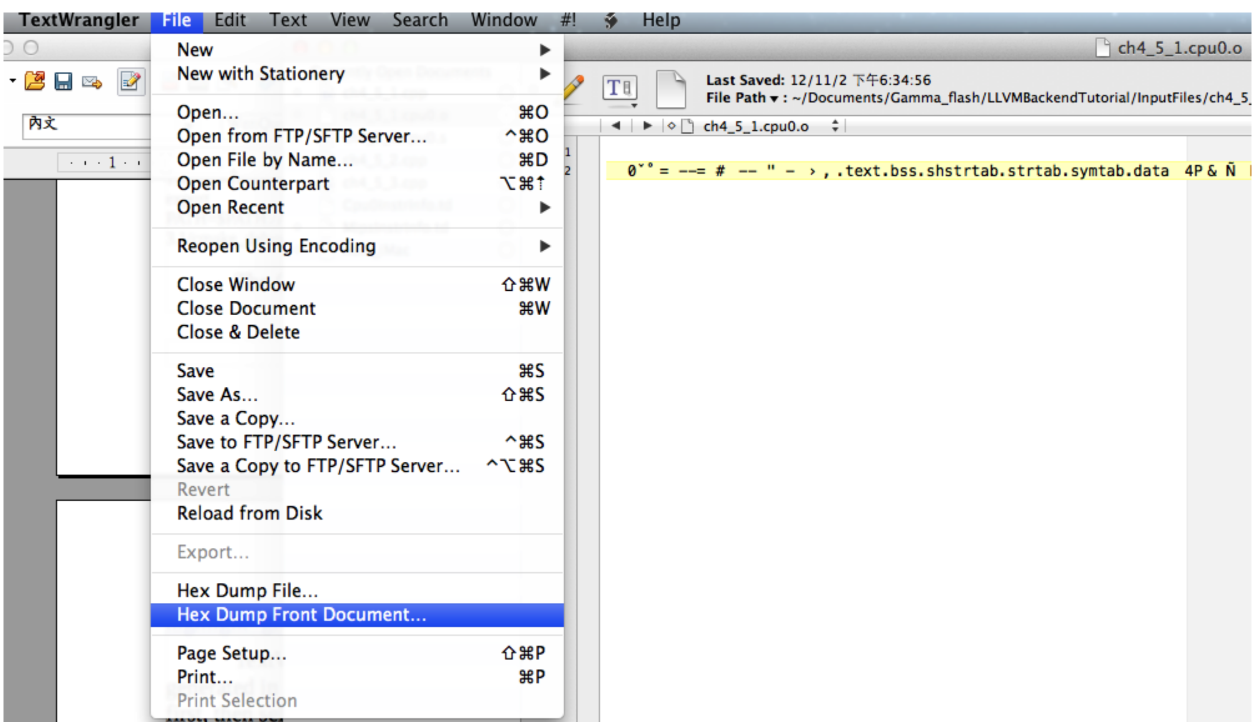


Figure 2.20: Fig 1.20 Select Hex Dump menu

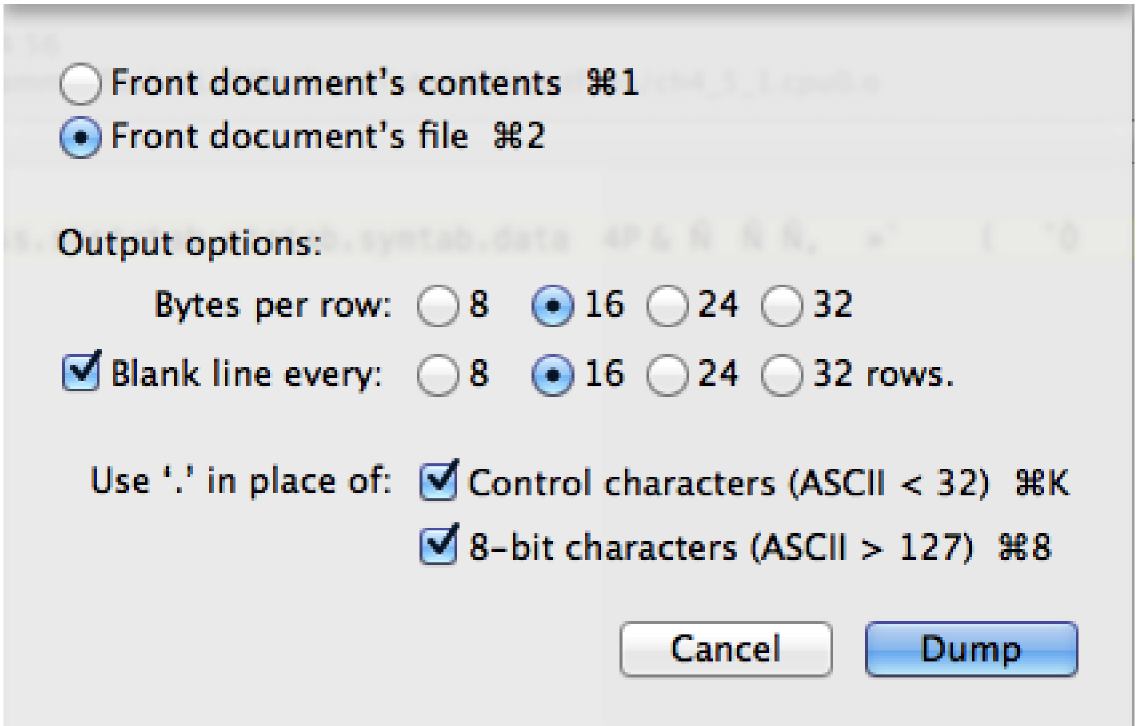


Figure 2.21: Fig 1.21 Select Front document's file in TextWrangler

2.7 Install cpu0 debug build on Linux

Make another copy /usr/local/llvm/3.1.test/cpu0/1/src for cpu0 debug working project according the following list steps, the corresponding commands shown in Fig 1.25:

1. Enter /usr/local/llvm/3.1.test/cpu0/1 and `cp -rf /usr/local/llvm/3.1/src .`.
2. Update my modified files to support cpu0 by command, `cp -rf /home/Gamma/Gamma_flash/LLVMBackendTutorial/src_files_modify/src .`.
3. Enter src/lib/Target and copy example code LLVMBackendTutorial/1/Cpu0 to the directory by command `cd src/lib/Target/` and `cp -rf /home/Gamma/Gamma_flash/LLVMBackendTutorial/1/Cpu0 .`.
4. Go into directory 3.1.test/cpu0/1/src and Check step 3 is effect by command `grep -R "Cpu0" . | more`'. I add the Cpu0 backend support, so check with grep.
5. Remove clang from 3.1.test/cpu0/1/src/tools/clang, and mkdir 3.1.test/cpu0/1/cmake_debug_build. Without this you will waste extra time for command `make` in cpu0 example code build.

Now, go into directory 3.1.test/cpu0/1, create directory cmake_debug_build and do cmake like build the 3.1 release, but we do Debug build and use clang as our compiler instead, as follows,

```
[Gamma@localhost src]$ cd ..
[Gamma@localhost 1]$ pwd
/usr/local/llvm/3.1.test/cpu0/1
[Gamma@localhost 1]$ mkdir cmake_debug_build
[Gamma@localhost 1]$ cd cmake_debug_build/
[Gamma@localhost cmake_debug_build]$ cmake
-DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang
-DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" ../src/
```

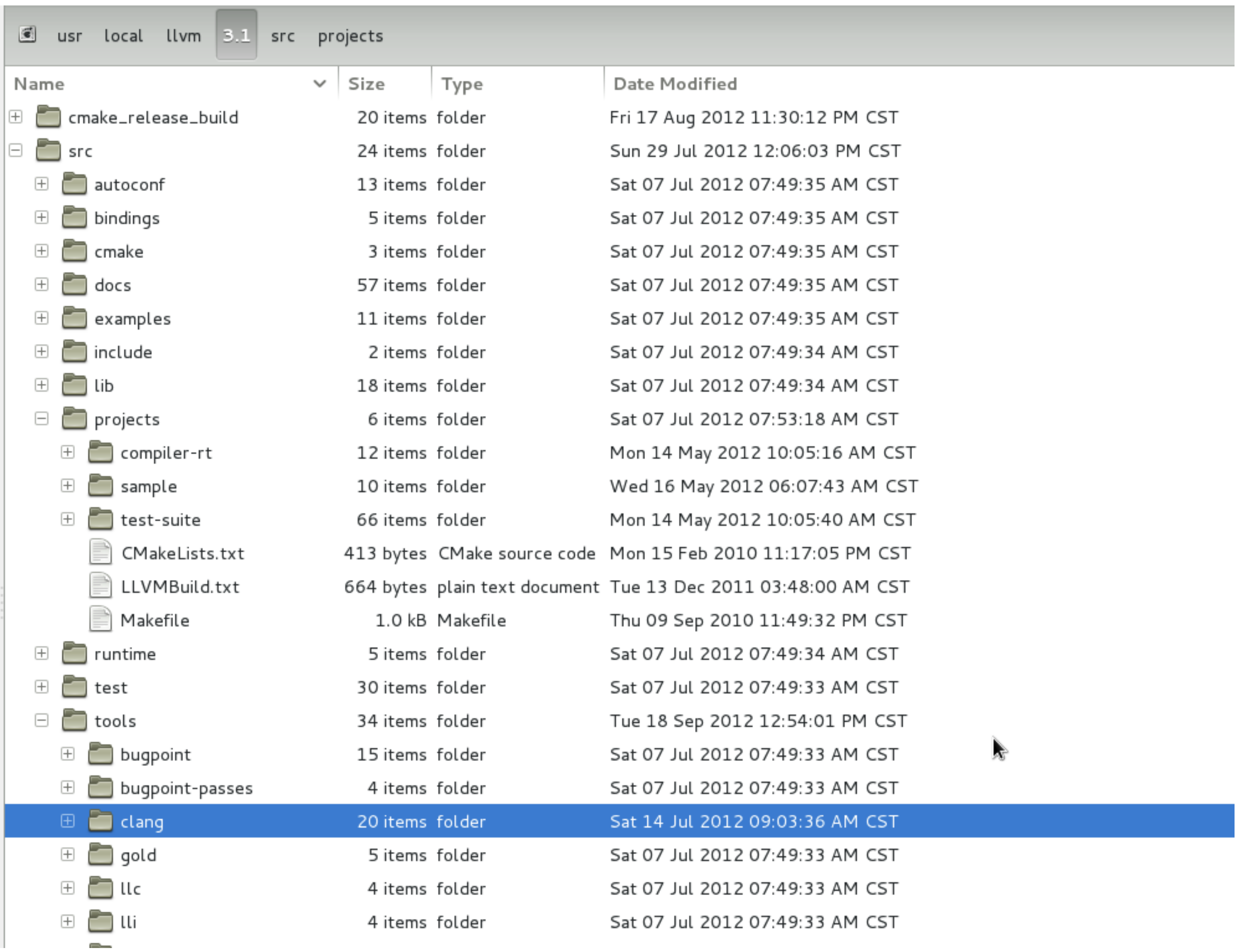
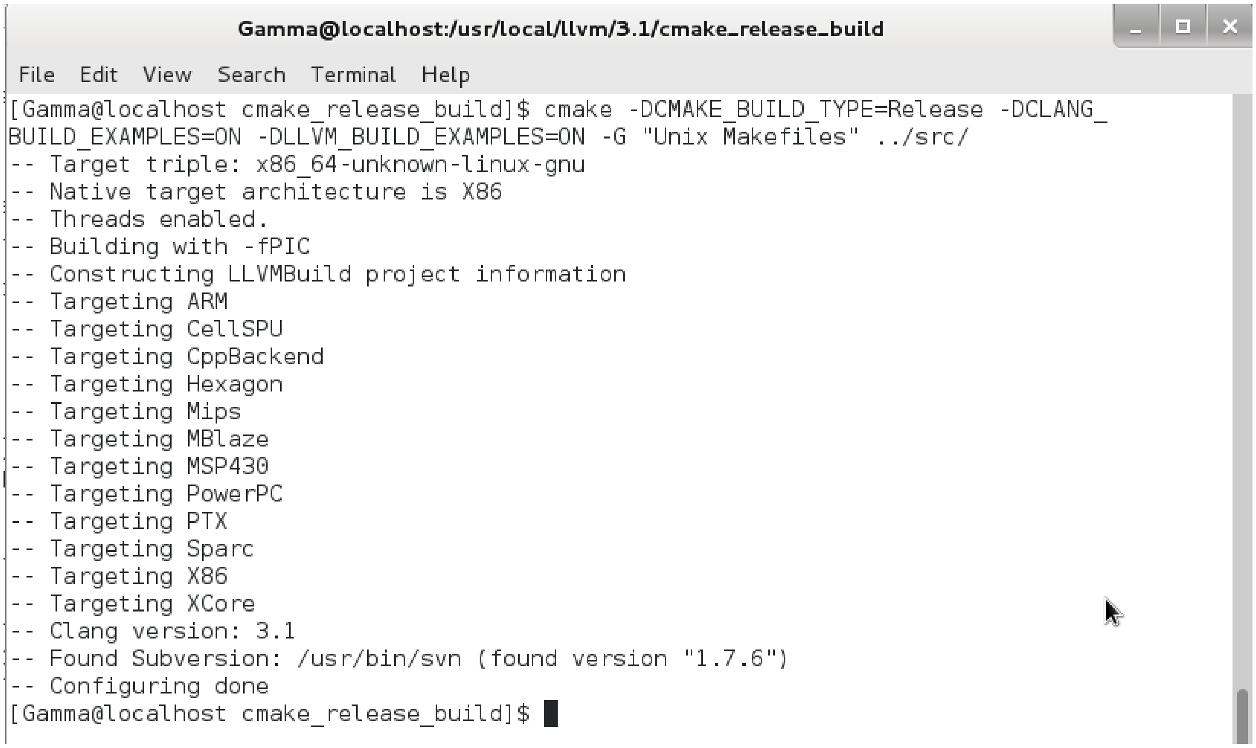
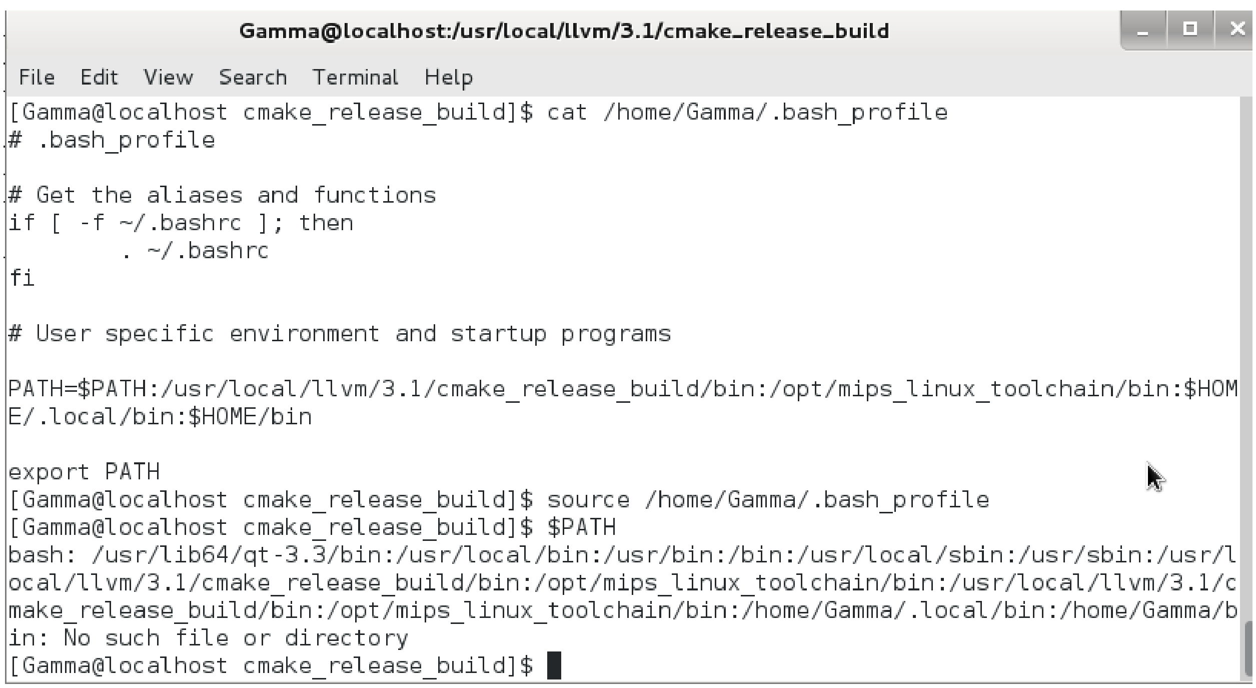


Figure 2.22: Fig 1.22 Create llvm release build



```
Gamma@localhost:/usr/local/llvm/3.1/cmake_release_build
File  Edit  View  Search  Terminal  Help
[Gamma@localhost cmake_release_build]$ cmake -DCMAKE_BUILD_TYPE=Release -DCLANG_
BUILD_EXAMPLES=ON -DLLVM_BUILD_EXAMPLES=ON -G "Unix Makefiles" ../src/
-- Target triple: x86_64-unknown-linux-gnu
-- Native target architecture is X86
-- Threads enabled.
-- Building with -fPIC
-- Constructing LLVMBuild project information
-- Targeting ARM
-- Targeting CellSPU
-- Targeting CppBackend
-- Targeting Hexagon
-- Targeting Mips
-- Targeting MBlaze
-- Targeting MSP430
-- Targeting PowerPC
-- Targeting PTX
-- Targeting Sparc
-- Targeting X86
-- Targeting XCore
-- Clang version: 3.1
-- Found Subversion: /usr/bin/svn (found version "1.7.6")
-- Configuring done
[Gamma@localhost cmake_release_build]$
```

Figure 2.23: Fig 1.23 Create llvm 3.1 release build



```
Gamma@localhost:/usr/local/llvm/3.1/cmake_release_build
File  Edit  View  Search  Terminal  Help
[Gamma@localhost cmake_release_build]$ cat /home/Gamma/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:/usr/local/llvm/3.1/cmake_release_build/bin:/opt/mips_linux_toolchain/bin:$HOM
E/.local/bin:$HOME/bin

export PATH
[Gamma@localhost cmake_release_build]$ source /home/Gamma/.bash_profile
[Gamma@localhost cmake_release_build]$ $PATH
bash: /usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/usr/l
ocal/llvm/3.1/cmake_release_build/bin:/opt/mips_linux_toolchain/bin:/usr/local/llvm/3.1/c
make_release_build/bin:/opt/mips_linux_toolchain/bin:/home/Gamma/.local/bin:/home/Gamma/b
in: No such file or directory
[Gamma@localhost cmake_release_build]$
```

Figure 2.24: Fig 1.24 Setup llvm command path

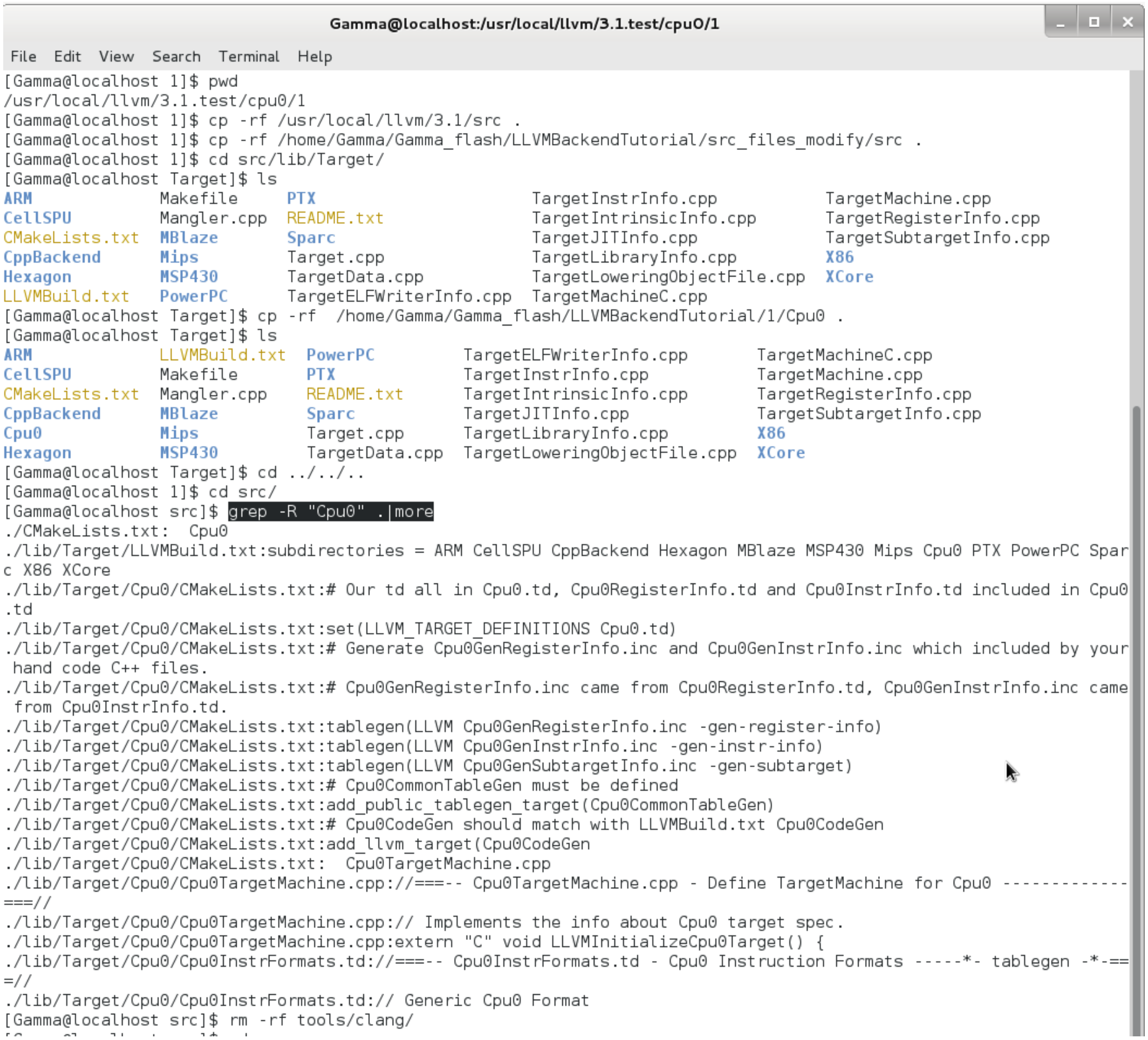


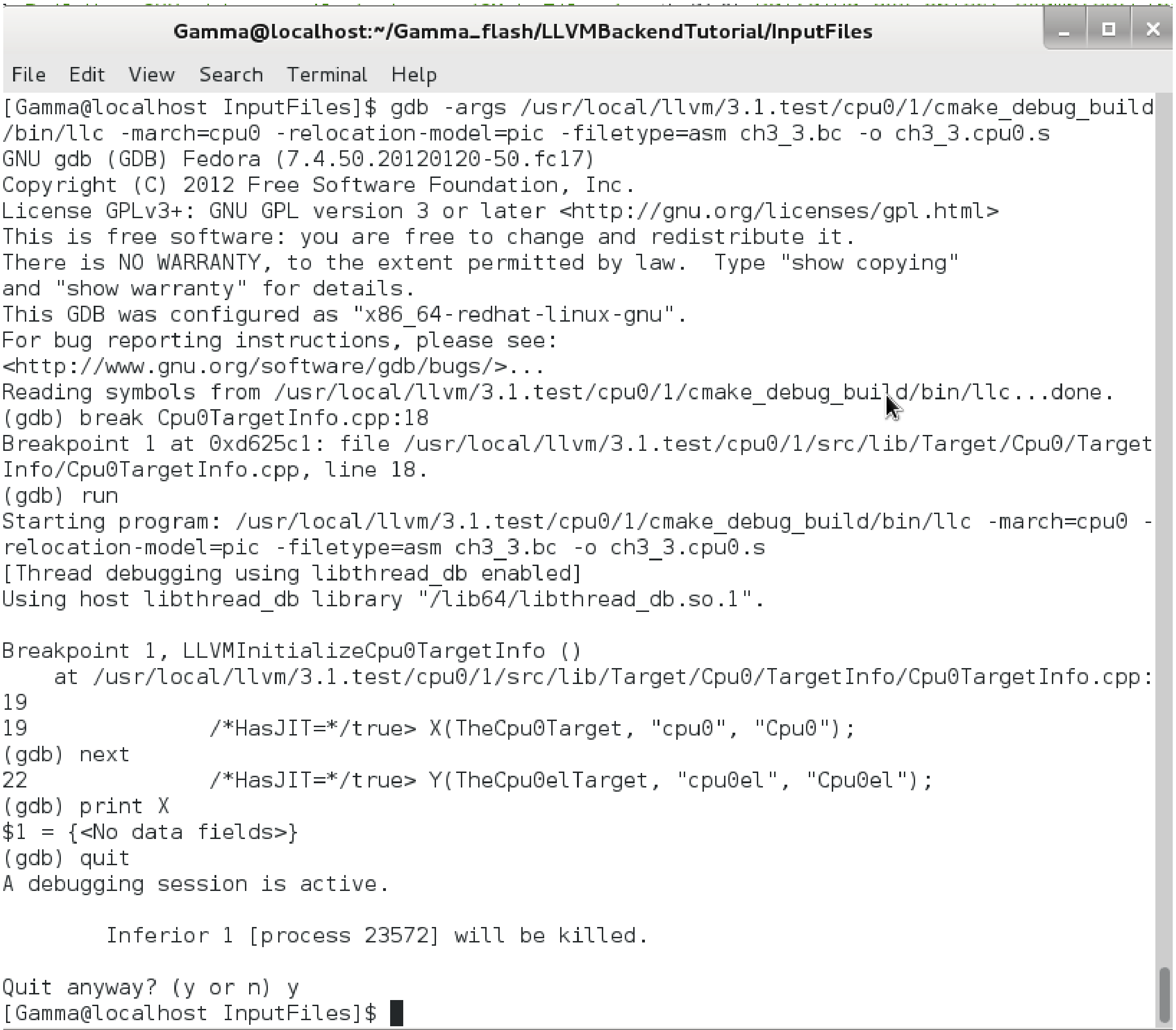
Figure 2.25: Fig 1.25 Create llvm 3.1 debug copy

```
-- The C compiler identification is Clang 3.1.0
-- The CXX compiler identification is Clang 3.1.0
-- Check for working C compiler: /usr/local/llvm/3.1/cmake_release_build/bin/clang
-- Check for working C compiler: /usr/local/llvm/3.1/cmake_release_build/bin/clang
 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/local/llvm/3.1/cmake_release_build/bin/clang++
-- Check for working CXX compiler: /usr/local/llvm/3.1/cmake_release_build/bin/clang++
 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done ...
-- Targeting Mips
-- Targeting Cpu0
-- Targeting MBlaze
-- Targeting MSP430
-- Targeting PowerPC
-- Targeting PTX
-- Targeting Sparc
-- Targeting X86
-- Targeting XCore
-- Configuring done
-- Generating done
-- Build files have been written to: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build
[Gamma@localhost cmake_debug_build]$
```

Then do make as follows,

```
[Gamma@localhost cmake_debug_build]$ make
Scanning dependencies of target LLVMSupport
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/APFloat.cpp.o
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/APInt.cpp.o
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/APSInt.cpp.o
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/Allocator.cpp.o
[ 1%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/BlockFrequency.cpp.o ...
Linking CXX static library ../../lib/libgtest.a
[100%] Built target gtest
Scanning dependencies of target gtest_main
[100%] Building CXX object utils/unittest/CMakeFiles/gtest_main.dir/UnitTestMain/
TestMain.cpp.o Linking CXX static library ../../lib/libgtest_main.a
[100%] Built target gtest_main
[Gamma@localhost cmake_debug_build]$
```

Now, we are ready for the cpu0 backend development. We can run gdb debug as follows. If your setting has anything about gdb errors, please follow the errors indication (maybe need to download gdb again). Finally, try gdb as Fig 1.26.



```
Gamma@localhost:~/Gamma_flash/LLVMBackendTutorial/InputFiles
File Edit View Search Terminal Help
[Gamma@localhost InputFiles]$ gdb -args /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build
/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3_3.bc -o ch3_3.cpu0.s
GNU gdb (GDB) Fedora (7.4.50.20120120-50.fc17)
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc...done.
(gdb) break Cpu0TargetInfo.cpp:18
Breakpoint 1 at 0xd625c1: file /usr/local/llvm/3.1.test/cpu0/1/src/lib/Target/Cpu0/Target
Info/Cpu0TargetInfo.cpp, line 18.
(gdb) run
Starting program: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
relocation-model=pic -filetype=asm ch3_3.bc -o ch3_3.cpu0.s
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, LLVMInitializeCpu0TargetInfo ()
    at /usr/local/llvm/3.1.test/cpu0/1/src/lib/Target/Cpu0/TargetInfo/Cpu0TargetInfo.cpp:
19
19              /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "Cpu0");
(gdb) next
22              /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "Cpu0el");
(gdb) print X
$1 = {<No data fields>}
(gdb) quit
A debugging session is active.

        Inferior 1 [process 23572] will be killed.

Quit anyway? (y or n) y
[Gamma@localhost InputFiles]$
```

Figure 2.26: Fig 1.26 Debug llvm cpu0 backend by gdb

CHAPTER

THREE

CPU0 INSTRUCTION AND LLVM TARGET DESCRIPTION

In this chapter, I show you the cpu0 instruction format first. Next, I introduce the llvm structure by copy and paste the related article from llvm web site. After that I will show you how to write register and instruction definitions (Target Description File) which will be used in chapter 3.

3.1 CPU0 processor architecture

I copy and redraw figures in english in this section. This web site is chinese version and here is english version.

3.1.1 Brief introduction

CPU0 is a 32-bit processor registers R0 .. R15, IR, MAR, MDR, etc., and its structure is shown below.

Uses of each register as follows:

3.1.2 Instruction Set for CPU0

The CPU0 instruction divided into three types, L-type usually load the saved instruction, A-type arithmetic instruction-based J-type usually jump instruction, the following figure shows the three types of instruction encoding format.

The following is the CPU0 processor's instruction table format

In the second edition of CPU0_v2 we fill the following command:

3.1.3 Status register

CPU0 status register contains the state of the N, Z, C, V, and I, T and other interrupt mode bit. Its structure is shown below.

When CMP Ra, Rb instruction execution, the state flag will thus change.

If Ra> Rb, then the setting state of N = 0, Z = 0. If Ra <Rb, it will set the state of N = 1, Z = 0. If Ra = Rb, then the setting state of N = 0, Z = 1.

So conditional jump the JGT, JLT, JGE, JLE, JEQ, JNE instruction jumps N, Z flag in the status register.

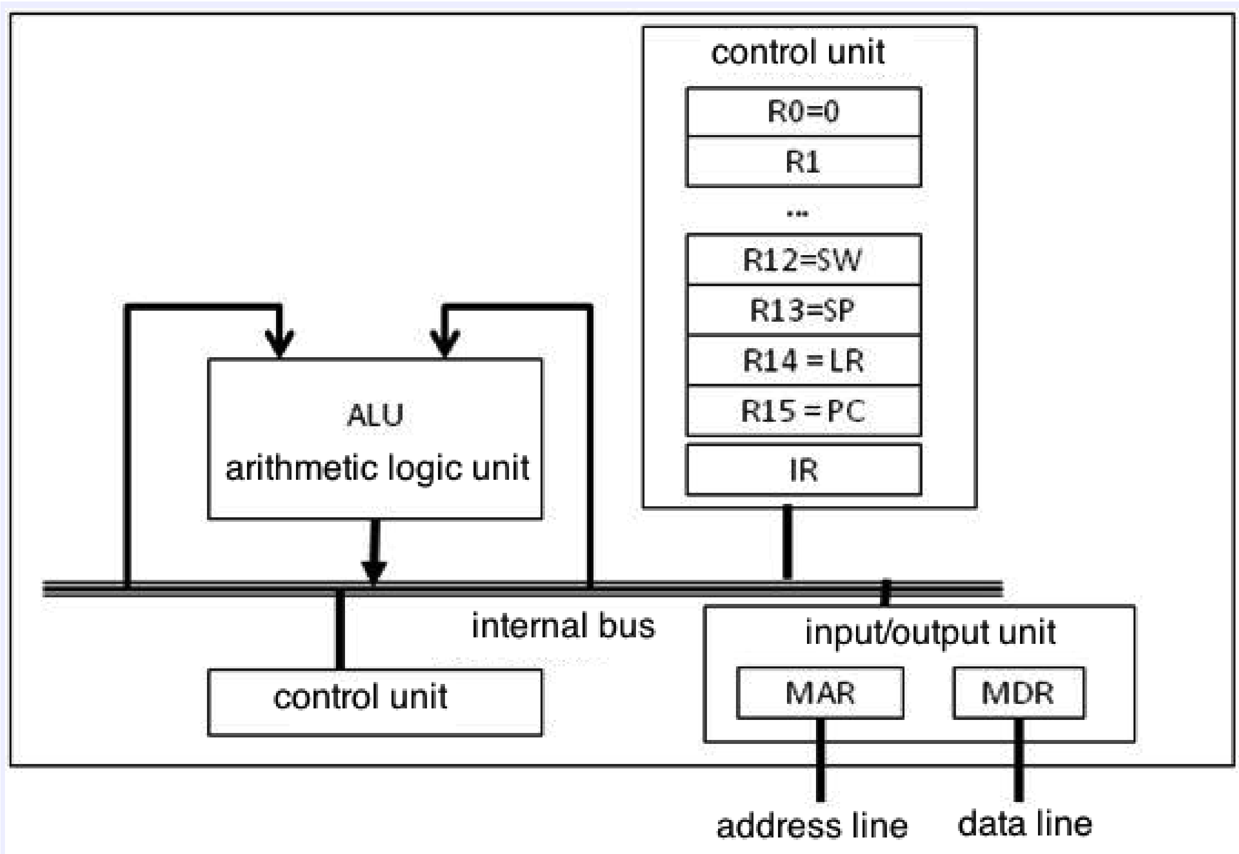


Figure 3.1: The structure of the processor of Figure 2.1: CPU0

IR	Instruction register
R0	Constant registers, its value is always 0.
R1 ~ R11	General-purpose registers.
R12	Status register (Status Word: SW)
R13	Stack pointer register (Stack Pointer: SP)
R14	Link register (Link Register: LR)
R15	Program counter (Program Counter: PC)
MAR	Address register (Memory Address Register)
MDR	Data register (Memory Data Register)

Figure 3.2: Table 2.1 Cpu0 registers table

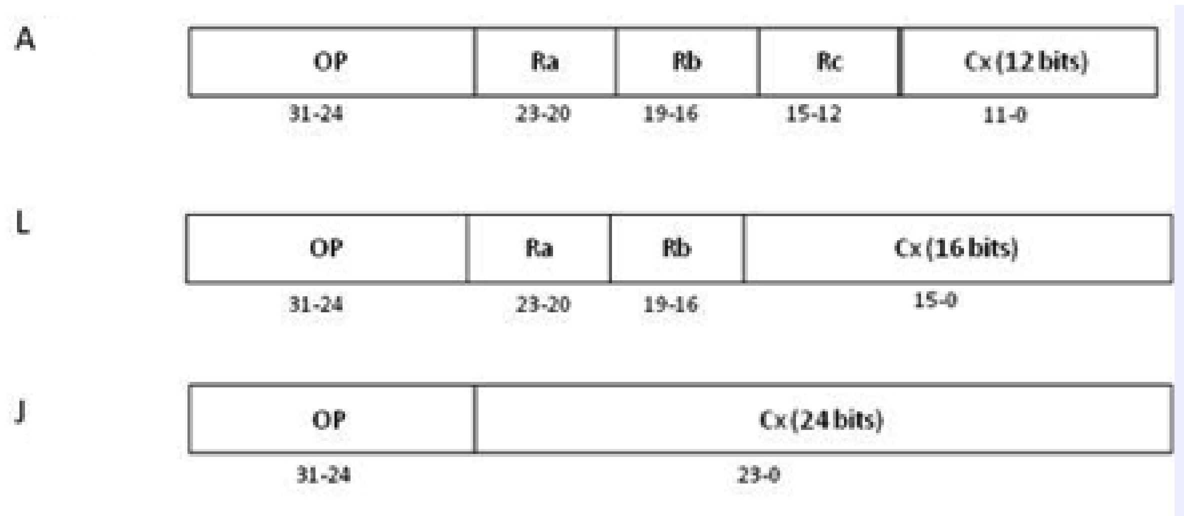


Figure 3.3: Fig 2.2: CPU0 three instruction formats

3.1.4 The execution of the instruction step

CPU0 in executing an instruction must be extracted, decoding and execution of three stages.

Extraction stage

> Action 1, the instruction fetch: IR = [PC] Action 2 update counter: PC = PC + 4

Decode stage

> Action 3, the decoding: the control unit for IR decoding, setting the data flow modes of operation of the switch with the ALU

Runtime stage

> Action 4 Execution: information inflow ALU, after the operation, the flow back into the specified register

3.1.5 Replace ldi instruction by addiu instruction

I have recognized the ldi instruction is a bad design and replace it with mips instruction addiu. The reason I replace ldi with addiu is that ldi use only one register even though ldi is L type format and has two registers, as Fig 2.4. Mips addiu which allow programmer to do load constant to register like ldi, and add constant to a register. So, it's powerful and fully contains the ldi ability. These two instructions format as Fig 2.4 and Fig 2.5.

From Fig 2.4 and Fig 2.5, you can find ldi $Ra, 5 can be replaced by addiu $Ra, $zero, 5. And more, addiu can do addiu $Ra, $Rb, 5 which add $Rb and 5 then save to $Ra, but ldi cannot. As a cpu design, it's common to redesign CPU instruction when find a better solution during design the compiler backend for that CPU. So, I add addiu instruction to cpu0. The cpu0 is my brother's work, I will find time to talk with him.

3.2 LLVM structure

Following came from http://www.aosabook.org/en/llvm.html.

The most popular design for a traditional static compiler (like most C compilers) is the three phase design whose major components are the front end, the optimizer and the back end (Fig 2.6). The front end parses source code, checking

type	format	instruction	OP	meaning	syntax	semantic
Load / Store	L	LD[1]	00	Load word	LD Ra, [Rb+Cx]	Ra ← [Rb+Cx]
	L	ST	01	Store word	ST Ra, [Rb+Cx]	Ra → [Rb+Cx]
	L	LDB	02	Load byte	LDB Ra, [Rb+Cx]	Ra ← (byte)[Rb+Cx]
	L	STB	03	Store byte	STB Ra, [Rb+Cx]	Ra → (byte)[Rb+Cx]
	A	LDR	04	LD (register version)	LDR Ra, [Rb+Rc]	Ra → (byte)[Rb+Rc]
	A	STR	05	LD (register version)	STR Ra, [Rb+Rc]	Ra → [Rb+Rc]
	A	LBR	06	LDB (register version)	LBR Ra, [Rb+Rc]	Ra ← (byte)[Rb+Rc]
	A	SBR	07	STB (register version)	SBR Ra, [Rb+Rc]	Ra → (byte)[Rb+Rc]
	L	LDI	08	Load immediate	LDI Ra, Cx	Ra ← Cx
Mathematic	A	CMP[2]	10	Compare	CMP Ra, Rb	SW ← Ra >=< Rb
	A	MOV	12	Move	MOV Ra, Rb	Ra ← Rb
	A	ADD	13	Add	ADD Ra, Rb, Rc	Ra ← Rb + Rc
	A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra ← Rb - Rc
	A	MUL	15	Multiply	MUL Ra, Rb, Rc	Ra ← Rb * Rc
	A	DIV	16	Divide	DIV Ra, Rb, Rc	Ra ← Rb / Rc
	A	AND	18	And	AND Ra, Rb, Rc	Ra ← Rb and Rc
	A	OR	19	Or	OR Ra, Rb, Rc	Ra ← Rb or Rc
	A	XOR	1A	Exclusive Or	XOR Ra, Rb, Rc	Ra ← Rb xor Rc
	A	ROL[3]	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
Jump	J	JEQ	20	Jump (=)	JEQ Cx	if SW(=) PC ← PC + Cx
	J	JNE	21	Jump (!=)	JNE Cx	if SW(=) PC ← PC + Cx
	J	JLT	22	Jump (<)	JLT Cx	if SW(=) PC ← PC + Cx
	J	JGT	23	Jump (>)	JGT Cx	if SW(=) PC ← PC + Cx
	J	JLE	24	Jump (<=)	JLE Cx	if SW(=) PC ← PC + Cx
	J	JGE	25	Jump (>=)	JGE Cx	if SW(=) PC ← PC + Cx
	J	JMP	26	Jump (unconditional)	JMP Cx	PC ← PC + Cx
	J	SWI	2A	Software Interrupt	SWI Cx	LR ← PC; PC ← Cx
	J	JSUB	2B	Jump Subroutine	JSUB Cx	LR ← PC; PC ← PC + Cx
	J	RET	2C	Return	RET	PC ← LR
Push / Pop	A	PUSH	30	Push word	PUSH Ra	SP-=4; [SP] = Ra;
	A	POP	31	Pop word	POP Ra	Ra = [SP]; SP+=4;
	A	PUSHB	32	Push byte	PUSHB Ra	SP==; [SP] = Ra; (byte)
	A	POPB	33	Pop byte	POPB Ra	Ra = [SP]; SP++; (byte)

Figure 3.4: Table 2.2: CPU0 instruction table

Type	Format	Instruction	OP	Explain	Grammar	Semantic
Floating point operation	A	FADD	41	Floating-point addition	FADD Ra, Rb, Rc	Ra = Rb + Rc
Floating point operation	A	FSUB	42	Floating-point subtraction	FSUB Ra, Rb, Rc	Ra = Rb + Rc
Floating point operation	A	FMUL	43	Floating-point multiplication	FMUL Ra, Rb, Rc	Ra = Rb * Rc
Floating point operation	A	FADD	44	Floating-point division	FDIV Ra, Rb, Rc	Ra = Rb / Rc
Interrupt handling	J	IRET	2D	Interrupt return	IRET	PC = LR; INT 0

Figure 3.5: Table 2.3 CPU0_v2 instruction table

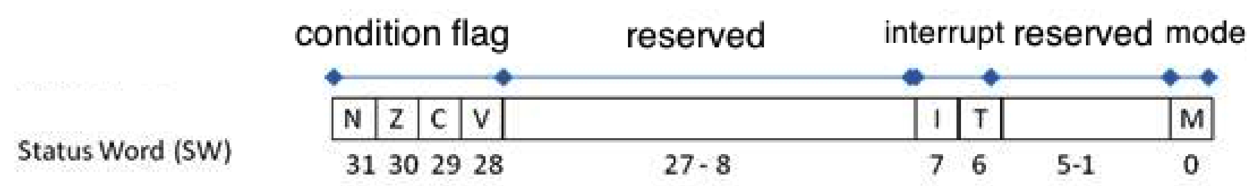
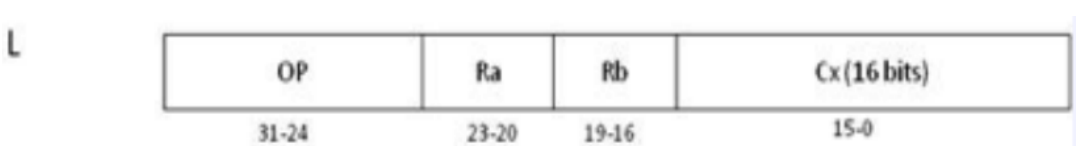


Figure 3.6: Fig 2.3: CPU0 status register



L	LDI	08	Load immediate	LDI Ra, Cx	Ra ← Cx

Figure 3.7: Fig 2.4 cpu0 ldi instruction

ADDIU Add Immediate Unsigned ADDIU

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

32	T:	$GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \mid\mid immediate_{15...0}$
64	T:	$temp \leftarrow GPR[rs] + (immediate_{15})^{48} \mid\mid immediate_{15...0}$ $GPR[rt] \leftarrow (temp_{31})^{32} \mid\mid temp_{31...0}$

Exceptions:

None

Figure 3.8: Fig 2.5 Mips addiu instruction format

it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code. The AST is optionally converted to a new representation for optimization, and the optimizer and back end are run on the code.

Figure 3.9: Fig 2.6 Tree major components of a Three Phase Compiler

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target. The back end (also known as the code generator) then maps the code onto the target instruction set. In addition to making correct code, it is responsible for generating good code that takes advantage of unusual features of the supported architecture. Common parts of a compiler back end include instruction selection, register allocation, and instruction scheduling.

This model applies equally well to interpreters and JIT compilers. The Java Virtual Machine (JVM) is also an implementation of this model, which uses Java bytecode as the interface between the front end and optimizer.

The most important win of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown in Figure 2.7.

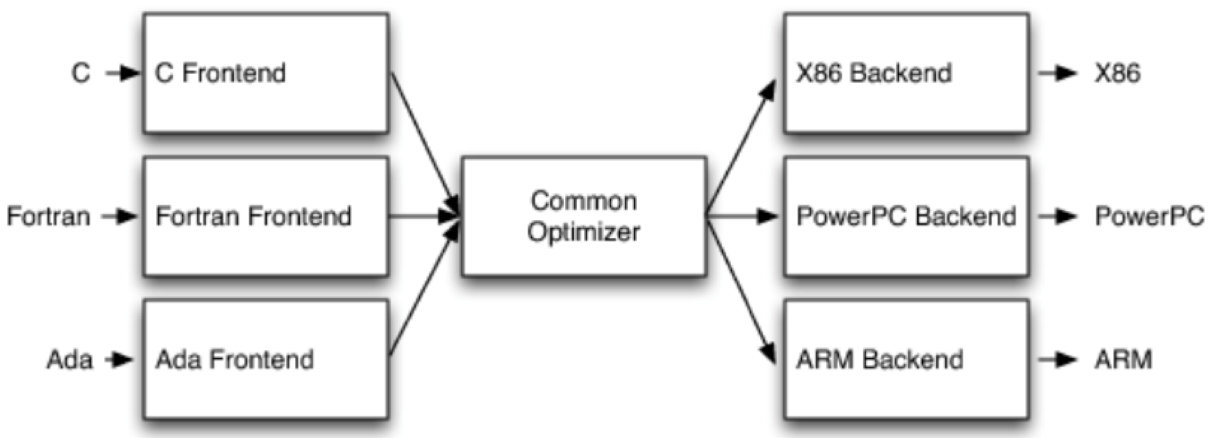


Figure 3.10: Fig 2.7 Retargetablity

With this design, porting the compiler to support a new source language (e.g., Algol or BASIC) requires implementing a new front end, but the existing optimizer and back end can be reused. If these parts weren't separated, implementing a new source language would require starting over from scratch,so supporting N targets and M source languages would need N*M compilers.

Another advantage of the three-phase design (which follows directly from retargetability) is that the compiler serves a broader set of programmers than it would if it only supported one source language and one target. For an open source project, this means that there is a larger community of potential contributors to draw from, which naturally leads to more enhancements and improvements to the compiler. This is the reason why open source compilers that serve many communities (like GCC) tend to generate better optimized machine code than narrower compilers like FreePASCAL. This isn't the case for proprietary compilers, whose quality is directly related to the project's budget. For example, the Intel ICC Compiler is widely known for the quality of code it generates, even though it serves a narrow audience.

A final major win of the three-phase design is that the skills required to implement a front end are different than those required for the optimizer and back end. Separating these makes it easier for a "front-end person" to enhance and maintain their part of the compiler. While this is a social issue, not a technical one, it matters a lot in practice, particularly for open source projects that want to reduce the barrier to contributing as much as possible.

The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represent code in the compiler. LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics. To make this concrete, here is a simple example of a .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
  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
}
This 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 you can see from 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 like add, subtract, compare, and branch. These instructions are in three address form, which means that they take some number of inputs and produce a result in a different register. LLVM IR supports labels and generally looks like a weird form of assembly language.

Unlike most RISC instruction sets, LLVM is strongly typed with a simple type system (e.g., i32 is a 32-bit integer, i32** is a pointer to pointer to 32-bit integer) and some details of the machine are abstracted away. For example, the calling convention is abstracted through call and ret instructions and explicit arguments. Another significant difference from machine code is that the LLVM IR doesn't use a fixed set of named registers, it uses an infinite set of temporaries named with a % character.

Beyond being implemented as a language, LLVM IR is actually defined in three isomorphic forms: the textual format above, an in-memory data structure inspected and modified by optimizations themselves, and an efficient and dense on-disk binary "bitcode" format. The LLVM Project also provides tools to convert the on-disk format from text to binary: llvm-as assembles the textual .ll file into a .bc file containing the bitcode goop and llvm-dis turns a .bc file into a .ll file.

The intermediate representation of a compiler is interesting because it can be a "perfect world" for the compiler optimizer: unlike the front end and back end of the compiler, the optimizer isn't constrained by either a specific source language or a specific target machine. On the other hand, it has to serve both well: it has to be designed to be easy for a front end to generate and be expressive enough to allow important optimizations to be performed for real targets.

3.3 Target Description td

The "mix and match" approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets. This brings up another challenge: each shared component needs to be able to reason about target specific properties in a generic way. For example, a shared register allocator needs to know the register file of each target and the constraints that exist between instructions and their register operands. LLVM's solution to this is for each target to provide a target description in a declarative domain-specific language (a set of .td files) processed by the tblgen tool. The (simplified) build process for the x86 target is shown in Figure 2.8.

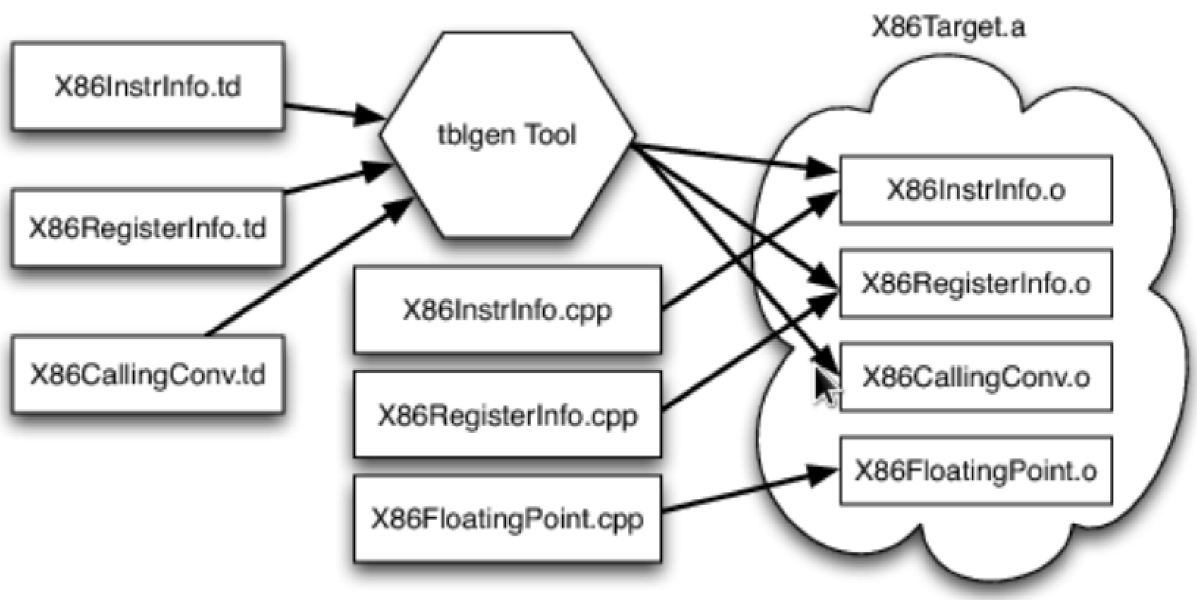


Figure 3.11: Fig 2.8 Simplified x86 Target Definition

The different subsystems supported by the .td files allow target authors to build up the different pieces of their target. For example, the x86 back end defines a register class that holds all of its 32-bit registers named "GR32" (in the .td files, target specific definitions are all caps) like this:

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

3.4 Write td (Target Description)

The llvm using .td file (Target Description) to describe register and instruction format. After finish the .td files, llvm can generate C++ files (*.inc) by llvm-tblgen tools. The *.inc file is a text file (C++ file) with table driven in concept. http://llvm.org/docs/TableGenFundamentals.html is the web site.

Every back end has a target td which define it's own target information. Td is like C++ in syntax. For example we have Cpu0.td as follows,

```
/===-- 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.
//
//===----------------------------------------------------------------------===//
// This is the top level entry point for the Cpu0 target.
//===----------------------------------------------------------------------===//

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

include "llvm/Target/Target.td"
//===----------------------------------------------------------------------===//
// Register File, Calling Conv, Instruction Descriptions
//===----------------------------------------------------------------------===//

include "Cpu0RegisterInfo.td"
include "Cpu0Schedule.td"
include "Cpu0InstrInfo.td"

def Cpu0InstrInfo : InstrInfo;

def Cpu0 : Target {
// def Cpu0InstrInfo : InstrInfo as before.
  let InstructionSet = Cpu0InstrInfo;
}
```

The registers td named Cpu0RegisterInfo.td included by Cpu0.td defined as follows,

```
//  Cpu0RegisterInfo.td
//===----------------------------------------------------------------------===//
//  Declarations that describe the CPU0 register file
//===----------------------------------------------------------------------===//
// We have banks of 16 registers each.
class Cpu0Reg<string n> : Register<n> {
  field bits<4> Num;
  let Namespace = "Cpu0";
}

// Cpu0 CPU Registers
class Cpu0GPRReg<bits<4> num, string n> : Cpu0Reg<n> {
  let Num = num;
}
//===----------------------------------------------------------------------===//
//  Registers
//===----------------------------------------------------------------------===//
let Namespace = "Cpu0" in {
  // General Purpose Registers
  def ZERO : Cpu0GPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
  def AT   : Cpu0GPRReg< 1, "AT">,   DwarfRegNum<[1]>;
  def V0   : Cpu0GPRReg< 2, "2">,    DwarfRegNum<[2]>;
  def V1   : Cpu0GPRReg< 3, "3">,    DwarfRegNum<[3]>;
  def A0   : Cpu0GPRReg< 4, "4">,    DwarfRegNum<[6]>;
  def A1   : Cpu0GPRReg< 5, "5">,    DwarfRegNum<[7]>;
  def T9   : Cpu0GPRReg< 6, "6">,    DwarfRegNum<[6]>;
```

```
  def S0   : Cpu0GPRReg< 7, "7">,    DwarfRegNum<[7]>;
  def S1   : Cpu0GPRReg< 8, "8">,    DwarfRegNum<[8]>;
  def S2   : Cpu0GPRReg< 9, "9">,    DwarfRegNum<[9]>;
  def GP   : Cpu0GPRReg< 10, "GP">,  DwarfRegNum<[10]>;
  def FP   : Cpu0GPRReg< 11, "FP">,  DwarfRegNum<[11]>;
  def SW   : Cpu0GPRReg< 12, "SW">,   DwarfRegNum<[12]>;
  def SP   : Cpu0GPRReg< 13, "SP">,   DwarfRegNum<[13]>;
  def LR   : Cpu0GPRReg< 14, "LR">,   DwarfRegNum<[14]>;
  def PC   : Cpu0GPRReg< 15, "PC">,   DwarfRegNum<[15]>;
//  def MAR  : Cpu0GPRReg< 16, "MAR">,  DwarfRegNum<[16]>;
//  def MDR  : Cpu0GPRReg< 17, "MDR">,  DwarfRegNum<[17]>;
}
//===----------------------------------------------------------------------===//
// Register Classes
//===----------------------------------------------------------------------===//
def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  // Return Values and Arguments
  V0, V1, A0, A1,
  // Not preserved across procedure calls
  T9,
  // Callee save
  S0, S1, S2,
  // Reserved
  ZERO, AT, GP, FP, SW, SP, LR, PC)>;
```

In C++ the data layout is declared by class. Declare meaning tell the variable layout. The define meaning instance the variable (create the real variable). For example,

```
class Date {         // declare Date
    int year, month, day;
};
Date date;         // define(instance) date
```

Just like C++ class, the keyword "class" is used for declaring data structure layout. Cpu0Reg<string n> declare a derived class from Register<n> which is declared by llvm already, and the n is the argument which type is string. In addition to Register class fields, Cpu0Reg add a new field Num of type 4 bits. Namespace same as C++'s namespace. "Def" is used by define(instance) a concrete variable.

As above, we define a ZERO register which type is Cpu0GPRReg, it's field Num is 0 (4 bits) and field n is "ZERO" (declared in Register class). Note the use of "let" expressions to override values that are initially defined in a superclass. For example, let Namespace = "Cpu0" in class Cpu0Reg of our example, will override Namespace declared in Register class. We also define CPURegs is a variable for type of RegisterClass, where the RegisterClass is llvm built-in class. The RegisterClass type is a set/group of Register, so we define a set of Register in CPURegs variable.

I named the instructions td as Cpu0InstrInfo.td which contents as follows,

```
/===- 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.
//
//===----------------------------------------------------------------------===//
//===----------------------------------------------------------------------===//
```

```
// Instruction format superclass
//===----------------------------------------------------------------------===//
 include "Cpu0InstrFormats.td"
//===----------------------------------------------------------------------===//
// Cpu0 profiles and nodes
//===----------------------------------------------------------------------===//
def SDT_Cpu0Ret          : SDTypeProfile<0, 1, [SDTCisInt<0>]>;
// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDT_Cpu0Ret, [SDNPHasChain,
                     SDNPOptInGlue]>;
//===----------------------------------------------------------------------===//
// Cpu0 Operand, Complex Patterns and Transformations Definitions.
//===----------------------------------------------------------------------===//
def simm16      : Operand<i32> {
  let DecoderMethod= "DecodeSimm16";
}
// Address operand
def mem : Operand<i32> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops CPURegs, simm16);
  let EncoderMethod = "getMemEncoding";
}
// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addiu
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;

// Cpu0 Address Mode! SDNode frameindex could possibily 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();
}]>;
// Load/Store PatFrags.
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 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;
}
```

```
// Move immediate imm16 to register ra.
class MoveImm<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, $imm16"),
     [(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
  let rb = 0;
  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.
multiclass LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
                   bit Pseudo = 0> {
  def #NAME# : LoadM<op, instr_asm, OpNode, CPURegs, 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>;
}
//===----------------------------------------------------------------------===//
// Instruction definition
//===----------------------------------------------------------------------===//
//===----------------------------------------------------------------------===//
// Cpu0I Instructions
//===----------------------------------------------------------------------===//
/// Load and Store Instructions
///  aligned
defm LW      : LoadM32<0x00,  "lw",  load_a>;
defm ST      : StoreM32<0x01, "st",  store_a>;

/// Arithmetic Instructions (ALU Immediate)
//def LDI     : MoveImm<0x08, "ldi", add, simm16, immSExt16, CPURegs>;
// 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, isCodeGenOnly=1,
    isBarrier=1, hasCtrlDep=1 in
  def RET : FJ <0x2C, (outs), (ins CPURegs:$target),
               "ret\t$target", [(Cpu0Ret CPURegs:$target)], IIBranch>;

//===----------------------------------------------------------------------===//
//  Arbitrary patterns that map to one or more instructions
//===----------------------------------------------------------------------===//
// Small immediates

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

The Cpu0InstrFormats.td is included by Cpu0InstInfo.td as follows,

```
//===-- 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 FrmFR     : Format<4>;
def FrmFI     : Format<5>;
def FrmOther  : Format<6>; // Instruction w/ a custom format

// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
               InstrItinClass itin, Format f>: Instruction
{
```

```
  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> imm12;

  let Opcode = op;

  let Inst{23-20} = ra;
  let Inst{19-16} = rb;
  let Inst{15-12} = rc;
  let Inst{11-0}  = imm12;
}

//===----------------------------------------------------------------------===//
// Format I 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;
}

//===----------------------------------------------------------------------===//
// 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;
}
```

ADDiu is class ArithLogicI inherited from FL, can expand and get member value 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 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;
}

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

Expand with FL further,

```
 :  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>

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;
}

So,
op = 0x09
outs = CPURegs:$ra
ins = CPURegs:$rb,simm16:$imm16
asmstr = "addiu\t$ra, $rb, $imm16"
pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
itin = IIAlu

Members are,
ra = CPURegs:$ra
rb = CPURegs:$rb
imm16 = simm16:$imm16
Opcode = 0x09;
Inst{23-20} = CPURegs:$ra;
Inst{19-16} = CPURegs:$rb;
Inst{15-0}  = simm16:$imm16;
```

Expand with Cpu0Inst further,

```
class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
         InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>

class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
               InstrItinClass itin, Format f>: Instruction
{
  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;
}

So,
outs = CPURegs:$ra
ins = CPURegs:$rb,simm16:$imm16
asmstr = "addiu\t$ra, $rb, $imm16"
pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
itin = IIAlu
f =  FrmL

Members are,
Inst{31-24} = 0x09;
OutOperandList = CPURegs:$ra
InOperandList  = CPURegs:$rb,simm16:$imm16
AsmString = "addiu\t$ra, $rb, $imm16"
Pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
Itinerary = IIAlu

Summary with all members are,
// Inherited from parent like Instruction
Namespace = "Cpu0";
DecoderNamespace = "Cpu0";
Inst{31-24} = 0x08;
Inst{23-20} = CPURegs:$ra;
Inst{19-16} = CPURegs:$rb;
Inst{15-0}  = simm16:$imm16;
OutOperandList = CPURegs:$ra
InOperandList  = CPURegs:$rb,simm16:$imm16
AsmString = "addiu\t$ra, $rb, $imm16"
Pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
Itinerary = IIAlu
// From Cpu0Inst
Opcode = 0x09;
// From FL
ra = CPURegs:$ra
rb = CPURegs:$rb
imm16 = simm16:$imm16
```

It's a lousy process. Similarly, LW and ST instruction definition can be expanded in this way. Please notify the Pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))] which include keyword "add". We will use it in DAG transformations later.

3.5 Write cmake file

In Target/Cpu0 directory, we have 2 files CMakeLists.txt and LLVMBuild.txt, contents as follows,

```
# CMakeLists.txt
# Our td all in Cpu0.td, Cpu0RegisterInfo.td and Cpu0InstrInfo.td included in Cpu0.td
set(LLVM_TARGET_DEFINITIONS Cpu0.td)
```

```
# Generate Cpu0GenRegisterInfo.inc and Cpu0GenInstrInfo.inc which include by your hand code C++ files
# Cpu0GenRegisterInfo.inc came from Cpu0RegisterInfo.td, Cpu0GenInstrInfo.inc came from Cpu0InstrInfo
tablegen(LLVM Cpu0GenRegisterInfo.inc -gen-register-info)
tablegen(LLVM Cpu0GenInstrInfo.inc -gen-instr-info)

# Used by llc
add_public_tablegen_target(Cpu0CommonTableGen)

# Cpu0CodeGen should match with LLVMBuild.txt Cpu0CodeGen
add_llvm_target(Cpu0CodeGen
  Cpu0TargetMachine.cpp
  )
# Should match with "subdirectories =  MCTargetDesc TargetInfo" in LLVMBuild.txt
add_subdirectory(TargetInfo)
add_subdirectory(MCTargetDesc)
```

CMakeLists.txt is the make information for cmake, # is comment.

```
;===- ./lib/Target/Cpu0/LLVMBuild.txt ---------------------------*- Conf -*--===;
;
;                     The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;===------------------------------------------------------------------------===;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;===------------------------------------------------------------------------===;

# Following comments extracted from http://llvm.org/docs/LLVMBuild.html

[common]
subdirectories =  MCTargetDesc TargetInfo

[component_0]
# TargetGroup components are an extension of LibraryGroups, specifically for defining LLVM targets (w
type = TargetGroup
# The name of the component should always be the name of the target. (should match "def Cpu0 : Target
name = Cpu0
# Cpu0 component is located in directory Target/
parent = Target
# Whether this target defines an assembly parser, assembly printer, disassembler, and supports JIT co
#has_asmparser = 1
#has_asmprinter = 1
#has_disassembler = 1
#has_jit = 1

[component_1]
# component_1 is a Library type and name is Cpu0CodeGen. After build it will in lib/libLLVMCpu0CodeGe
type = Library
name = Cpu0CodeGen
# Cpu0CodeGen component(Library) is located in directory Cpu0/
```

```
parent = Cpu0
# If given, a list of the names of Library or LibraryGroup components which must also be linked in wh
# dependencies for this component. When tools are built, the build system will include the transitive
# the tool needs.
required_libraries = CodeGen Core MC Cpu0Desc Cpu0Info SelectionDAG Support Target
# All LLVMBuild.txt in Target/Cpu0 and subdirectory use 'add_to_library_groups = Cpu0'
add_to_library_groups = Cpu0
```

LLVMBuild.txt files are written in a simple variant of the INI or configuration file format. # is comment in both 2 files. I explain the setting for these 2 files in comments. Please spend a little time to read it.

Both CMakeLists.txt and LLVMBuild.txt also exist in sub-directories MCTargetDesc and TargetInfo. Their contents indicate they will generate Cpu0Desc and Cpu0Info libraries. After build, you will find libLLVM-Cpu0CodeGen.a, libLLVMCpu0Desc .a and libLLVMCpu0Info .a 3 libraries in lib/ of your build directory. http://llvm.org/docs/CMake.html?highlight=cmake and http://llvm.org/docs/LLVMBuild.html are their web site.

3.6 Target Registration

You must also register your target with the TargetRegistry, which is what other LLVM tools use to be able to lookup and use your target at runtime. The TargetRegistry can be used directly, but for most targets there are helper templates which should take care of the work for you.

All targets should declare a global Target object which is used to represent the target during registration. Then, in the target's TargetInfo library, the target should define that object and use the RegisterTarget template to register the target. For example, the file TargetInfo/Cpu0TargetInfo.cpp register TheCpu0Target for big endian and TheCpu0elTarget for little endian, as follows.

```
Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;
extern "C" void LLVMInitializeCpu0TargetInfo() {
  RegisterTarget<Triple::cpu0,
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "Cpu0");

  RegisterTarget<Triple::cpu0el,
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "Cpu0el");
}
```

Files Cpu0TargetMachine.cpp and MCTargetDesc/Cpu0MCTargetDesc.cpp just define the empty initialize function since we register nothing in them for this moment.

```
extern "C" void LLVMInitializeCpu0Target() {
}

extern "C" void LLVMInitializeCpu0TargetMC() {
}
```

http://llvm.org/docs/WritingAnLLVMBackend.html#TargetRegistration for reference.

3.7 Build libraries and td

I put my llvm3.1 source code in /usr/local/llvm/3.1/src and have llvm3.1 release-build in /usr/local/llvm/3.1/configure_release_bu

Except directory src/lib/Target/Cpu0, there are a couple of files modified to support cpu0 new Target. Please check files in src_files_modify/src/. You can search cpu0 without case sensitive to find the modified files by command,

```
[Gamma@localhost cmake_debug_build]$ grep -R -i "cpu0" ../src/
../src/CMakeLists.txt:  Cpu0
../src/lib/Target/LLVMBuild.txt:subdirectories = ARM CellSPU CppBackend Hexagon MBlaze MSP430 Mips Cp
...
../src/lib/MC/MCELFStreamer.cpp:    case MCSymbolRefExpr::VK_Cpu0_TLSGD:
...
../src/lib/MC/MCDwarf.cpp:  // AT_language, a 4 byte value.  We use DW_LANG_Cpu0_Assembler as the dwa
../src/lib/MC/MCDwarf.cpp: // MCOS->EmitIntValue(dwarf::DW_LANG_Cpu0_Assembler, 2);
../src/lib/Support/Triple.cpp:  case cpu0:    return "cpu0";
 ...
../src/include/llvm/Support/ELF.h:  EM_LATTICEMICO32 = 138, // RISC processor for Lattice CPU0 archit
...
```

You can update your llvm working copy by,

```
cp -rf LLVMBackendTutorial/src_files_modified/src/*   yourllvm/workingcopy/sourcedir/.
```

Now, run the cmake and make command to build td (the following cmake command is for my setting),

```
[Gamma@localhost cmake_debug_build]$ cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang -DCM

-- Targeting Cpu0
...
-- Targeting XCore
-- Configuring done
-- Generating done
-- Build files have been written to: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build

[Gamma@localhost cmake_debug_build]$ make
...
[100%] Built target gtest_main
```

After build, you can type command llc –version to find the cpu0 backend,

```
[Gamma@localhost cmake_debug_build]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc --vers
LLVM (http://llvm.org/):
  LLVM version 3.1svn
  DEBUG build with assertions.
  Built Sep 21 2012 (18:27:58).
  Default target: x86_64-unknown-linux-gnu
  Host CPU: penryn

  Registered Targets:
    arm      - ARM
    cellspu  - STI CBEA Cell SPU [experimental]
    cpp      - C++ backend
    cpu0     - Cpu0
    cpu0el   - Cpu0el
...
```

The "llc -version" can display "cpu0" and "cpu0el" message, because the following code from file TargetInfo/Cpu0TargetInfo.cpp what in section Target Registration we made. List them as follows again,

```
// Cpu0TargetInfo.cpp
Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;

extern "C" void LLVMInitializeCpu0TargetInfo() {
  RegisterTarget<Triple::cpu0,
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "Cpu0");
```

```
  RegisterTarget<Triple::cpu0el,
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "Cpu0el");
}
```

Now try to do llc command to compile input file ch3.cpp as follows,

```
// ch3.cpp
int main()
{
        return 0;
}
```

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

```
[Gamma@localhost InputFiles]$ clang -c ch3.cpp -emit-llvm -o ch3.bc
```

Next step, transfer bitcode .bc to human readable text format as follows,

```
[Gamma@localhost InputFiles]$ llvm-dis ch3.bc -o ch3.ll

// ch3.ll
; 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-f32:32:32-f64:64:64-v64
target triple = "x86_64-unknown-linux-gnu"

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

Now, compile ch3.bc into ch3.cpu0.s, we get the error message as follows,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
llc: /usr/local/llvm/3.1.test/cpu0/1/src/tools/llc/llc.cpp:456: int main(int, char **): Assertion 'ta
Stack dump:
0.        Program arguments: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -r
Aborted (core dumped)
```

Currently we just define target td files (Cpu0.td, Cpu0RegisterInfo.td, ...). According to LLVM structure, we need to define our target machine and include those td related files. The error message say we didn't define our target machine.

CHAPTER

FOUR

BACK END STRUCTURE

I will introduce the back end class inherit tree and class members first. Next, following the back end structure, add individual class implementation in each section. There are compiler knowledge like DAG (Directed-Acyclic-Graph) and instruction selection needed in this chapter. I explain these knowledge just when needed. At the end of this chapter, we will have a back end to compile llvm intermediate code into cpu0 assembly code.

4.1 TargetMachine structure

Your back end should define a TargetMachine class, for example, we define the Cpu0TargetMachine class. Cpu0TargetMachine class contains it's own instruction class, frame/stack class, DAG (Directed-Acyclic-Graph) class, and register class. The Cpu0TargetMachine contents as follows,

```
//- TargetMachine.h
class TargetMachine {
  TargetMachine(const TargetMachine &);    // DO NOT IMPLEMENT
  void operator=(const TargetMachine &);   // DO NOT IMPLEMENT

public:
  // Interfaces to the major aspects of target machine information:
  // -- Instruction opcode and operand information
  // -- Pipelines and scheduling information
  // -- Stack frame information
  // -- Selection DAG lowering information
  //
  virtual const TargetInstrInfo         *getInstrInfo() const { return 0; }
  virtual const TargetFrameLowering *getFrameLowering() const { return 0; }
  virtual const TargetLowering    *getTargetLowering() const { return 0; }
  virtual const TargetSelectionDAGInfo *getSelectionDAGInfo() const{ return 0; }
  virtual const TargetData              *getTargetData() const { return 0; }
  ...
  /// getSubtarget - This method returns a pointer to the specified type of
  /// TargetSubtargetInfo.  In debug builds, it verifies that the object being
  /// returned is of the correct type.
  template<typename STC> const STC &getSubtarget() const {
    return *static_cast<const STC*>(getSubtargetImpl());
  }

}

//- TargetMachine.h
class LLVMTargetMachine : public TargetMachine {
protected: // Can only create subclasses.
```

```
  LLVMTargetMachine(const Target &T, StringRef TargetTriple,
                    StringRef CPU, StringRef FS, TargetOptions Options,
                    Reloc::Model RM, CodeModel::Model CM,
                    CodeGenOpt::Level OL);
        ...
};

class Cpu0TargetMachine : public LLVMTargetMachine {
  Cpu0Subtarget       Subtarget;
  const TargetData    DataLayout; // Calculates type size & alignment
  Cpu0InstrInfo       InstrInfo;        //- Instructions
  Cpu0FrameLowering   FrameLowering;        //- Stack(Frame) and Stack direction
  Cpu0TargetLowering  TLInfo;        //- Stack(Frame) and Stack direction
  Cpu0SelectionDAGInfo TSInfo;        //- Map .bc DAG to backend DAG
public:
  virtual const Cpu0InstrInfo   *getInstrInfo()     const
  { return &InstrInfo; }
  virtual const TargetFrameLowering *getFrameLowering()     const
  { return &FrameLowering; }
  virtual const Cpu0Subtarget   *getSubtargetImpl() const
  { return &Subtarget; }
  virtual const TargetData      *getTargetData()    const
  { return &DataLayout;}
   virtual const Cpu0TargetLowering *getTargetLowering() const {
    return &TLInfo;
  }

  virtual const Cpu0SelectionDAGInfo* getSelectionDAGInfo() const {
    return &TSInfo;
  }
};

//- TargetInstInfo.h
class TargetInstrInfo : public MCInstrInfo {
  TargetInstrInfo(const TargetInstrInfo &);  // DO NOT IMPLEMENT
  void operator=(const TargetInstrInfo &);   // DO NOT IMPLEMENT
public:
        ...
}

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

//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifdef GET_INSTRINFO_HEADER
#undef 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);
};
```

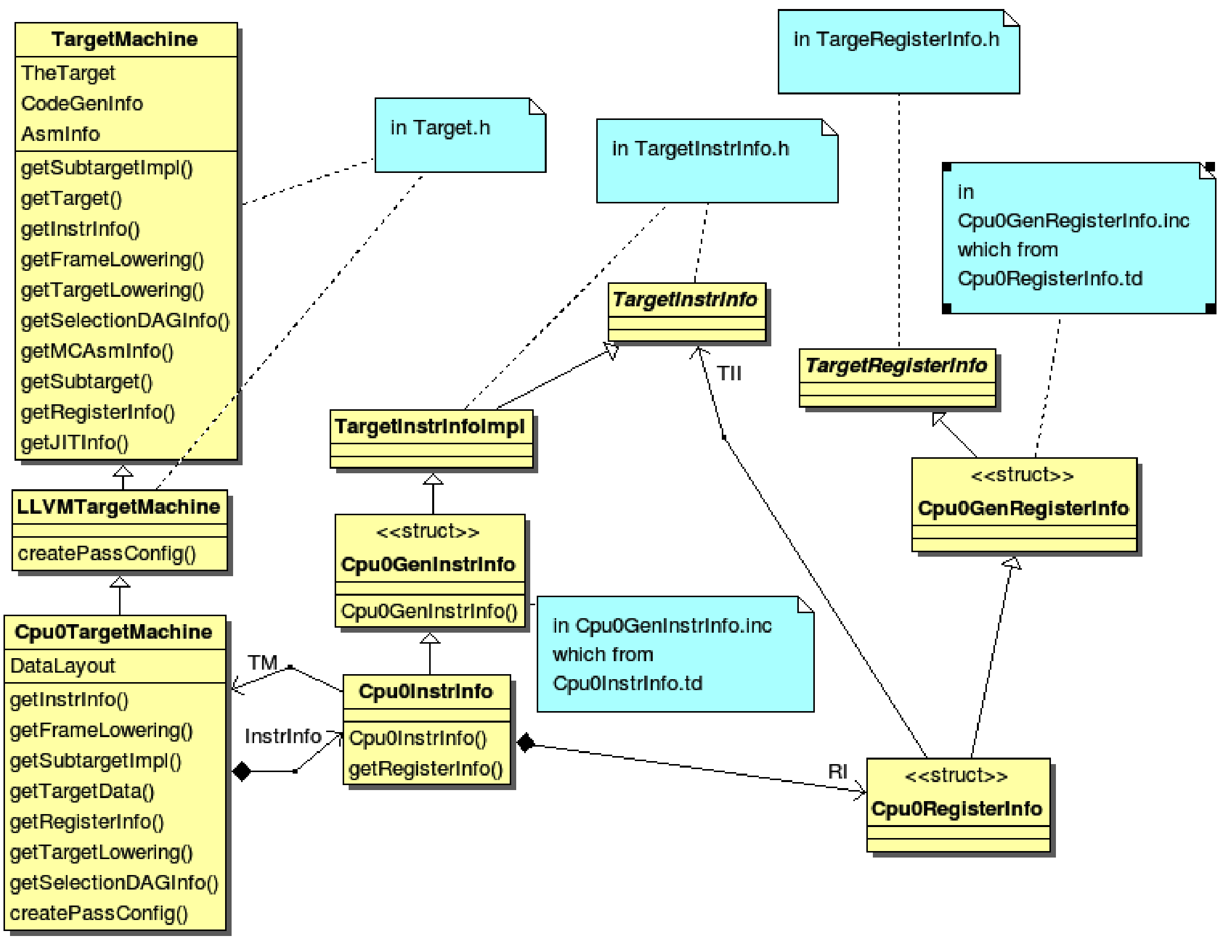


Figure 4.1: Fig 3.1 TargetMachine class diagram 1

The Cpu0TargetMachine inherit tree is TargetMachine <- LLVMTargetMachine <- Cpu0TargetMachine. Cpu0TargetMachine has class Cpu0Subtarget, Cpu0InstrInfo, Cpu0FrameLowering, Cpu0TargetLowering and Cpu0SelectionDAGInfo. Class Cpu0Subtarget, Cpu0InstrInfo, Cpu0FrameLowering, Cpu0TargetLowering and Cpu0SelectionDAGInfo are inherited from parent class TargetSubtargetInfo, TargetInstrInfo, TargetFrameLowering, TargetLowering and TargetSelectionDAGInfo.

Fig 3.1 shows Cpu0TargetMachine inherit tree and it's Cpu0InstrInfo class inherit tree. Cpu0TargetMachine contains Cpu0InstrInfo and ... other class. Cpu0InstrInfo contains Cpu0RegisterInfo class, RI. Cpu0InstrInfo.td and Cpu0RegisterInfo.td will generate Cpu0GenInstrInfo.inc and Cpu0GenRegisterInfo.inc which contain some member functions implementation for class Cpu0InstrInfo and Cpu0RegisterInfo.

Fig 3.2 as below shows Cpu0TargetMachine contains class TSInfo: Cpu0SelectionDAGInfo, FrameLowering:

Cpu0FrameLowering, Subtarget: Cpu0Subtarget and TLInfo: Cpu0TargetLowering.

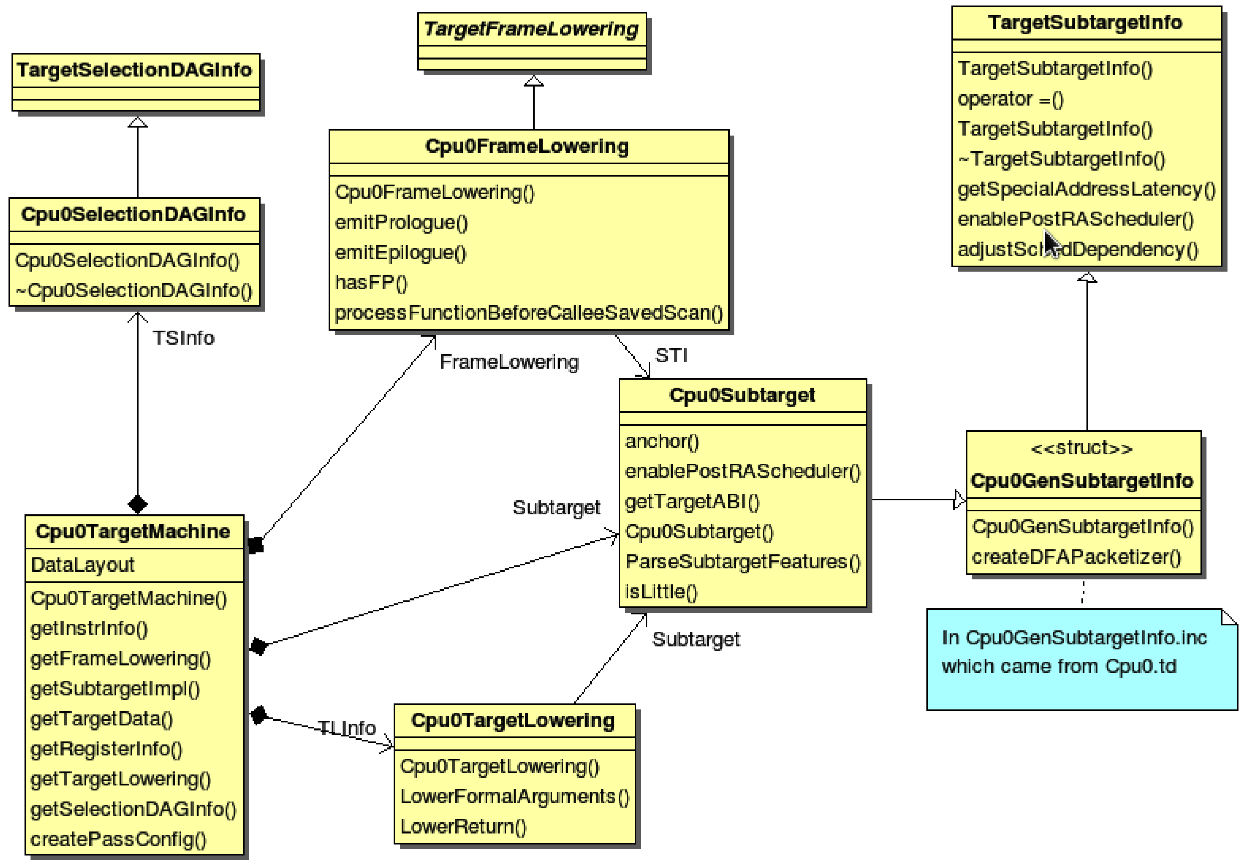


Figure 4.2: Fig 3.2 TargetMachine class diagram 2

Fig 3.3 shows some members and operators (member function) of the parent class TargetMachine's. Fig 3.4 as below shows some members of class InstrInfo, RegisterInfo and TargetLowering. Class DAGInfo is skipped here.

Benefit from the inherit tree structure, we just need to implement few code in instruction, frame/stack, select DAG class. Many code implemented by their parent class. The llvm-tblgen generate Cpu0GenInstrInfo.inc from Cpu0InstrInfo.td. Cpu0InstrInfo.h extract those code it need from Cpu0GenInstrInfo.inc by define "#define GET_INSTRINFO_HEADER". Following is the code fragment from Cpu0GenInstrInfo.inc. Code between "#if def GET_INSTRINFO_HEADER" and "#endif // GET_INSTRINFO_HEADER" will be extracted by Cpu0InstrInfo.h.

```
//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifdef GET_INSTRINFO_HEADER
#undef GET_INSTRINFO_HEADER
namespace llvm {
struct Cpu0GenInstrInfo : public TargetInstrInfoImpl {
  explicit Cpu0GenInstrInfo(int SO = -1, int DO = -1);
};
} // End llvm namespace
#endif // GET_INSTRINFO_HEADER
```

http://llvm.org/docs/WritingAnLLVMBackend.html#TargetMachine

Now, the code in 3/1/Cpu0 add class Cpu0TargetMachine(Cpu0TargetMachine.h and cpp), Cpu0Subtarget (Cpu0Subtarget.h and .cpp), Cpu0InstrInfo (Cpu0InstrInfo.h and .cpp), Cpu0FrameLowering (Cpu0FrameLowering.h

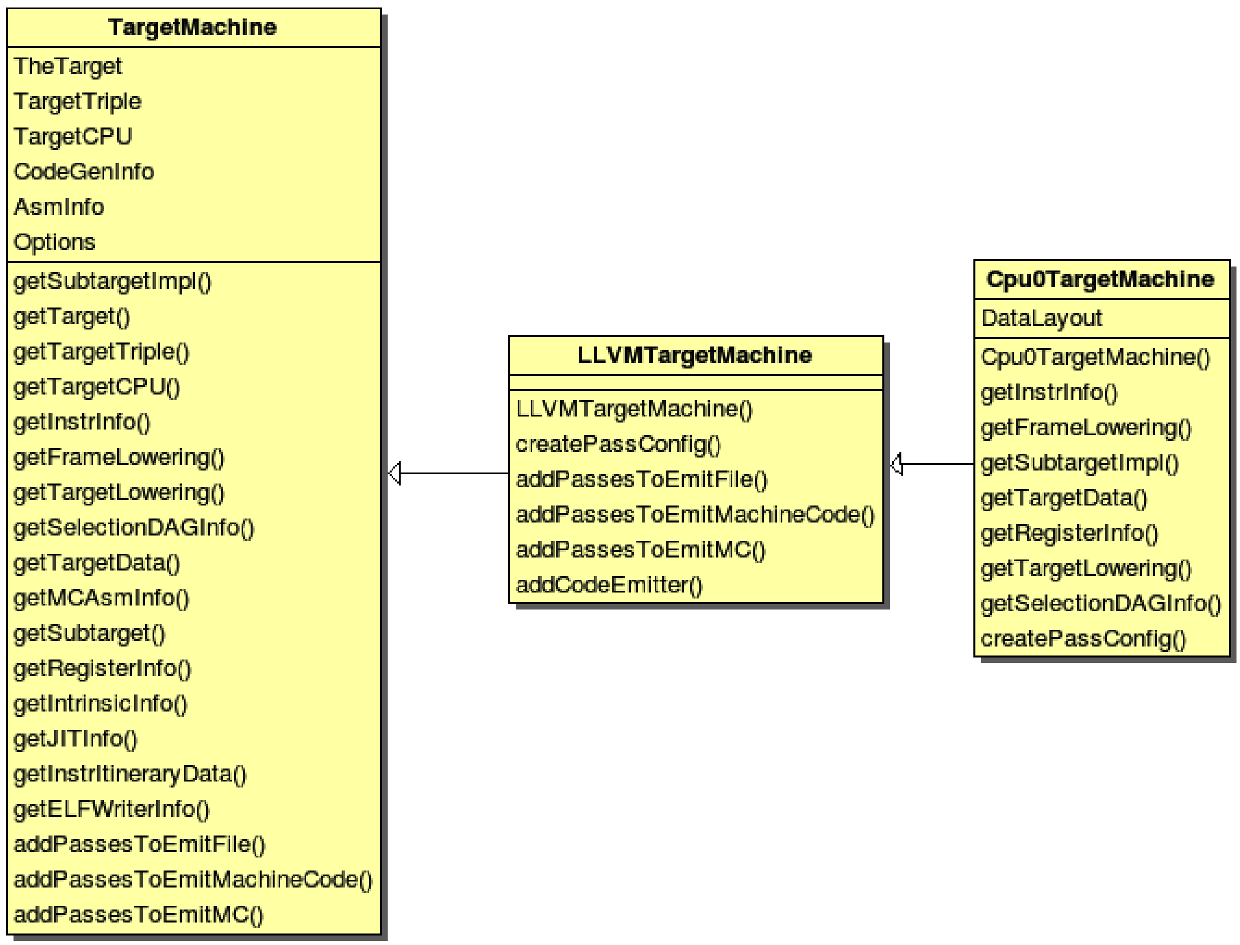


Figure 4.3: Fig 3.3 TargetMachine members and operators

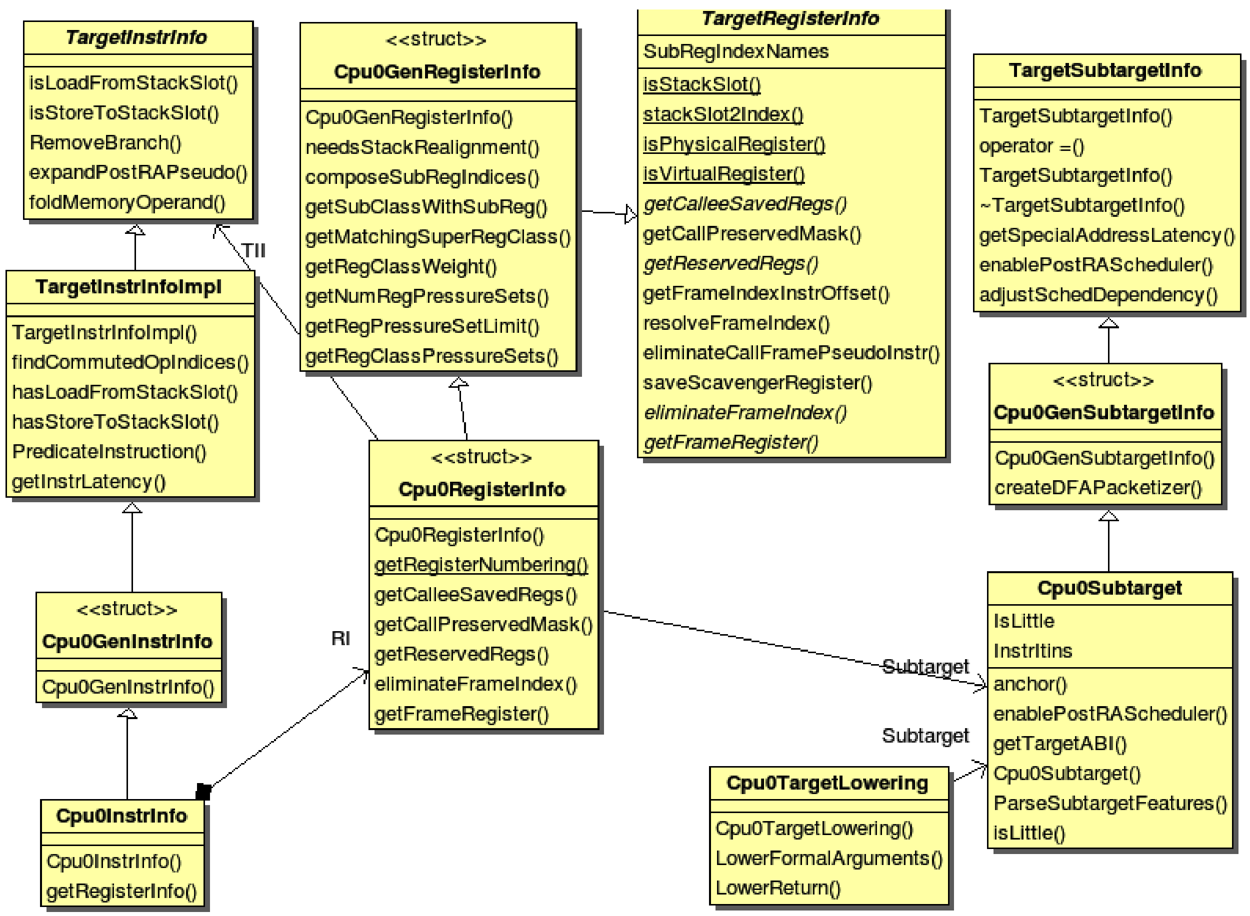


Figure 4.4: Fig 3.4 Other class members and operators

and .cpp), Cpu0TargetLowering (Cpu0ISelLowering.h and .cpp) and Cpu0SelectionDAGInfo (Cpu0SelectionDAGInfo.h and .cpp). CMakeLists.txt modified with those new added *.cpp as follows,

```
# CMakeLists.txt
...
add_llvm_target(Cpu0CodeGen
  Cpu0ISelLowering.cpp
  Cpu0InstrInfo.cpp
  Cpu0FrameLowering.cpp
  Cpu0Subtarget.cpp
  Cpu0TargetMachine.cpp
  Cpu0SelectionDAGInfo.cpp
  )
```

Please take a look for 3/1 code. After that, we build 3/1 by make as chapter 2 (of course, you should remove old Target/Cpu0 and replace with 3/1/Cpu0). You can remove lib/Target/Cpu0/*.inc before do "make" to ensure your code rebuild completely. By remove *.inc, all files those have included .inc will be rebuild, then your Target library will regenerate. Command as follows,

```
[Gamma@localhost cmake_debug_build]$ rm -rf lib/Target/Cpu0/*
```

4.2 Add RegisterInfo

As depicted in Fig 3.1, the Cpu0InstrInfo class should contains Cpu0RegisterInfo. So in 3/2/Cpu0, we add Cpu0RegisterInfo class (Cpu0RegisterInfo.h, Cpu0RegisterInfo.cpp), and Cpu0RegisterInfo class in files Cpu0InstrInfo.h, Cpu0InstrInfo.cpp, Cpu0TargetMachine.h, and modify CMakeLists.txt as follows,

```
// Cpu0InstrInfo.h
class Cpu0InstrInfo : public Cpu0GenInstrInfo {
  Cpu0TargetMachine &TM;
  const Cpu0RegisterInfo RI;
public:
  explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);

  /// 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;

public:
};

// Cpu0InstrInfo.cpp
Cpu0InstrInfo::Cpu0InstrInfo(Cpu0TargetMachine &tm)
  :
    TM(tm),
    RI(*TM.getSubtargetImpl(), *this) {}

const Cpu0RegisterInfo &Cpu0InstrInfo::getRegisterInfo() const {
  return RI;
}

//  Cpu0TargetMachine.h
    virtual const Cpu0RegisterInfo *getRegisterInfo()  const {
      return &InstrInfo.getRegisterInfo();
```

```
    }

# CMakeLists.txt
...
add_llvm_target(Cpu0CodeGen
  ...
  Cpu0RegisterInfo.cpp
  ...
  )
```

Now, let's replace 3/1/Cpu0 with 3/2/Cpu0 for adding register class definition and rebuild. After that, we try to run the llc compile command to see what happen,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
llc: /usr/local/llvm/3.1.test/cpu0/1/src/lib/CodeGen/LLVMTargetMachine.cpp:78: llvm::LLVMTargetMachin
Stack dump:
0.        Program arguments: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -r
Aborted (core dumped)
```

The errors say that we have not Target AsmPrinter. Let's add it in next section.

4.3 Add AsmPrinter

3/3/cpu0 contains the Cpu0AsmPrinter definition. First, I add definitions in Cpu0.td to support AssemblyWriter. Cpu0.td is added with the following fragment,

```
// Cpu0.td
//...
//===----------------------------------------------------------------------===//
// Cpu0 processors supported.
//===----------------------------------------------------------------------===//

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

def : Proc<"cpu032", [FeatureCpu032]>;

def Cpu0AsmWriter : AsmWriter {
  string AsmWriterClassName  = "InstPrinter";
  bit isMCAsmWriter = 1;
}

// 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;
  let AssemblyWriters = [Cpu0AsmWriter];
}
```

As comments indicate, it will generate Cpu0GenAsmWrite.inc which is included by Cpu0InstPrinter.cpp. Cpu0GenAsmWrite.inc has the implementation of Cpu0InstPrinter::printInstruction() and Cpu0InstPrinter::getRegisterName(). Both of these functions can be auto-generated from the information we defined in Cpu0InstrInfo.td and Cpu0RegisterInfo.td. To let these two functions work in our code, the only thing need to do is add a class Cpu0InstPrinter and include them.

File 3/3/Cpu0/InstPrinter/Cpu0InstPrinter.cpp include Cpu0GenAsmWrite.inc and call the auto-generated functions as follows,

```
//  Cpu0InstPrinter.cpp
#include "Cpu0GenAsmWriter.inc"

void Cpu0InstPrinter::printRegName(raw_ostream &OS, unsigned RegNo) const {
//- getRegisterName(RegNo) defined in Cpu0GenAsmWriter.inc which came from Cpu0.td indicate.
  OS << '$' << StringRef(getRegisterName(RegNo)).lower();
}

void Cpu0InstPrinter::printInst(const MCInst *MI, raw_ostream &O,
                                StringRef Annot) {
//- printInstruction(MI, O) defined in Cpu0GenAsmWriter.inc which came from Cpu0.td indicate.
  printInstruction(MI, O);
  printAnnotation(O, Annot);
}
```

Next, add Cpu0AsmPrinter (Cpu0AsmPrinter.h, Cpu0AsmPrinter.cpp), Cpu0MCInstLower (Cpu0MCInstLower.h, Cpu0MCInstLower.cpp), Cpu0BaseInfo.h, Cpu0FixupKinds.h and Cpu0MCAsmInfo (Cpu0MCAsmInfo.h, Cpu0MCAsmInfo.cpp) in sub-directory MCTargetDesc.

Finally, add code in Cpu0MCTargetDesc.cpp to register Cpu0InstPrinter as follows,

```
//  Cpu0MCTargetDesc.cpp
static MCAsmInfo *createCpu0MCAsmInfo(const Target &T, StringRef TT) {
  MCAsmInfo *MAI = new Cpu0MCAsmInfo(T, TT);

  MachineLocation Dst(MachineLocation::VirtualFP);
  MachineLocation Src(Cpu0::SP, 0);
  MAI->addInitialFrameState(0, Dst, Src);

  return MAI;
}

static MCInstPrinter *createCpu0MCInstPrinter(const Target &T,
                                              unsigned SyntaxVariant,
                                              const MCAsmInfo &MAI,
                                              const MCInstrInfo &MII,
                                              const MCRegisterInfo &MRI,
                                              const MCSubtargetInfo &STI) {
  return new Cpu0InstPrinter(MAI, MII, MRI);
}

extern "C" void LLVMInitializeCpu0TargetMC() {
  // Register the MC asm info.
  RegisterMCAsmInfoFn X(TheCpu0Target, createCpu0MCAsmInfo);
  RegisterMCAsmInfoFn Y(TheCpu0elTarget, createCpu0MCAsmInfo);

  // Register the MCInstPrinter.
  TargetRegistry::RegisterMCInstPrinter(TheCpu0Target,
                                        createCpu0MCInstPrinter);
  TargetRegistry::RegisterMCInstPrinter(TheCpu0elTarget,
                                        createCpu0MCInstPrinter);
}
```

Now, it’s time to work with AsmPrinter. According section “2.6 Target Registration”, we can register our AsmPrinter when we need it as follows,

```
// Cpu0AsmPrinter.cpp
// Force static initialization.
extern "C" void LLVMInitializeCpu0AsmPrinter() {
  RegisterAsmPrinter<Cpu0AsmPrinter> X(TheCpu0Target);
  RegisterAsmPrinter<Cpu0AsmPrinter> Y(TheCpu0elTarget);
}
```

The dynamic register mechanism is a good idea, right. Except add the new .cpp files to CMakeLists.txt, please remember to add subdirectory InstPrinter, enable asmprinter, add libraries AsmPrinter and Cpu0AsmPrinter to LLVMBuild.txt as follows,

```
//  LLVMBuild.txt
[common]
subdirectories = InstPrinter MCTargetDesc TargetInfo

[component_0]
...
# Please enable asmprinter
has_asmprinter = 1
...

[component_1]
# Add AsmPrinter Cpu0AsmPrinter
required_libraries = AsmPrinter CodeGen Core MC Cpu0AsmPrinter Cpu0Desc Cpu0Info
```

Now, run 3/3/Cpu0 for AsmPrinter support, we get error message as follows,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
/usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc: target does not support generation of this
```

The llc fails to compile IR code into machine code since we didn't implement class Cpu0DAGToDAGISel. Before the implementation, I will introduce the LLVM Code Generation Sequence, DAG, and LLVM instruction selection in next 3 sections.

4.4 LLVM Code Generation Sequence

Following diagram came from tricore_llvm.pdf.

LLVM is a Static Single Assignment (SSA) based representation. LLVM provides an infinite virtual registers which can hold values of primitive type (integral, floating point, or pointer values). So, every operand can save in different virtual register in llvm SSA representation. Comment is ";" in llvm representation. Following is the llvm SSA instructions.

```
store i32 0, i32* %a  ; store i32 type of 0 to virtual register %a, %a is pointer type which point to
store i32 %b, i32* %c ; store %b contents to %c point to, %b isi32 type virtual register, %c is point
%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
```

I explain the code generation process as below. If you don't feel comfortable, please check tricore_llvm.pdf section 4.2 first. You can read "The LLVM Target-Independent Code Generator" (http://llvm.org/docs/CodeGenerator.html) and "LLVM Language Reference Manual" (http://llvm.org/docs/LangRef.html) before go ahead, but I think read section 4.2 of tricore_llvm.pdf is enough. I suggest you read the web site documents as above only when you are still not quite understand, even though you have read this section and next 2 sections article for DAG and Instruction Selection.

1. Instruction Selection

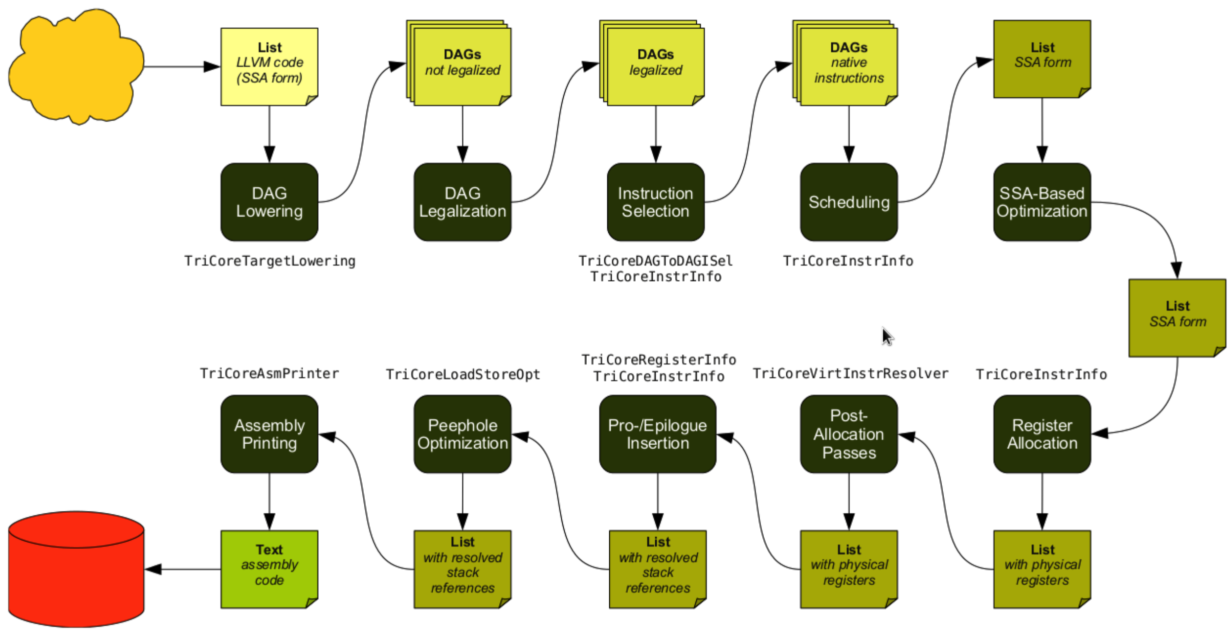


Figure 4.5: Fig 3.5 of tricore_llvm.pdf: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

```
// In this stage, transfer the llvm opcode into machine opcode, but the operand still is llvm virtua
      store i16 0, i16* %a // store 0 of i16 type to where virtual register %a point to
=>  addiu i16 0, i32* %a
```

2. Scheduling and Formation

```
// In this stage, reorder the instructions sequence for optimization in instructions cycle or in reg
      st i32 %a, i16* %b,  i16 5 // st %a to *(%b+5)
      st %b, i32* %c, i16 0
      %d = ld i32* %c

// Transfer above instructions order as follows. In RISC like Mips the ld %c use the previous instru
// cycles. Meaning the ld cannot follow st immediately.
=>  st %b, i32* %c, i16 0
      st i32 %a, i16* %b,  i16 5
      %d = ld i32* %c, i16 0
// If without reorder instructions, a instruction nop which do nothing must be filled, contribute one
// optimization. (Actually, Mips is scheduled with hardware dynamically and will insert nop between s
// didn't insert nop.)
      st i32 %a, i16* %b,  i16 5
      st %b, i32* %c, i16 0
      nop
      %d = ld i32* %c, i16 0

// Minimum register pressure
// Suppose %c is alive after the instructions basic block (meaning %c will be used after the basic b
// %b are not alive after that.
// The following no reorder version need 3 registers at least
      %a = add i32 1, i32 0
      %b = add i32 2, i32 0
      st %a,  i32* %c, 1
```

```
      st %b,  i32* %c, 2

// The reorder version need 2 registers only (by allocate %a and %b in the same register)
=> %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 expression remove, shown in next section DAG.

4. Register Allocation

   Allocate real register for virtual register.

5. Prologue/Epilogue Code Insertion

   Explain in section Add Prologue/Epilogue functions

6. Late Machine Code Optimizations

   Any "last-minute" peephole optimizations of the final machine code can be applied during this phase. For example, replace x = x * 2 by x = x < 1 for integer operand.

7. **Code Emission** Finally, the completed machine code is emitted. For static compilation, the end result is an assembly code file; for JIT compilation, the opcodes of the machine instructions are written into memory.

4.5 DAG (Directed Acyclic Graph)

Many important techniques for local optimization begin by transforming a basic block into DAG. For example, the basic block code and it's corresponding DAG as Fig 3.6.

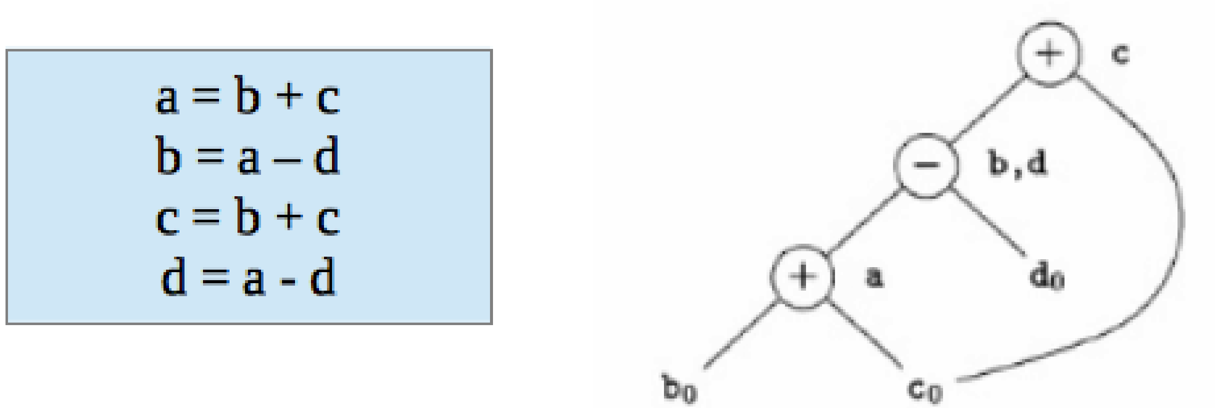


Figure 4.6: Fig 3.6 DAG example

If b is not live on exit from the block, then we can do common expression remove to get the following code.

```
a = b + c
d = a - d
c = d + c
```

As you can imagine, the common expression remove can apply in IR or machine code.

DAG like a tree which opcode is the node and operand (register and const/immediate/offset) is leaf. It can also be represented by list as prefix order in tree. For example, (+ b, c), (+ b, 1) is IR DAG representation.

4.6 Instruction Selection

In back end, we need to translate IR code into machine code at Instruction Selection Process as Fig 3.7.

MOV	$r_d = r_s$	ADDI	$r_d = r_s + 0$
MOV	$r_d = r_s$	ADD	$r_d = r_{s1} + r_0$
MOVI	$r_d = c$	ADDI	$r_d = r_0 + c$

Figure 4.7: Fig 3.7 IR and it's corresponding machine instruction

For machine instruction selection, the better solution is represent IR and machine instruction by DAG. In Fig 3.7, we skip the register leaf. The rj + rk is IR DAG representation (for symbol notation, not llvm SSA form). ADD is machine instruction.

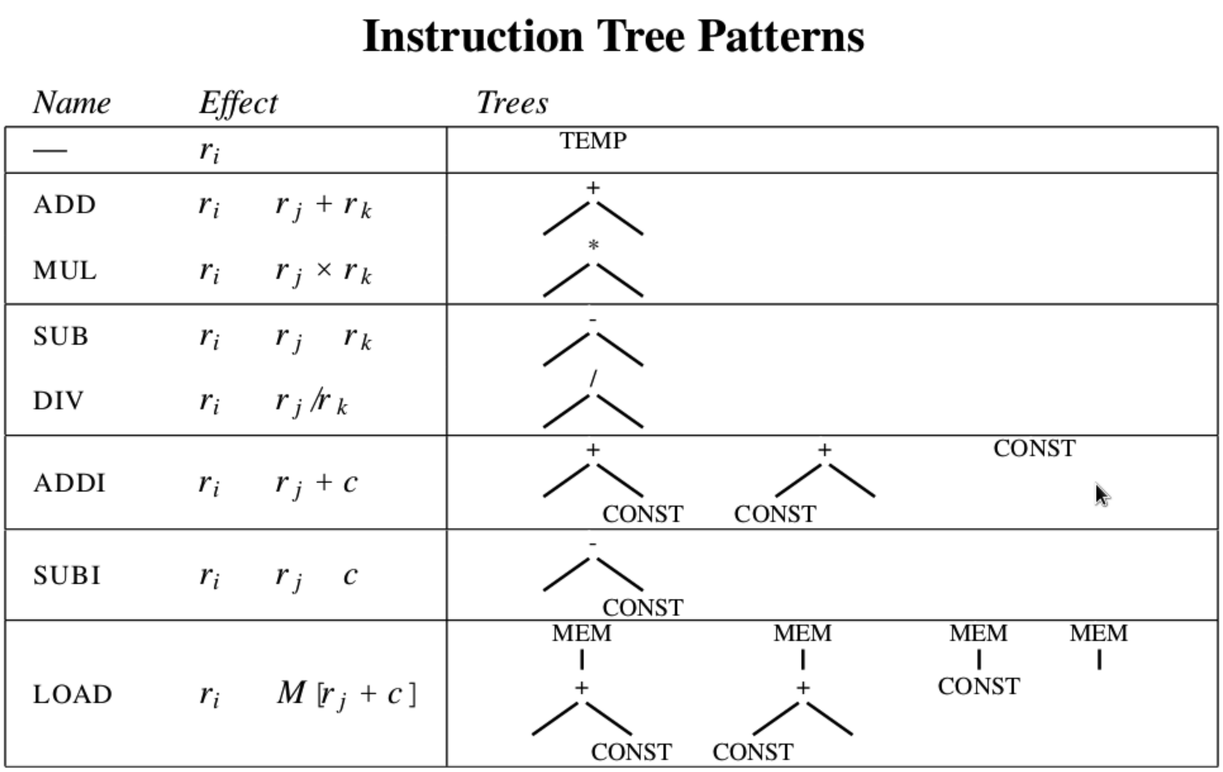


Figure 4.8: Fig 3.8 Instruction DAG representation

We can also represent IR DAG and machine instruction DAG as list. For example, (+ ri, rj), (- ri, 1) are lists for IR DAG; (ADD ri, rj), (SUBI ri, 1) are lists for machine instruction DAG.

Now, let's recall the ADDiu instruction defined on Cpu0InstrInfo.td in chapter 2. And It will expand to the following Pattern as mentioned in section Write td (Target Description) of chapter 2 as follows,

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

Pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
```

This pattern meaning the IR DAG node **add** can translate into machine instruction DAG node ADDiu by pattern match mechanism. Similarly, the machine instruction DAG node LW and ST can be got from IR DAG node **load** and **store**.

Some cpu/fpu (floating point processor) has multiply-and-add floating point instruction, fmadd. It can be represented by DAG list (fadd (fmul ra, rc), rb). For this implementation, we can assign fmadd DAG pattern to instruction td as follows,

```
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 with ADDiu, [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC), F4RC:$FRB))] is the pattern which include node **fmul** and node **fadd**.

Now, for 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 llvm SelectionDAG Optimization Phase (is part of Instruction Selection Process) prefered to translate this 2 IR DAG node (fmul %a, %b) (fadd %d, %c) into one machine instruction DAG node (**fmadd** %a, %c, %b), than translate them into 2 machine instruction nodes **fmul** and **fadd**.

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

As you can see, the IR notation representation is easier to read then llvm SSA IR form. So, we use the notation form in this book sometimes.

For 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 3.7 Instruction tree pattern to get 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;
```

4.7 Add Cpu0DAGToDAGISel class

We have introduced the IR DAG to machine instruction DAG transformation in the previous section. Now, let's check what IR DAG node the file ch3.bc has. 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 use the IR DAG node **store**, **ret**. Actually, it also use **add** for sp (stack point) register adjust. So, the definitions in Cpu0InstInfo.td as follows is enough. IR DAG is defined in file include/llvm/Target/TargetSelectionDAG.td.

```
/// Load and Store Instructions
///  aligned
defm LW      : LoadM32<0x00,  "lw",  load_a>;
defm ST      : StoreM32<0x01, "st",  store_a>;

/// Arithmetic Instructions (ALU Immediate)
//def LDI     : MoveImm<0x08, "ldi", add, simm16, immSExt16, CPURegs>;
// 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, isCodeGenOnly=1,
    isBarrier=1, hasCtrlDep=1 in
  def RET : FJ <0x2C, (outs), (ins CPURegs:$target),
                "ret\t$target", [(Cpu0Ret CPURegs:$target)], IIBranch>;
```

Add class Cpu0DAGToDAGISel (Cpu0ISelDAGToDAG.cpp) to CMakeLists.txt, and add following fragment to Cpu0TargetMachine.cpp,

```
//  Cpu0TargetMachine.cpp
...
// Install an instruction selector pass using
// the ISelDag to gen Cpu0 code.
bool Cpu0PassConfig::addInstSelector() {
  PM->add(createCpu0ISelDag(getCpu0TargetMachine()));
  return false;
}

//  Cpu0ISelDAGToDAG.cpp
/// createCpu0ISelDag - This pass converts a legalized DAG into a
/// CPU0-specific DAG, ready for instruction scheduling.
FunctionPass *llvm::createCpu0ISelDag(Cpu0TargetMachine &TM) {
  return new Cpu0DAGToDAGISel(TM);
}
```

In this version, we add the following code in Cpu0InstInfo.cpp to enable debug information which called by llvm at proper time.

```
// Cpu0InstInfo.cpp
...
MachineInstr*
Cpu0InstrInfo::emitFrameIndexDebugValue(MachineFunction &MF, int FrameIx,
                                        uint64_t Offset, const MDNode *MDPtr,
                                        DebugLoc DL) const {
```

```
  MachineInstrBuilder MIB = BuildMI(MF, DL, get(Cpu0::DBG_VALUE))
    .addFrameIndex(FrameIx).addImm(0).addImm(Offset).addMetadata(MDPtr);
  return &*MIB;
}
```

Build 3/4, run it, we find the error message in 3/3 is gone. The new error message for 3/4 as follows,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
Target didn't implement TargetInstrInfo::storeRegToStackSlot!
UNREACHABLE executed at /usr/local/llvm/3.1.test/cpu0/1/src/include/llvm/Target/TargetInstrInfo.h:39
Stack dump:
0.       Program arguments: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
1.       Running pass 'Function Pass Manager' on module 'ch3.bc'.
2.       Running pass 'Prologue/Epilogue Insertion & Frame Finalization' on function '@main'
Aborted (core dumped)
```

4.8 Add Prologue/Epilogue functions

Following came 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. Figure 3.9 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.

I will explain the Prologue and Epilogue further by example code. So for the following llvm IR code, Cpu0 back end will emit the corresponding machine instructions as follows,

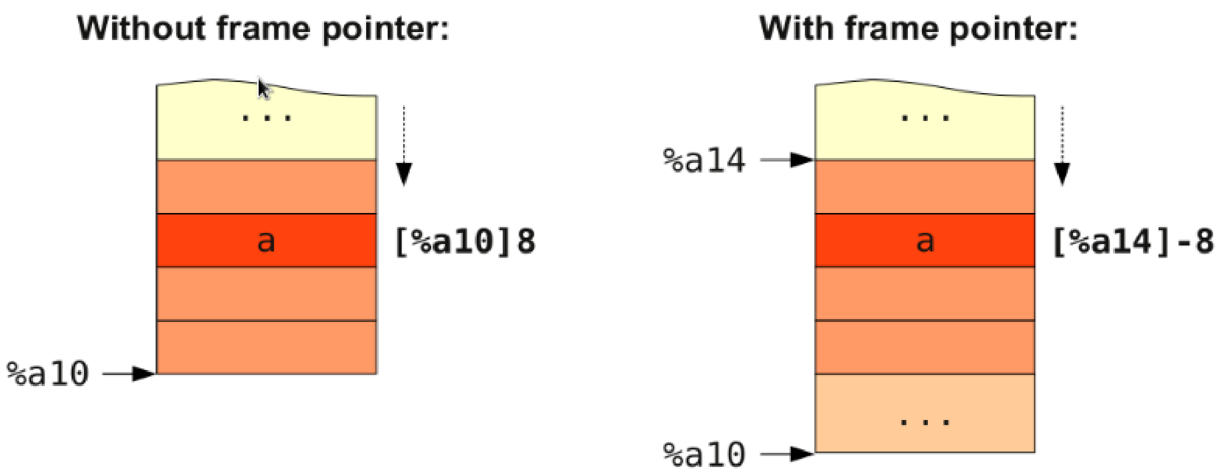


Figure 4.9: Fig 3.9 Addressing of a variable a located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

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

        .section .mdebug.abi32
        .previous
        .file        "ch3.bc"
        .text
        .globl        main
        .align        2
        .type        main,@function
        .ent        main                    # @main
main:
        .frame        $sp,8,$lr
        .mask         0x00000000,0
        .set        noreorder
        .set        nomacro
# BB#0:                                # %entry
        addiu        $sp, $sp, -8
        addiu        $2, $zero, 0
        st        $2, 4($sp)
        addiu        $sp, $sp, 8
        ret        $lr
        .set        macro
        .set        reorder
        .end        main
$tmp1:
        .size        main, ($tmp1)-main
```

LLVM get the stack size by parsing IR and counting how many virtual registers is assigned to local variables. After that, it call emitPrologue(). This function will emit machine instructions to adjust sp (stack pointer register) for local variables since we don't use fp (frame pointer register). For our example, it will emit the instructions,

```
addiu       $sp, $sp, -8
```

The emitEpilogue will emit "addiu $sp, $sp, 8", 8 is the stack size.

Since Instruction Selection and Register Allocation occurs before Prologue/Epilogue Code Insertion, eliminateFrameIndex() is called after machine instruction and real register allocated. It translate the frame index of local variable (%1 and %2 in the following example) into stack offset according the frame index order upward (stack grow up downward from high address to low address, 0($sp) is the top, 52($sp) is the bottom) as follows,

```
define i32 @main() nounwind uwtable {
       %1 = alloca i32, align 4
       %2 = alloca i32, align 4
      ...
      store i32 0, i32* %1
      store i32 5, i32* %2, align 4
      ...
      ret i32 0

=> # BB#0:
        addiu       $sp, $sp, -56
$tmp1:
        addiu       $3, $zero, 0
        st       $3, 52($sp)   // %1 is the first frame index local variable, so allocate in 52($sp)
        addiu       $2, $zero, 5
        st       $2, 48($sp)    // %2 is the second frame index local variable, so allocate in 48($s
                ...
                ret       $lr
```

After add these Prologue and Epilogue functions, and build with 3/5/Cpu0. Now we are ready to compile our example code ch3.bc into cpu0 assembly code. Following is the command and output file ch3.cpu0.s,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc -march=cpu0 -
[Gamma@localhost InputFiles]$ cat ch3.cpu0.s
        .section .mdebug.abi32
        .previous
        .file       "ch3.bc"
        .text
        .globl       main
        .align       2
        .type       main,@function
        .ent       main                    # @main
main:
        .frame       $sp,8,$lr
        .mask       0x00000000,0
        .set       noreorder
        .set       nomacro
# BB#0:                                    # %entry
        addiu       $sp, $sp, -8
        addiu       $2, $zero, 0
        st       $2, 4($sp)
        addiu       $sp, $sp, 8
        ret       $lr
        .set       macro
        .set       reorder
        .end       main
$tmp1:
        .size       main, ($tmp1)-main
```

4.9 Summary of Chapter 3

We have finished a simple assembler for cpu0 which only support **addiu**, **st** and **ret** 3 instructions.

I am satisfied with this result. But you may think "After so many codes we program, and just get the 3 instructions". The point is we have created a frame work for cpu0 target machine (please look back the llvm back end structure class inherit tree early in this chapter). Until now, we have 3000 lines of source code with comments which include files *.cpp, *.h, *.td, CMakeLists.txt and LLVMBuild.txt. LLVM front end tutorial have 700 lines of source code without comments totally. Don't feel down with this result. In reality, write a back end is warm up slowly but run fast. Clang has over 500,000 lines of source code with comments in clang/lib directory which include C++ and Obj C support. Mips back end has only 15,000 lines with comments. Even the complicate X86 CPU which CISC outside and RISC inside (micro instruction), has only 45,000 lines with comments. In next chapter, I will show you that add a new instruction support is as easy as 123.

CHAPTER

FIVE

TODO LIST

Todo

Add info about LLVM documentation licensing.

(The *original entry* is located in /Users/ajamshidi/Documents/lbd/source/about.rst, line 38.)

CHAPTER

SIX

ALTERNATE FORMATS

The book is also available in: