UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Corso di Laurea in Informatica

A MicroBlaze Backend for the LLVM Compiler

Relatore:

Prof.
Andrea Marongiu

Candidato:

Matr. 136174
Lorenzo
Stigliano

Anno Accademico 2022-2023

Contents

1 Introduction

Modern embedded systems have evolved into powerful processors, capable of delivering very high performance while at the same time consuming little power, which was always their trademark. This has made possible the realization of sophisticated applications in several contexts like autonomous cars and drones, smart cities, robotic surgery and so on. The main change in the way the architecture of this new type of computers is designed involves massive use of parallel and heterogeneous compute resources, which has led to the era of heterogeneous systems on a chip (HeSoC). A modern HeSoC combines a main *host* processor (a standard multi-core CPU) with one or more *accelerators*, so as to combine the benefits of general-purpose computing with those of domain-specific, efficient processing capabilities. Among the many types of accelerators that HeSoCs can exploit there are two that are largely used: (i) general-purpose graphics processing units (GP-GPUs) and (ii) Field Programmable Gate Arrays (FPGA).

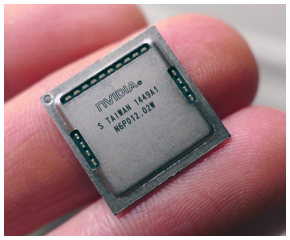


Figure 1: NVIDIA Tegra X1 GPGPU

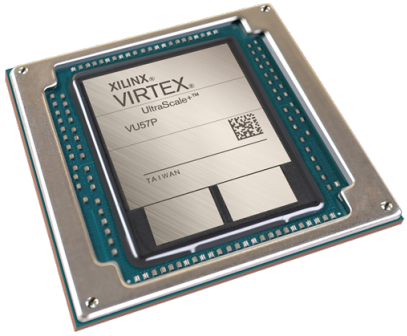


Figure 2: Virtex Ultrascale+ FPGA

While HeSoCs are nowadays largely exploited in commercial products, a lot of research is still being done to study how to improve their characteristics, both at the hardware and software level. This raises the need for efficient research tools aimed at enabling fast and accurate hardware and software development, a typical example being system simulators and emulators. The HERO plat-

form [19] [38] is a research platform aimed at emulating a HeSoC that combines a general-purpose ARM Cortex-A multicore processor with a programmable manycore accelerators (PMCA) composed of *clusters* of RISC-V cores. For its implementation HERO leverages an FPGA-based [36] HeSoC (e.g., a Xilinx Ultrascale+ chip), where the *host* processor is physically available on the SoC and the PMCA is implemented as soft cores on the FPGA fabric. The emulation that can be achieved on FPGA vastly surpasses any type of architecture simulation in terms of speed and accuracy. Over the years, HERO has also been extended to enrich its compute *clusters* with custom accelerators besides RISC-V cores [18], which enables its use as a real application accelerator, beyond simple emulation capabilities.

With the increasing complexity of the system, methodologies to simplify the way applications are written is mandatory. HERO includes a complete software stack that consists of a heterogeneous compilation toolchain with support for OpenMP accelerator programming [17], a Linux driver and runtime libraries for both the host and the PMCA. Since specifications v4.0, OpenMP provides an ideal programming model for this type of hardware, easily unlocking all the opportunities for parallels computing in applications, achieving high performance while still maintaining low coding complexity. Supporting a full-fledged OpenMP toolchain and runtime system on HERO also makes it an ideal platform for doing research in the field of programming model and compiler support for HeSoCs.

Figure 3: OpenMP logo

HERO, as part of the PULP project, is developed using low-level hardware description languages, which makes it difficult for non experts to explore variants of its architecture [37]. For this reason, a version of the HERO clusters based on Xilinx IPs (soft cores, interconnect, memory) has been developed in the past few years. This allows to rely on the much easier-to-use development tools offered by Xilinx to implement extensions or variants to the base architecture (see Figure 31).

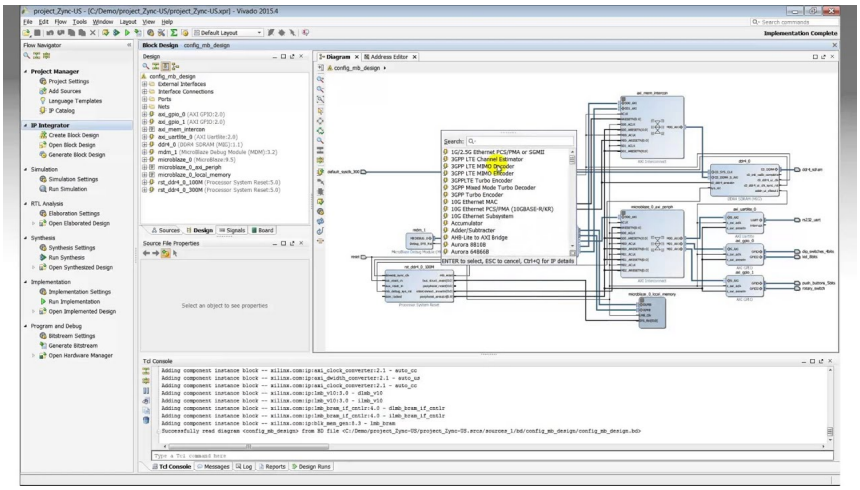


Figure 4: Xilinx Vivado Software for Hardware Design

This thesis places its roots in this very version of the platform, which combines the mature software stack of the original HERO infrastructure with the ease of use of Xilinx's tools for the architecture space exploration. This variant of the platform relies on the MicroBlaze soft core (owned by Xilinx) on FPGA, instead of the RISC-V ISA. This poses a new challenge: the software stack for the original HERO platform, with support for OpenMP programming, is based on the LLVM toolchain. LLVM, however, does not support compilation for the MicroBlaze architecture: in the past (LLVM 3.3 release) this target was supported, but it was discontinued and Xilinx started to rely only on the GCC compiler to support its architecture. As a consequence, it is not straightforward to support OpenMP compilation for the new platform, as GCC compilation cannot be readily be integrated in the LLVM infrastructure (nor could this type of integration provide optimal results in terms of code optimization opportunities).

This thesis proposes an implementation of MicroBlaze as a new backend inside the LLVM project, enabling the use of all the code that was written until now for the HERO platform (that largely relies on OpenMP API).

1.1 Thesis Structure

The thesis starts from the description of MicroBlaze 2 architecture in order to give the reader an idea about what is MicroBlaze in term of structure and ISA. In this chapter are explained the main type of instructions 2.2, registers 2.3 and memory architecture 2.5. The information are taken from the official Xilinx's documentation [1].

In the third chapter 3 is explained how LLVM works and how it is structured. In the beginning there is an overview about its main components 3.1 and then are explained how the frontend 3.3 and the backend 3.4 part behave.

In the fourth chapters is given an overview about the implementation of a custom backend inside LLVM 4, while in the in the fifth chapter there is a more detailed explanation its implementation 5. In this section are present most of the code snippets and details about the working of the code, in relation with what was declared about MicroBlaze ISA.

In the sixth chapter there is the evaluation of the project6, in which are described the instructions/registers and functionalities that have been implemented along with the main benchmark that was used to test the backend 6.2.

2 MBLAZE Architecture

2.1 Introduction

The MicroBlaze embedded processor soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx Field Programmable Gate Arrays (FPGAs). The processor is based on the Harvard Architecture; this is a computer architecture with separate bus for: data and instruction accesses. This means that instruction fetches and data accesses do not contend a single memory pathway, so the CPU can read an instruction and perform a data memory access at the same time.

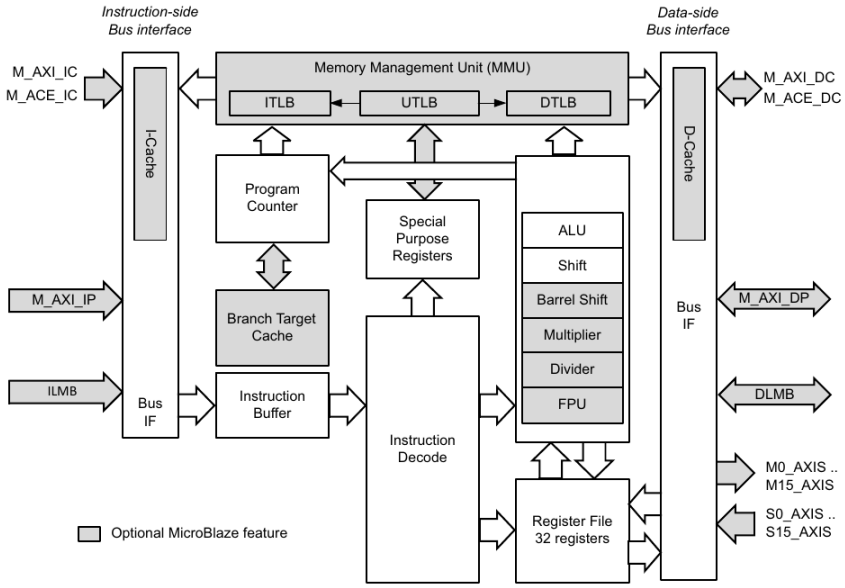
The processor has a versatile interconnect system to support a variety of embedded applications: primary I/O bus, the AXI interconnect (more information about AXI) , is a system-memory mapped transaction bus with master–slave capability. Older versions of the MicroBlaze used the CoreConnect PLB bus (the goal of the PLB is to provide a standard interface between the processor cores and integrated bus controllers). For access to local-memory (FPGA RAM), MicroBlaze uses a dedicated LMB bus, which provides fast on-chip storage. User-defined co-processors are supported through dedicated AXI4-Stream connections. The co-processor(s) interface can accelerate computationally intensive algorithms by offloading parts or the entirety of the computation to a user-designed hardware module.

Many aspects of the MicroBlaze can be user configured: cache size, pipeline depth (3-stage, 5-stage, or 8-stage), embedded peripherals, memory management unit, and bus-interfaces can be customized. The area-optimized version of MicroBlaze, which uses a 3-stage pipeline, sacrifices clock frequency for reduced logic area. The performance-optimized version expands the execution pipeline to 5 stages, allowing top speeds of more than 700 MHz (on Virtex UltraScale+ FPGA family). Also, key processor instructions which are rarely used but more expensive to implement in hardware can be selectively added/removed (e.g. multiply, divide, and floating point operations). This customization enables a developer to make the appropriate design trade-offs for a specific set of host hardware and application software requirements. The only fixed features are:

- Thirty-two 32-bit or 64-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- Default 32-bit address bus, extensible to 64 bits
- Single issue pipeline

With the memory management unit, MicroBlaze is capable of hosting operating systems requiring hardware-based paging and protection, such as the Linux kernel. Otherwise it is limited to operating systems with a simplified protection and virtual memory model, like FreeRTOS or Linux without MMU

support. MicroBlaze's overall throughput is substantially less than a comparable hard CPU core (such as the ARM Cortex-A9 in the Zynq). The MicroBlaze processor can use Big-Endian or Little-Endian format to represent data (depending on the selected endianess); by default the parameter `C_ENDIANESS` is set to 1 (little endian enabled). For 32-bit MicroBlaze he has hardware supported data types like: word, half word and byte. With 64-bit MicroBlaze also long and double are available in hardware.



2.2 Instructions

MicroBlaze processor has 32 bits instructions that can be defined in two formats:

1. Type A: up to two source register operands and one destinations register operand;
2. Type B: one source register and a 16-bit immediate operand;

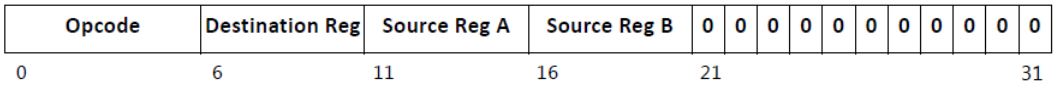


Figure 5: Type A instruction format

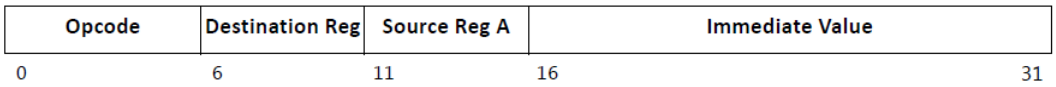


Figure 6: Type B instruction format

Instructions are provided in this functional categories:

1. arithmetic (add, sub, ...)
2. logical (and, or, ...)
3. branch (beq, beqi, ...)
4. load/store (lw, swi, ...)
5. special

From these categories is possible to generate all the instructions like: *add*, *sub*, branch conditional instructions, etc. ... and in the documentation are represented as follows:

add **Arithmetic Add**

add	rD, rA, rB	Add
addc	rD, rA, rB	Add with Carry
addk	rD, rA, rB	Add and Keep Carry
addkc	rD, rA, rB	Add with Carry and Keep Carry

0 0 0 K C 0	rD	rA	rB	0 0 0 0 0 0 0 0 0 0 0
0	6	11	16	21 31

Figure 7: ADD instruction definitions

There is the indication of the specifi subgroup, in this example *Arithmetic* and then there is the definition of the instruction itself. All the instructions are listed in the *MicroBlaze 32-bit Instructions* and *MicroBlaze 64-bit Instructions* section of the documentation[1].

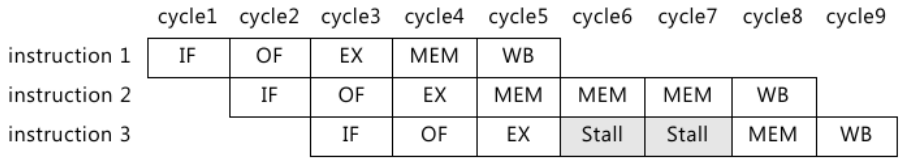
2.2.1 Pipeline

MicroBlaze instruction execution is pipelined and for most instructions each stage of the pipeline takes one clock cycle to complete. So, in order to fully complete a instruction, are needed as many clock cycles as the number of pipeline stages. In absence of data/control/structural hazards it completes one instruction per cycle, but this is not a best practice at all. Thanks to `C_AREA_OPTIMISED` is possible to set three different types of pipeline:

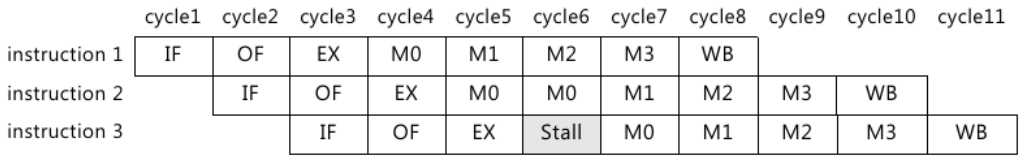
1. Three Stage Pipeline: `C_AREA_OPTIMISED` is set to 1 and the pipeline is divided into 3 stages: Fetch, Decode and Execute

	cycle1	cycle2	cycle3	cycle4	cycle5	cycle6	cycle7
instruction 1	Fetch	Decode	Execute				
instruction 2		Fetch	Decode	Execute	Execute	Execute	
instruction 3			Fetch	Decode	Stall	Stall	Execute

2. Five Stage Pipeline: `C_AREA_OPTIMISED` is set to 0, so the pipeline is divided into 5 stages: Fetch, Decode, Execute, Access Memory, Writeback



3. Eight Stage Pipeline: `C_AREA_OPTIMISED` is set to 2 and the pipeline is divided in 8 stages: Fetch, Decode, Execute, Access Memory 0, Access Memory 1, Access Memory 2, Access Memory 3, Writeback



2.2.2 Branches

Typically the instructions that are in the fetch/decode stage are flushed when executing a taken branch and the fetch stage is reloaded with a new instruction (taken from the calculated branch address). A taken branch takes 3 clock cycles to be executed, 2 of which are just refilling the pipeline, so in order to reduce this latency overhead, the processor supports branches with:

- Delay Slots: thanks to that MicroBlaze flushes only fetch pipeline stage and the instruction in decode can be completed. This reduces branch penalty from 2 clock cycles to 1.
- Optional Branch Target Cache: this technique is coupled with a branch prediction scheme can help not producing more overhead for correctly predicted immediate branch or return instruction. Basically BTC saves the target address of each immediate branch/return instruction the first time is encountered. The next time it is encountered it can be found in the BTC, so the Instruction Fetch Program Counter is then simply changed to the saved target address (in case the branch should be taken).

2.3 Registers

MicroBlaze can have 32 32-bit or 64-bit general purpose registers and up to 16 special purpose registers.

2.3.1 General Purpose Register

In 32-bit version registers are numbered from **R0** to **R31** and this is the standard use of them:

1. **R0**: anything written to r0 is discarded
2. **R1 to R13** to R13: general purpose register
3. **R14**: register sued to store return addresses for interrupts
4. **R15**: recommended for storing return addresses for user vectors
5. **R16**: used to store return addresses for breaks
6. **R17**: if MicroBlaze is configured to support hardware exceptions, this register is loaded with the address of the instruction following the instruction causing hardware exception; if not, it is a general purpose register
7. **R18** to **R31**: general purpose register

2.4 Special Purpose Registers

- **Program Counter (PC)**: is the address of the executions instruction.
- **Machine Status Register (MSR)**: the Machine Status Register contains control and status bits for the processor.
- **Exception Address Register (EAR)**: it stores value from load/store instructions that caused exception for the following cases:
  - unaligned access exception that specifies the unaligned access data address
  - `M_AXI_DP` exception that specifies the failing AXI4 data access address
  - data storage exception that specifies the (virtual) effective address accessed
  - instruction storage exception that specifies the (virtual) effective address read
  - data TLB miss exception that specifies the (virtual) effective address accessed
  - instruction TLB miss exception that specifies the (virtual) effective address read

- **Exception Status Register (ESR)**: it stores value for the processor, in the same way of EAR.
- **Branch Target Register (BTR)**: it works if MBLAZE is configured to handle exceptions. It stores the branch target address for all delay slot branch instructions that are executed while MSR[EIP] is set to zero. If an exception is caused by an instruction in a delay slot, the exception handler should return exectuion to the address stored in BTR instead of the exception return address that is typically stored in R17.
- **Floating Point Status Register (FSR)**: contains satus bits for the floating point unit. It can be read with an MFS, and writte with a MTS instruction.
- **Exception Data Register (EDR)**: stores data read on an AXI4-Stream link that have caused a stream exception. The AXI4-Stream protocol is used as a standard interface to connect components that want to exchange data. It can be used to connect a single master, that generates data, to a single slave, that receives data. The protocol can also be used when connecting larger numbers of master and slave components and supports multiple data streams using the same set of shared wires, allowing a generic interconnect to be constructed that can perform upsizing, downsizing and routing operations. The AXI4-Stream interface also supports a wide variety of different stream types and the protocol defines the association between Transfers and Packets.
- **Stack Low Register (SLR)**: SLR stores the stack low limit use to detect stack overflow. This is the situation: if an address of load/store instructions is using the stack pointer as rA and it is less then the Stack Low Register, a stack overflow occurs, causing Stack Protection Violation exception (if exceptions are enabled in MSR).
- **Stack High Register (SHR)**: stores the stack high limit use to detect stack underflow. When the address of a load/store instructions that is using the stack pointer rA is greater than the SHR, a stack underflow occurs, causing a Stack Protection Violation exception if exceptions are enabled in MSR.
- **Process Identifier Register (PID)**: is used to uniquely identify a software process during MMU address translation; it is controlled by `C_USE_MMU` configuration option on MBLAZE.
- **Zone Protection Register (ZPR)**: is used to override MMU memory protection that is defined in the TLB entries. It is possibile to configure it by `C_USE_MMU` option, in combination with the following options:
  - `C_USE_MMU` is greater than 1 (User Mode)
  - `C_AREA_OPTIMIZED` is set to 0 (Performance) or 2 (Frequency)

– the number of specified memory protection zones is greater than zero (C_MMU_ZONES> 0)

2.5 Memory Architecture and Instructions

MicroBlaze is implemented with a Harvard memory architecture; instruction and data accesses are done in separate address spaces. The instruction address space has a 32-bit virtual address range with 32-bit MicroBlaze (with 64-bit MicroBlaze address space can be extended to 64-bit) and can be extended up to a 64-bit physical address range when using the MMU in virtual mode. The bit configuration of the address space remains for the data address space.

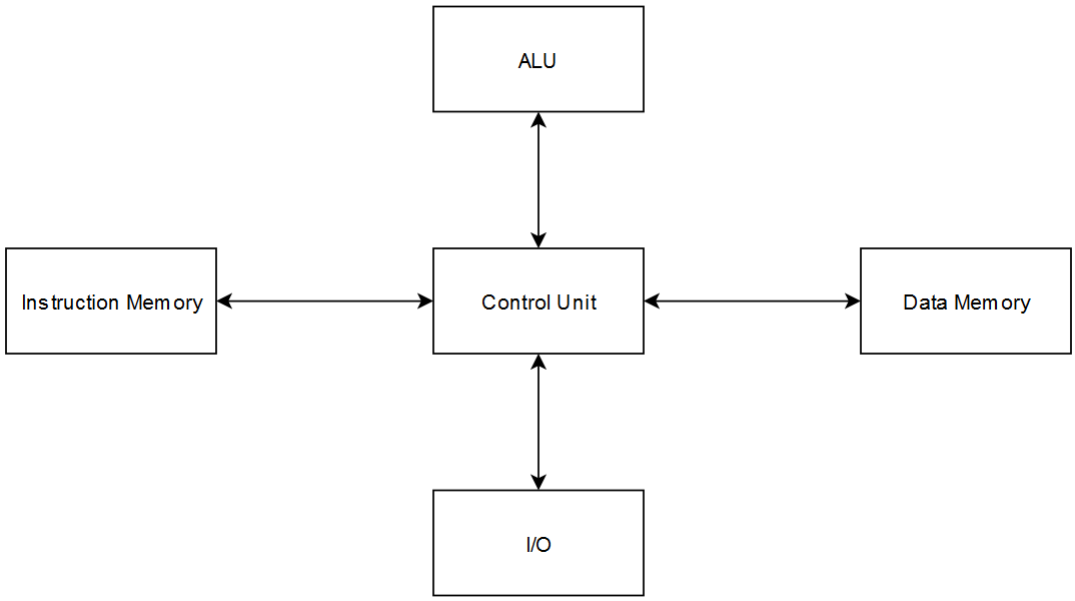


Figure 8: Harvard Architecture

MicroBlaze supports memory operation in three data sizes 8 bits (Byte), 16 bits (Halfword) and 32 bits (Word). The following schematics are an example for the load instruction:

lbu — Load Byte Unsigned

lbu	rD_x, rA_x, rB_x
lbur	rD_x, rA_x, rB_x
lbuea	rD, rA, rB

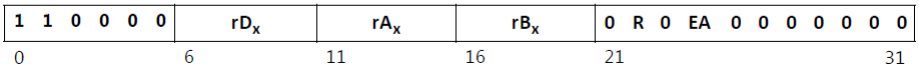


Figure 9: Load Byte Unsigned

lhu — Load Halfword Unsigned

lhu	rD_x, rA_x, rB_x
lhur	rD_x, rA_x, rB_x
lhuea	rD, rA, rB

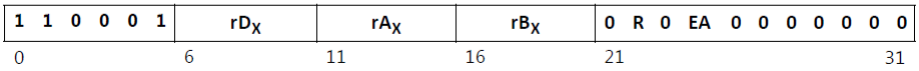


Figure 10: Load Half Word Unsigned

lw — Load Word

lw	rD_x, rA_x, rB_x
lwr	rD_x, rA_x, rB_x
lwea	rD, rA, rB

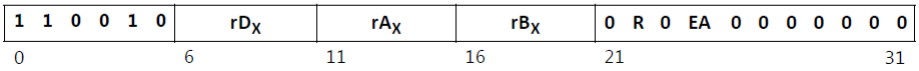


Figure 11: Load Word

3 LLVM Structure

LLVM is a compiler toolchains namely a set of tools, organises as a library, that work together to achieve compilation. The project started in 2000 at the University of Illinois at Urbana–Champaign, under the direction of Vikram Adve (professor in the Department of Computer Science, Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign) and the research assistant and M.Sc. student Chris Lattner. LLVM was originally developed as a research infrastructure to investigate dynamic compilation techniques for static and dynamic programming languages. LLVM was released under the University of Illinois/NCSA Open Source License. In 2005, Apple Inc. hired Lattner and formed a team to work on the LLVM system for various uses within Apple's development systems. LLVM has been an integral part of Apple's Xcode development tools for macOS and iOS since Xcode 4. Over time, Lattner built out the technology, personally implementing many major new features in LLVM, formed and built a team of LLVM developers at Apple, started the Clang project, took responsibility for evolving Objective-C. In 2006, Lattner started working on a new project called Clang. The combination of Clang front-end and LLVM back-end is called Clang/LLVM or simply Clang. The name LLVM was originally an initialism for Low Level Virtual Machine. However, the LLVM project evolved into an umbrella project that has little relationship to what most current developers think of as a virtual machine. As such the initialism was "confusing" and "inappropriate", and as of 2011 LLVM was "officially no longer an acronym". Since 2011, LLVM is a brand that applies to the LLVM umbrella project, encompassing the LLVM intermediate representation (IR), the LLVM debugger, the LLVM implementation of the C++ Standard Library (with full support of C++11 and C++14), etc. LLVM is administered by the LLVM Foundation. Its president is compiler engineer Tanya Lattner.

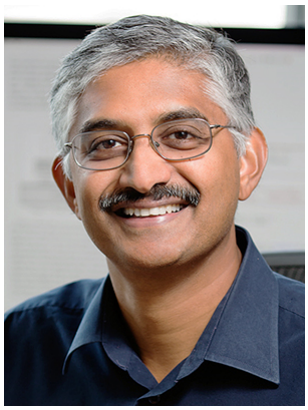


Figure 12: Vikram Adve

Figure 13: Chris Lattner

The idea behind LLVM is that everything is a library, so that most of the code is highly reusable through all the tools and is more *easy* to understand. This allows the user to observe all the steps of the compilation and learn them gradually.

Through the compilation process is possible to represent 3 major phases:

1. translating from the source programming language to LLVM IR (llvm intermediate representation);
2. optimize the llvm-ir;
3. transform the LLVM IR to a machine-specific assembly language, at first the program is converted to a *Directed Acyclic Graph* (DAG), in order to select instruction easily. Then it is converted to a three address representation to allow instruction scheduling to happen (first: *SelectionDAG*, second: *MachineFunction*);
4. implement *assembler* and *linkers*;

What is LLVM Intermediate Representation? This is the bridge that connects frontends and backends. This allows LLVM to have a common language generated from all the frontends. The IR is the point where the majority of target-independent optimization takes place, before being taken as input from backends.

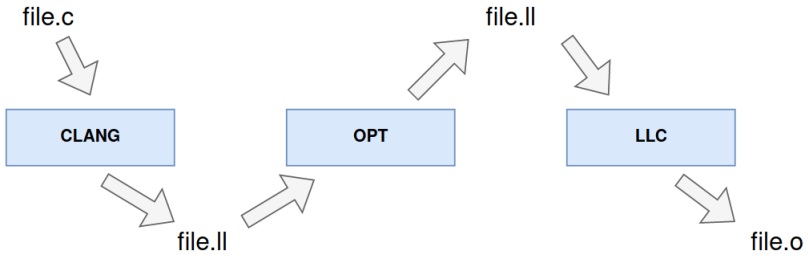


Figure 14: LLVM Toolchain

3.1 LLVM Component

In the project can be identified three major parts:

- **Frontend:** compiler step that translate programming language into Intermediate Representation (LLVM IR). This step includes: *lexical analyzer*, *syntax parser*, *semantyc analyzer*, *LLVM IR code generator*.
- **IR (Intermediate Representation):** LLVM IR is given in 2 formats: human readable and binary encoded representation. Tools and libraries provide an interfaces to IR construction, assembling and disassembling. There is an LLVM optimizer that operates IR; this is where most part of optimizations is applied.
- **Backend:** it is responsible for code generation and converts LLVM IR to target specific assembly code (or object binaries).

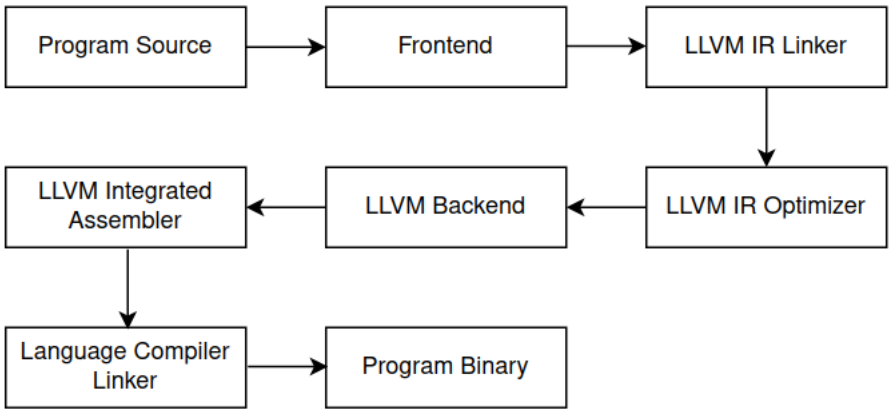


Figure 15: LLVM Workflow

The interaction between all this parts can happen in two different ways:

- **In memory:** this happens via a single supervisor tool (*Clang*, for example) that can use each *LLVM* component as a library and depends on data structures allocated in the memory to feed the output stage to the input of another (think about *pipe* in Bash scripting: `ls | grep Desktop`).
- **Through files:** this happens via a user who launches smaller standalone tools that write the result on file, and then the user can launch another tool that use this file as the input of the following operation.

So, thanks to the project's modularity is possible to use tool as stand alone. This is a list of the most important tool that ca be invoked:

- **opt:** optimizer IR level coding.
  - Input: IR file
  - Output: optimized IR
- **llc:** converts *llvm bitcode* into *target machine assembly language* file or object file via specific backend.
- **llvm-mc:** is able to assemble instructions and generate object files for several object formats such as ELF.
- **lli:** implements both the interpreter and JIT compiler for the LLVM IR.
- **llvm-link:** linker together several llvm bit codes to produce a single llvm bitcode that encompasses all inputs
- **llvm-as:** this tool transforms human readable LLVM IR files, called llvm assemblies, into llvm bitcode
- **llvm-dis:** decodes llvm bit codes into llvm assemblies

What is JIT compiler? JIT stands for *Just In Time*, so the program can compile at execution time [34].

3.2 LLVM Libraries Organization

LLVM is divided into several libraries and this is the list of the most important:

- **libLLVMCore:** contains all the logic related to the *llvm-ir*.
- **libllvanalysis:** several *llvm-ir* analysis passes, such as *alias* analysis, *dependence* analysis, constant folding loop, info memory, dependence analysis and instruction simplify.
- **libllvmcodegen:** implements analyses and transformations of:
  - target independent code generation;
  - machine level (the lower level of the *llvm-ir*);
- **libllvmtarget:** provides access to the target specific logic that is reserved for the next library
- **libllvmTargetcodegen:** this is a `<Target>` target specific code generation information, transformation, and analysis passes, which compose the backend.
- **libclang:** implements frontend for c/c++ which is the default language for llvm code. It has many functionalities such: diagnostic reporting, *ast* (abstract syntax tree) traversing, code implementation, ...

3.3 Frontend

A Compiler frontend converts source code (like C/C++ coding languages) into the IR, before target specific representation. It is possible to take as an example Clang and begin to see how it is divided in three major parts:

1. frontend implemented by Clang libraries (*libclang*).
2. the compiler driver implemented in the `clang` command and the Clang driver library.
3. the actual compiler implemented by the `clang -cc1` command; it makes extensive use of LLVM libraries to implement some other parts that need to work correctly.

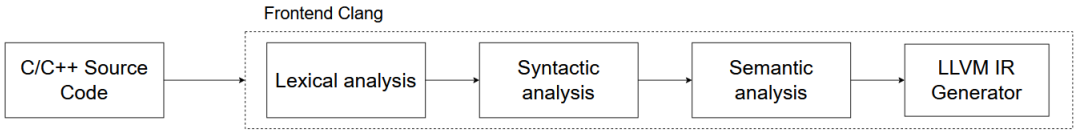


Figure 16: Clang workflow

3.3.1 Frontend's Phases: Lexical Analysis

Splitting the language constructs into **AST** of words and tokens, removing characters such as comments, white spaces, tabs,... each word/token must be part of the language subset and reserved language keywords are converted into internal compiler representations. The reserved words are defined in `llvm/clang/include/clang/Basic/TokenKinds.def`. If it encounters an invalid token it generates an error.

3.3.2 Frontend's Phases: Preprocessor

Acts before semantic analysis takes place and it is responsible for expanding *macros*, including files or skipping parts of the code thanks to comments (#). The preprocessor works in a tight dependence with *lexical analyzer (lexer)* and they interact with each other continuously. Is possible to see what happens to macros and preprocessor's activity with the following command: `pp-trace`.

3.3.3 Frontend's Phases: Syntactic Analysis

Syntactic Analysis starts immediately after the lexer and groups together *tokens* to form expression, statements and function bodies. It checks whether a group of tokens makes sense together, with respect of their physical layout, but the meaning of this code is not analyzed yet; in the same way a syntactic analysis of the English language is not worried about what the text says, but whether

the sentences is correct or not. This is what can expected from this phase in term of input/output:

- **Input:** tokens
- **Output:** abstract syntax tree

3.4 Backend Structure

The backend is the final step of the compilation and comprised a set of code generation, analysis and transform passes that convert the *LLVM IR* into object code. LLVM supports a lot of targets as: ARM, ARCH64, MIPS, SPARC, RISCV, X86, ... all these backends share a common interface, which is part of the target-dependent code generator, but at the same time each target must have a specialize code generator to implement target-specific behavior.

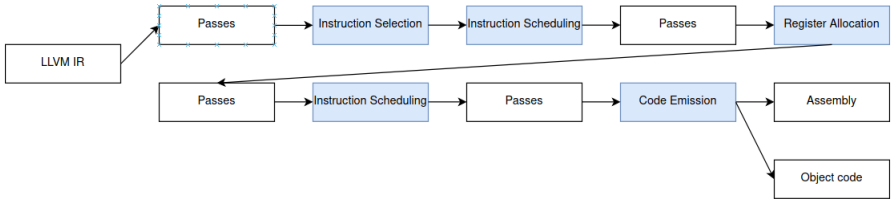


Figure 17: Backend workflow

There are different phases within the backend: the scheme before is the representation of the essential parts (from the IR to object code). The *light blue* box have more internal steps, but in this overview they will skipped and will be given a brief description:

- **Instruction Selection:** converts the in-memory IR representation into target-specific `SelectionDAG` nodes. In particular at first there is a conversion of the tree structure of the *LLVM IR* into a *DAG*. Each *DAG* is capable of representing the computation of a single basic block, which means that each basic block is associated with a different *DAG*. In the end, the *DAG* has all of its *LLVM IR* nodes converted to target-machine nodes (at this point every node represents a machine instruction rather then an llvm instruction).
- **Instruction Scheduling:** orders the instructions while trying to achieve instruction-level parallelism as much as possible. The instructions are then converted to the `MachineInstr` three-address representation.
- **Register Allocation:** transforms an infinite set of virtual register references into a finite set of target-specific registers.
- **Instruction Scheduling II (Post-Register Allocation Scheduling):** introducing extra hazards and delays associated with a certain type of register that improve the instructions order.
- **Code Emission:** converts instruction from the `MachineInstr` representation to `MCInst` instances, in which there are two possible outcomes:
  - emit assembly code
  - emit binary code

3.4.1 Backend Tools

As for the backend, there is the possiblity to use some stand alone tools and the main one is: `llc`. With the command `llc --version` it is possible to list all the available backends:

```
LLVM (http://llvm.org/):
  LLVM version 13.0.1
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    mblaze - mblaze 32 bit
```

Considering an example source code file:

```
int main(){
        int i = 1;
        return i;
}
```

Is possible to do a step-by-step compilation:

```
$ ./clang -c -emit-llvm hello.c -o hello.bc
$ ./opt -mem2reg hello.bc -o hello.opt.bc
$ ./llc -march=x86 hello.opt.bc -o hello.x86
$ cat hello.x86

        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0
        .globl        _main                   ## -- Begin function main
        .p2align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## %bb.0:
        subl        $8, %esp
        .cfi_def_cfa_offset 12
        movl        $0, 4(%esp)
        movl        $1, (%esp)
        movl        (%esp), %eax
        addl        $8, %esp
        retl
        .cfi_endproc
                                        ## -- End function
.subsections_via_symbols
```

The backend implementation is scattered into different directories starting from `lib` path and in its sub-directories like: `CodeGen`, `MC`, `TableGen`, `Target`:

- **CodeGen:** contains implementation files and headers for all generic code generation algorithms: instruction selection, scheduler, register allocation.
- **MC:** holds the low-level functionality for the assembler, relaxation algorithm (*disassembler* implementation). There also some specific object file idioms such ELF file that will be illustrated later in the creation and implementation of the backend.
- **TableGen:** this is the most important tool to generate backend code; it takes in input files that describe the hardware of the target and generate other files that will complete the project's build. The format of this file is *td*; this is a custom format used to describe the hardware configuration inside LLVM.

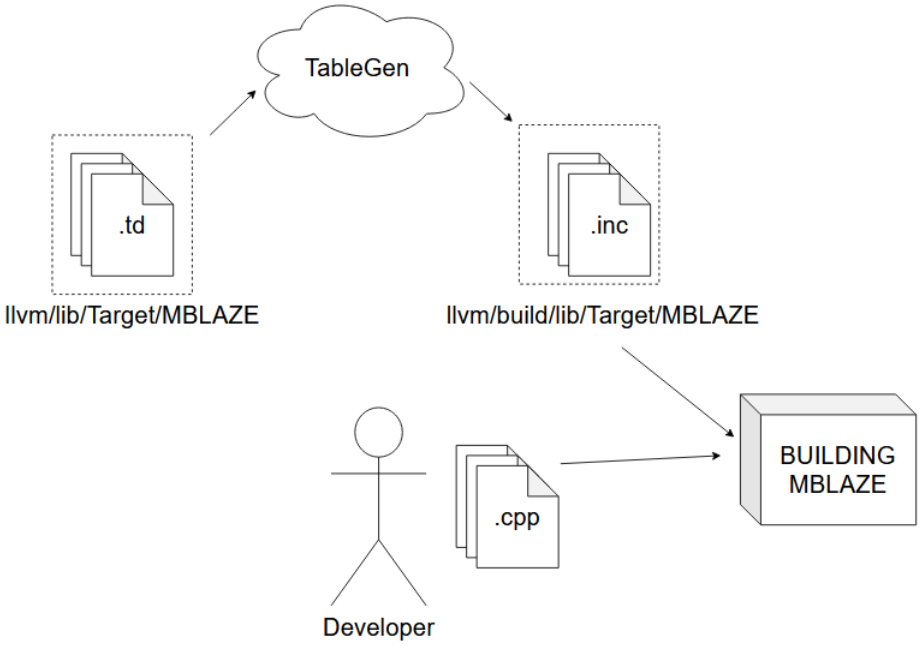


Figure 18: How .td files work

- **Target:** handles the core of different targets that are implemented with many *.cpp, .h, .td* files. This is the folder where backend developer spend most of their time defining the target description in *td* files and other information related with backend.

Figure 19: llvm tree

3.4.2 Backend Libraries

As the project is structured as libraries, also its inner components are library-based. For what concerns backend there are two type of libraries:

- **target-dependent:**
  - `AsmParser.a:` holds code to parse the assemlby text and implement an assembler.
  - `AsmPrinter.a:` contains code to print the assembly language and implement a backend able to generate assembly files.
  - `CodeGen.a:` includes code generation algorithms.
  - `MC.a:` is used to represent the program in the *lowest* level that LLVM allows.
  - `MCDisassembler.a:` this library manages the code to implement the *disassembler*.
  - `MCJIT.a:` holds the implementation for the *just-in-time* code generator.
  - `MCParser.a:` contains the interface to the MCAsmParser class and is able to parse the assembly text and perform partially the work of the assembler.

- `SelectionDAG.a`: this library includes *SelectionDAG* implementation.
- `Target.a`: this is where is possible to find the interfaces *target-independent* algorithms to request *target-dependent* functionality.

- **target-independent:**
  - `<Target>AsmParser.a`: this library contains *target-specific* part of the *AsmParser* library, responsible for implementing an assembler for the target machine.
  - `<Target>AsmPrinter.a`: includes printing's functionality of the target instructions and make possible for the backend to generate assembly language files.
  - `<Target>CodeGen.a`: holds the majority of the *target-dependent* functionality of the backend (register handling, instruction selection, scheduling, ...).
  - `<Target>Desc.a`: contains target information concerning the low level *MC* infrastructure and is related with the backend's ability of registering target specific *MC* objects (*MCCodeEmitter* for example).
  - `Disassembler.a`: complements the *MCDisassembler* with *target-dependent* functionality, that helps to build the system reading bytes and decoding them into *MCInsto* target instructions.
  - `<Target>Info.a`: it allows registering the target in the LLVM code generator system and also provides some classes that allow the *target-independent* code generator libraries to access target specific functionality.

3.4.3 Instruction Selection

The aim of this phase is to transform *LLVM IR* into a *SlectionDAG* nodes that represent target instructions. Once the DAG is created all the nodes need to go through the lowering, *DAG* combiner and legalization phases. Finally the instruction selection performs a *DAG* to *DAG* conversion using node pattern matching and transforms the *SelectionDAG* nodes into nodes representing target instructions.

What is Lowering? The idea is that any instruction can be"lowered" in order to be compliant with a specific architecture. It can be summarised as a *"simplification"* of *higher* level instructions to lower *level*, in order to be executed by the architecture.

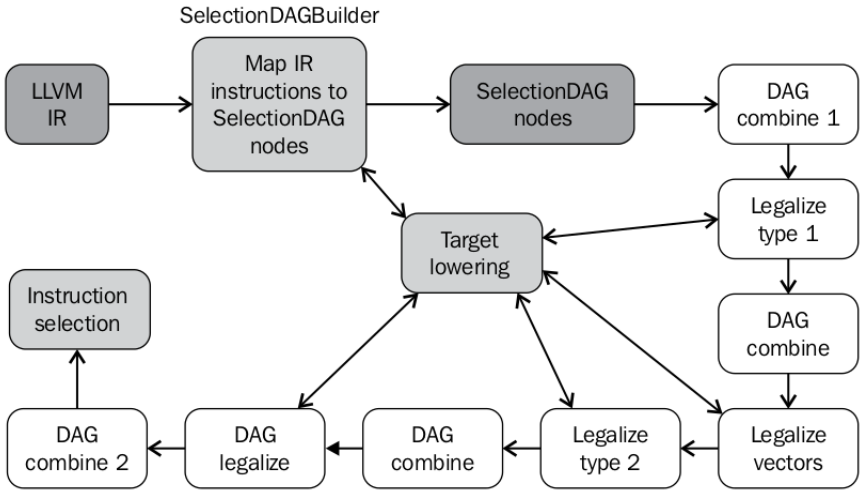


Figure 20: DAG Operation

The *SelectionDAGBuilder* visits every function and creates a *SelectionDAG* object for each basic block. During this first phase appears *TargetLowering* class: it is an abstract interface that is implemented by every target. This class helps the transformations of instructions in *SelectionDAG* nodes. The implementation is done by the `<Target>TargetLowerging` class inside all targets.

The output of this first phase is not ready yet and must pass through some additional transformations as follows:

- **DAG Combine:** it pass sub-optimal *SelectionDAG* constructions by matching a set of nodes and replacing them with an easier construct. The *general* implementation is located in the following path `llvm/lib/CodeGen/SelectionDAG/DAGCombiner.cpp`, while the *target-specific* one is located at `llvm/lib/Target/<Target_Name>/<Target>ISelLowering.cpp`. `setTargetDAGCombine()` is the method used to mark the nodes that need to be combined.

- **DAG Legalizer:** handles any operations with unsupported types (scalar or vectors). It supports the same actions: the promotion, expansion, and handling of custom nodes. This guarantees that the instruction selection is dealing only with *legal* types. *Legal* types are types that are natively supported by the target (it is possible to define to which classes are associated with).

After the *legalization* and *combine* phase it is time for the DAG to DAG Instruction selection phase. During this phase the aim is to transform target-independent nodes into target-specific ones, by using pattern matching. The instruction selection works on *SelectionDAG*, one instance at time.

Pattern Matching Each Target handles instruction selection by implementing the *select* method from their `SelectionDAGISel` subclass naming it like: `<Target>DAGToDAGISel`. This method takes in input an *SDNode* as parameter to be matched and returns an *SDNode* value representing a physical instruction (otherwise comes up with error). The *SDNode* stands for *SelectionDAG* nodes, that represent target instruction.

The select method has two ways to match instructions:

1. `select()` calls `SelectCode()`; TableGen generates the `SelectCode()` method for each target and in this code, TableGen generates the MatcherTable, mapping ISD and `<Target>ISD` nodes to physical-instruction nodes. The matcher table is generated from the instruction definitions in the *.td* files (usually, `<Target>InstrInfo.td`). The `SelectCode()` method ends by calling `SelectCodeCommon()` , a target-independent method to match the nodes by using the target matcher table.

2. Provide custom matching code in Select prior to the SelectCode invocation.

3.4.4 Fast Instruction Selection

LLVM supports alternative instruction selection implementation as follows:

- fast instruction selection `FASTISEL` class, which live in the `<llvm_source> /lib/CodeGen/SelectionDAG/FastISel.cpp`. The aim is to provide a quick code generation, which is more like doing -o0 optimization (during the optimization phase). The speed gain occurs by avoiding complicated folding and lowering logic.

3.4.5 Scheduler

After the instruction selection phase the scheduler works on the selection *DAG* nodes. There are few different schedulers to choose from and this can be done in `llc` using the `-pre-RA-sched=<scheduler name>` option. The possible scheduler value are:

- *list-ilp, list-hybrid, source, list-burr:* these options refer to list scheduling algorithms implemented by the `ScheduleDAGRRList` class.

- *fast:* `ScheduleDAGFast` class implements a suboptimal, but fast scheduler.

- *vliw-td:* vliw-specific scheduler implemented byt the `ScheduleDAGVLIW` class.

3.4.6 Machine Instructions

During this phase, thanks to the presence of *InstrEmitter*, the *SDNodes* are transformed into *MachineInstr* format. This representation is closer to the target than IR instructions. Each *Machine Instruction* contains an *opcode*

number and list of operands. Their format is a three-address representation of the program, more similar to a list of instruction rather than a DAG. Is possible to dump machine instructions information using the following option: `-print-machineinstr=<pass-name>`. *Machine Instruction* holds also miscellaneous information about the instructions: defined registers, register and memory operands, instruction type, predicates, ... All this information are helpful to the others step of compilation to perform better analysis.

3.4.7 Register Allocation

The vast use of variables in programs need to eventually be mapped onto physical registers. Saving variables inside of register means that these values can be available more quickly, but there is a limited amount of registers. Due to the fact that is not possible to allocate every variables to register is important to define which ones will ended up inside registers and which in RAM (*spilling* operation). During the lifetime of a program, a variable can be both *spilled* and stored in registers (this variable is then considered as *"split"*). An optimizing compiler aims to assign as many variables to registers as possible; accessing RAM is slower than accessing registers and so a compiled program may runs slower. At the same time,it also aims at reducing the number of memory spills.

So, in that phase takes place the register allocation and it has to to transform a number of virtual register into physical ones. The default Register Allocator is mapped to one of the four options and is selected depending on the current optimization level (the -O option). Although the register allocator, regardless of the chosen algorithm, is implemented in a single pass, it still depends on other analyses, composing the allocator framework. There are a few passes used in the allocator framework and they are shown here in the following schema:

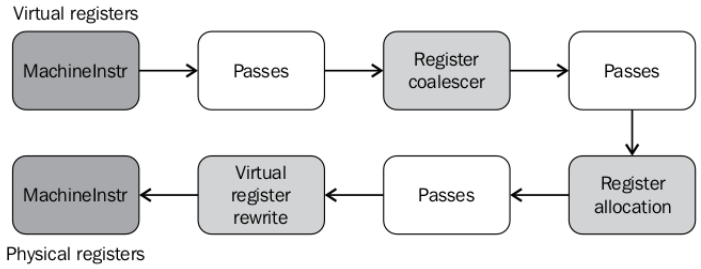


Figure 21: Register Allocator

3.4.8 Register Coalescer

The register coalescer removes redundant copy instructions and it is implemented in the *RegisterCoalescer*. During coalescing, the method `joinAllIntervals()` iterates over a list of copy instructions. The *joinCopy()* method creates *Coa-*

lescerPair instances from copy machine instructions and coalesces copies away whenever possible. The *interval* is a pair of program points, start and end, which starts when a value is produced and lasts while this value is held in a temporary location until it is finally used, that is, killed.

3.4.9 Virtual Register Rewrite

The register allocation pass selects the physical registers to be used for each virtual one.

3.4.10 Target Hooks

During coalescing, virtual registers need to be transformed from compatible register classes in order to be successfully coalesced. The code generator acquires this information from target-specific descriptions obtained by abstract methods. The allocator obtains all the information related to a register from `TargetRegisterInfo`'s subclasses ; this information defined if it is reserved or not, its parent register classes, and also if it is physical or virtual.

The `<Target>InstrInfo` class is another data structure that provides target-specific information that is necessary for register allocation. Here some examples:

- `isLoadFromStackSlot()` and `isStoreToStackSlot():` from `<Target>InstrInfo`, are used during *spill* code generation to discover whether the machine instruction is a memory access to a stack slot.
- *Spiller:* generates target-specific memory access instructions to stack slots using the `storeRegToStackSlot()` and `loadRegFromStackSlot()` methods.
- *Copy:* the `copyPhysReg()` method is used to generate a target-specific register copy, even among different register classes when necessary.

3.4.11 Prologue and Epilogue

Introduction: function prologue consists of a few of lines of code at the beginning of a function. These lines prepare the stack and registers for use within the function. Equally, the function epilogue appears at the end of the function and restores both the stack and registers to the state they were in, before the function was called.

The prologue and epilogue are not a part of the assembly language itself; they represent a convention used by assembly language programmers and compilers, of many higher-level languages. They are fairly rigid, having the same form in each function.

Prologue: A function prologue typically does the following actions if the architecture has a base pointer (frame pointer) and a stack pointer:

- Moves forward the current base pointer inside the stack, so it can be restored later. The value of the base pointer is set to the address of the stack pointer (it is pointed to the top of the stack), so that the base pointer will point to the top of the stack.
- Pushes the stack pointer further by decreasing or increasing its value, depending on whether the stack grows down or up.

Epilogue: Function epilogue reverses the actions of the function prologue and returns control to the calling function. It typically does the following operations:

- Releases the stack pointer to the current base pointer, so space reserved in the prologue for local variables is freed. Pops the base pointer off the stack, so it is restored to its value before the prologue. Returns to the calling function, by popping the previous frame's program counter off the stack and jumping to it.
- Epilogue will reverse the effects of either of the above prologues. Under certain calling conventions it is the *callee*'s responsibility to clean the arguments off the stack, so the epilogue can also include the step of moving the stack pointer down or up.

Functions need both: prologue and an epilogue to be complete. The former sets up the stack frame and *callee-saved* registers during the beginning of a function, whereas the latter cleans up the stack frame prior to function return.

3.4.12 Frame Indexes

LLVM uses a virtual stack frame during the code generation, and stack elements are referred using frame indexes. The prologue insertion allocates the stack frame and gives enough target-specific information to the code generator to replace virtual frame indices with real (target-specific) stack references. The method `eliminateFrameIndex()` in the `<Target>RegisterInfo` class implements this replacement by converting each frame index to a real stack offset for all machine instructions that contain stack references (usually loads and stores). Extra instructions are also generated whenever additional stack offset arithmetic is necessary. See the file `<llvm_source>/lib/Target/<Target>/<Target>RegisterInfo.cpp` for examples.

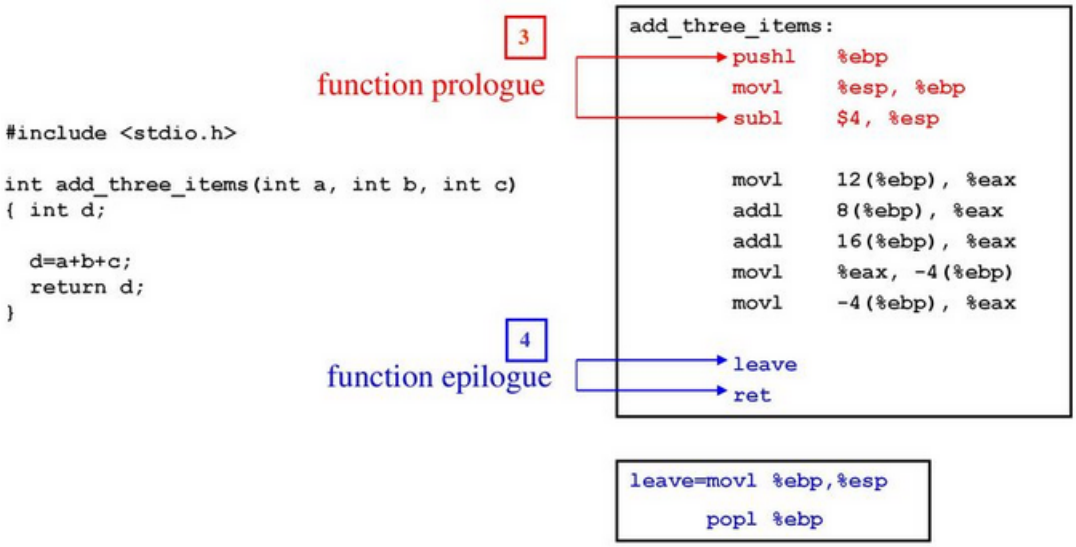


Figure 22: Example taken from: https://slideplayer.com/slide/13418616/

3.4.13 Machine Code Framework

The machine code (MC) classes comprehends an entire framework for low-level manipulation of functions and instructions. This is a new framework that was designed to aid in the creation of LLVM-based assemblers and disassemblers. In the beginning LLVM lacked an integrated assembler and it was only able to proceed with the compilation until the assembly language emission step, which created an assembly text file as output and depended on external tools to carry on the rest of the compilation (assembler and linker).

3.4.14 MC Instructions

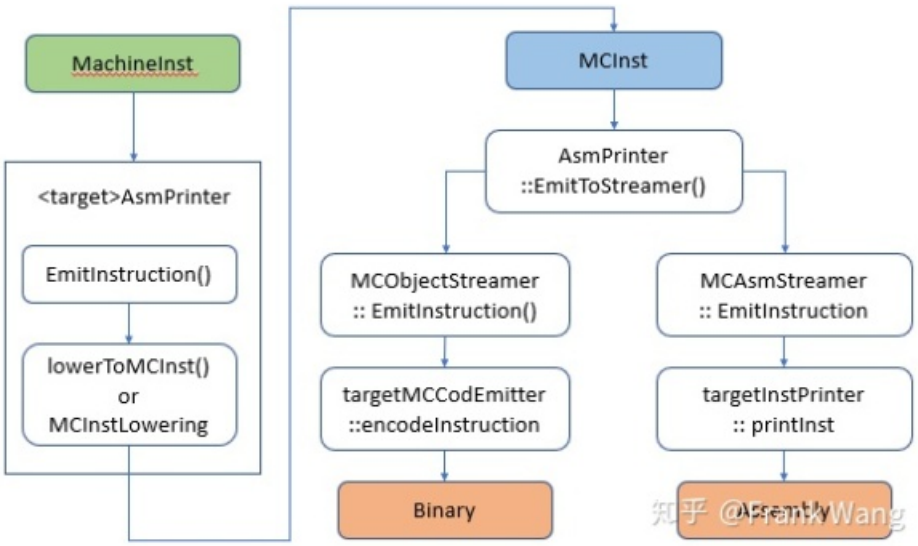


Figure 23: Example taken from: https://llvm.liuxfe.com/post/62627

In the MC framework, machine code instructions (*MCInst*) replace machine instructions (*MachineInstr*). The *MCInst* class, defined in the `<llvm_source>/include/llvm/MC/MCInst.h` file, defines a lightweight representation for instructions. Compared to *MI*, the *MCInst* carries less information about the program. For instance, an *MCInst* instance can be created not only by a backend, but also by a disassembler right out of binary code. It encodes the view of an assembler, that is, a tool whose purpose is not to apply rich optimizations, but rather to organize instructions in the object file. Each operand can be:

- register;
- immediate;
- expression (used to represent label computations and relocations);
- MCInstr instance;

The *MI* instructions are converted to *MCInst* instances early in the code emission phase, which is the subject of the next subsection.

3.4.15 Code Emission

After register allocation, *Code Emission* takes place. It starts with the assembly printer (*AsmPrinter*).

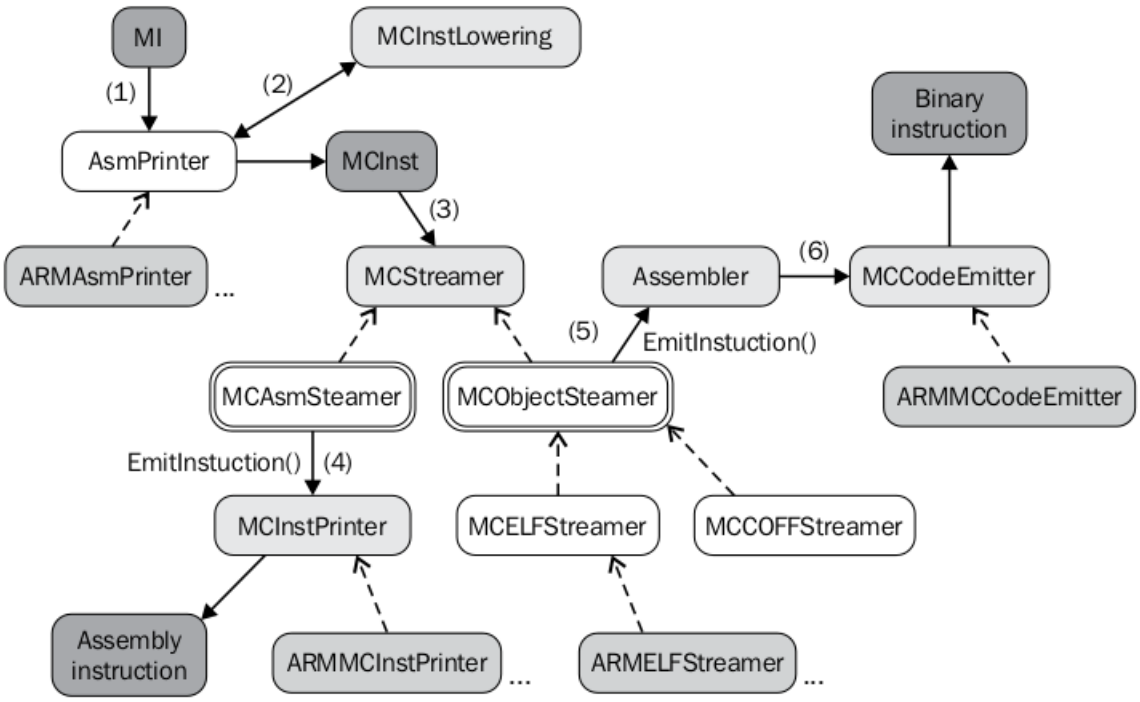


Figure 24: Example taken from: https://slideplayer.com/slide/13418616/

There is an overview on the diagram:

- `AsmPrinter:` is a machine functions that emit function's header and iterates over all basic blocks, while dispatching one *Machine Instruction* at the time to the `EmitInstruction()`. Each target in the backend provides an *AsmPrinter* subclass that overloads this basic method.
- `<Target>AsmPrinter::EmitInstruction():` this method obtains a *Machine Instruction* and transforms it into *MCInst* with the help of `MCInstLowering` interface. All targets in LLVM provides a subclass to this interface and custom code to generate *MCInst* instances.
- `MCStreamer:` process a stream of *MCInst* instructions to emit them to the chosen output via 2 subclasses:
  - `MCAsmStreamer`
  - `MCObjectStreamer`

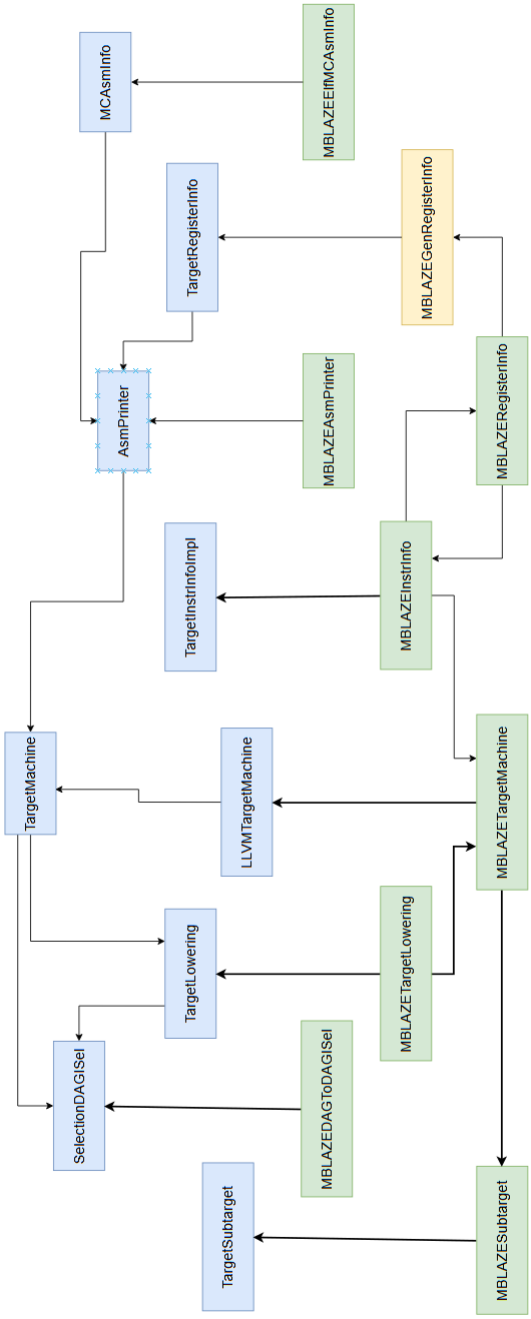
  At the end of this phase the former converts *MCInst* to assembly language and the latter converts it to binary instructions.

- `MCAsmStreamer::EmitInstruction()`: after generating a binary instructions there is a specialize target/object method (`MCObjectStreamer::EmitInstruction()` calls the LLVM object code assembler.
- `MCCodeEmitter::EncodeInstruction()`: this method provides encoding and dumping binary functionality for a specifi target.

4 Writing a custom backend in LLVM

4.1 Introduction

LLVM provides developers with an extensive generic framework for code generation that prescribes a fixed directory layout and class hierarchy template. Each backend is located in its own sub-directory within lib/Target, where most of the code to be implemented goes. Only a bunch of files of the original in the source tree have to be modified in order to integrate the new backend with the existing LLVM codebase. The hierarchy inside the project make possible to keep the bulk of the actual algorithms and procedures fully target-independent by accessing all the required target-specific information through the specified interfaces. A slightly simplified overview of the class hierarchy is shown in the next page. Pre-existing framework classes are coloured in blue, green denotes a manually implemented backend class, and yellow marks a class automatically generated by the *tblgen* tool.


MCAsmInfo
MBLAZEEfMCAsmInfo
TargetRegisterInfo
MBLAZEGenRegisterInfo
AsmPrinter
MBLAZEAsmPrinter
MBLAZERegisterInfo
TargetInstrInfoImpl
MBLAZEInstrInfo
TargetMachine
LLVMTargetMachine
MBLAZETargetMachine
TargetLowering
MBLAZETargetLowering
SelectionDAGISel
MBLAZEDAGToDAGISel
TargetSubtarget
MBLAZESubtarget

4.2 LLVM Integration

To add a new backend the target need to be registered inside the triple format templates. As it was written in the previous chapter this format handles all the possible targets inside the system The backend system is based on *Triple Design*. Every available target is registered with this pair keys:

- Architecture
- Vendor
- Operating System

For example: `x86-pc-linux-gnu` is an `x86` architecture, by `pc` vendor, on `linux-gnu` operating system. If there is the need to support compilation of C programs directly from source, integration with a C frontend is required (Clang for example).

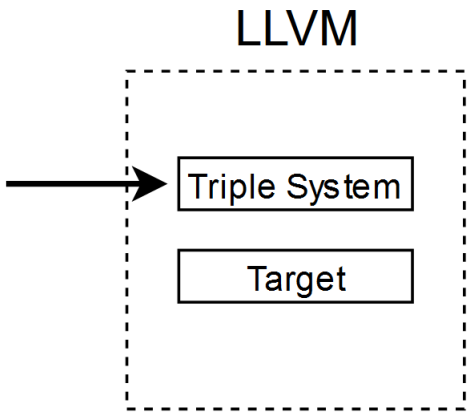


Figure 25: Triple format system position

There is a specific path that will be defined in the next chapter that holds all this information and also the frontend configuration to generate code starting directly from it.

4.3 Target Description

4.3.1 Instructions Formats

Taking the information about instruction formats from the first chapter, they need to be written inside the `MBLAZEInstrFormats.td` file. This file holds all possible instruction formats divided into different classes:

```
class TA<bits<6> op, bits<11> flags, dag outs, dag ins, string asmstr,
         list<dag> pattern> :
         InstMBLAZE<4, outs, ins, asmstr, pattern>
{
  bits<5> rd;
  bits<5> ra;
  bits<5> rb;

  let Inst{0-5} = op;
  let Inst{6-10}  = rd;
  let Inst{11-15} = ra;
  let Inst{16-20} = rb;
  let Inst{21-31} = flags;
}
```

This is a *TA* format class that is generated from its master class `InstMBLAZE`. Each of the arguments of TA class have an important role:

1. `bits<6> op`: this 6 bits indicate the exact type of operation.
2. `bits<11> flags`: this defines the flags (if there is any of it).
3. `dag out/ins`: these parameters are dag type and it is a special type that `TableGen` uses to `SelectionDAG` nodes. These nodes are used during the instruction selection phase to represent opcodes, registers or constants. In this specific case there is a *outs* node that denotes that its children are output operands, while *ins* indicates that are input ones.
4. `string asmstr`: represents the instruction assembly string.
5. `list<dag> pattern`: the list of dag objects that will be used to perform pattern matching during instruction selection

The master class is a generic template that is used to generate all other class:

```
class InstMBLAZE<int sz, dag outs, dag ins, string asmstr, list<dag> pattern>
    : Instruction, Encoding64 {

  let Namespace = "MBLAZE";
  dag OutOperandList = outs;          // A dag containing the MI def operand list.
  dag InOperandList = ins;          // A dag containing the MI use operand list.
  let AsmString = asmstr;         // The .s format to print the instruction with.
```

```
  let Pattern = pattern;        // Set to the DAG pattern for this instruction.
  let Size = sz;
  ...
```

This is a standard template that you can directly take from: `https://llvm.org/docs/WritingAnLLVMBackend.html#instruction-set` or copy it from another backend and then do some tweaks and adjustments to be compliant to the hardware specification.

4.3.2 Instructions Description

The instructions need to be implemented using the formats that have have defined before. All instructions are defined inside `MBLAZEInstrInfo.td` and they all are generated from the two main type of instructions and their subgroups that have been described in the first chapter. This is an example of the `ADD` implementation:

```
class Arith<bits<6> op, bits<11> flags, string instr_asm> :
            TA<op, flags, (outs GPR32:$dst), (ins GPR32:$b, GPR32:$c),
               !strconcat(instr_asm, "   $dst, $b, $c"),
               []>;
...
def ADD_rrr : Arith<0x00, 0x000, "add">;
def ADD_rrlimm : ArithI<0x08, "addi">;
...

defm : MultiPat<add, ADD_rrr, ADD_rru6, ADD_rrlimm>;
```

The last line of code is necessary to register correctly the instructions; if this is not done during the DAG lowering phase some nodes are not goint to be matched with the correct instruction inside the backend.

One way of defining all the instructions is to create a subgroup of instruction starting from the main type and the defining the instructions itself and this was done for the following instructions inside the MicroBlaze backend:

1. ADD,ADDI
2. RSUBK, RSUBKI
3. OR, ORI
4. AND, ANDI
5. XOR, XORI
6. BGE, BEQ, BNE, BGT, BLT, BLE
7. LW, LWI, LHU, LHUI, LBU, LBUI

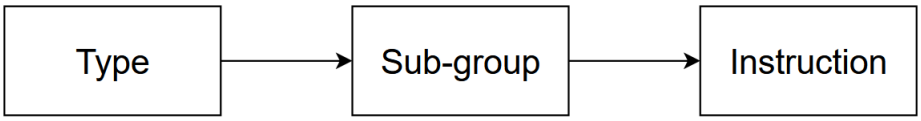


Figure 26: Classic instructions definition

Another method is a to make use of the multiclass system. With multiclass is possible to define multiple variant of the same of instructions at the same type. This behaviour is defined inside the following snippets of code taken from ARC target:

```
// Generic 2-operand unary instructions.
multiclass ArcUnaryInst<bits<5> major, bits<6> subop,
                        string opasm> {
  def _rr : F32_SOP_RR<major, subop, 0, (outs GPR32:$B), (ins GPR32:$C),
                       !strconcat(opasm, "\t$B, $C"), []>;

  def _f_rr : F32_SOP_RR<major, subop, 1, (outs GPR32:$B), (ins GPR32:$C),
                       !strconcat(opasm, ".f\t$B, $C"), []>
  { let Defs = [STATUS32]; }
}
```

It starts defining a multiclass in which the are the possible variants of the instructions that will be defined. In each method that defined the possible variants (`_rr` and `_f_rr`) are called the specific format that the architecture required directly from `ARCInstrFormats.td`. Then there is another specialization of the class that define a first encoding with hexadecimal numbers:

```
multiclass ArcUnaryGEN4Inst<bits<6> mincode, string opasm> :
  ArcUnaryInst<0b00100, mincode, opasm>;
```

And last we have a multiple definition (`defm`) of the actual instructions:

```
// General unary instruction definitions.
defm SEXB : ArcUnaryGEN4Inst<0b000101, "sexb">;
defm SEXH : ArcUnaryGEN4Inst<0b000110, "sexh">;
```

With this multiple defintion are being defined the following variant:

1. SEXB_rr, SEXB_f_rr
2. SEXH_rr, SEXH_f_rr

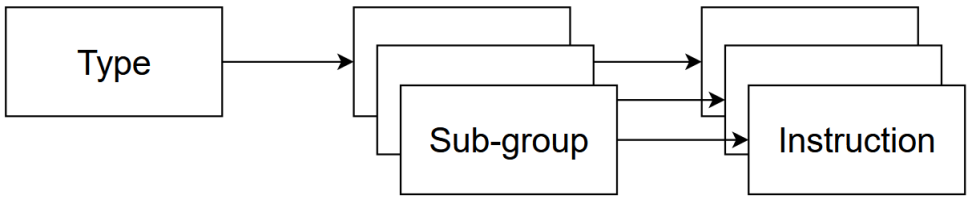


Figure 27: Definition of instructions using multipleclass method

4.3.3 Register Description

All information about registers need to be defined inside the following file: `TargetRegisterInfo.td`. To write a definition of a register must be defined a general class that indicates the encoding of the register and then specialize this class in a sub-class that defines more information: if it is a special register class or a general purpose one.

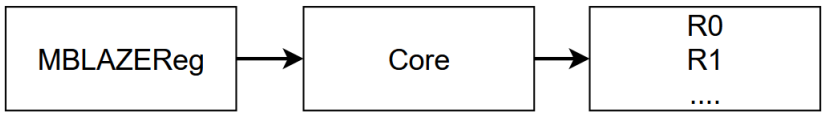


Figure 28: Register definition

This will be translated in code as follows:

```
1  class MBLAZEReg<string n, list<string> altNames> : Register<n, altNames> {
2    field bits<5> HwEncoding;
3    let Namespace = "MBLAZE";
4  }
5
6  class Core<int num, string n, list<string>altNames=[]> : MBLAZEReg<n, altNames> {
7    let HWEncoding = num;
8  }
9
10 def R0 : Core< 0, "%r0">, DwarfRegNum<[0]>;
11 ...
```

At first a general class for all *MicroBlaze* registers is defined and indicates how many bits are going to be used for the encoding of the registers. Then there is a specialization of this class with`Core` that will be used to define each register. All registers are defined in the same way, the only thing that change is the name based their behaviour.

4.3.4 Calling Convention

Last target description file that need to be handled is the calling convention. It is going to define which register are going to return values and which registers are going to be used to pass arguments. The syntax is the same in all backends so the only thing to do is to read the documentation and setup correctly the information inside of this file.

```
//===----------------------------------------------------------------------===//
// MBLAZE Return Value Calling Convention
//===----------------------------------------------------------------------===//
def RetCC_MBLAZE : CallingConv<[
  CCIfType<[i32, i64], CCAssignToReg<[R3, R4]>>,

  CCIfType<[i64], CCAssignToStack<8, 4>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;

//===----------------------------------------------------------------------===//
// MBLAZE Argument Calling Conventions
//===----------------------------------------------------------------------===//
def CC_MBLAZE : CallingConv<[
  // Promote i8/i16 arguments to i32.
  CCIfType<[i8, i16], CCPromoteToType<i32>>,

  // The first 8 integer arguments are passed in integer registers.
  CCIfType<[i32, i64], CCAssignToReg<[R0, R4, R2, R3, R4, R5, R6, R7]>>,

  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<[i64], CCAssignToStack<8, 4>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;

def CSR_MBLAZE : CalleeSavedRegs<(add (sequence "R%u", 21, 26), GP, FP)>;
```

4.4 Backend completion

TableGen will handle all the *td* files, but to complete the backend the output of *TableGen* need to be included inside the *cpp* files. The following files are necessary to build correctly the backend:

1. `MBLAZETargetInfo.cpp`
2. `MBLAZETargetMachine.cpp`
3. `MBLAZESubtarget.cpp`
4. `MBLAZEDAGToDAGISel.cpp`
5. `MBLAZEISelLowering.cpp`
6. `MBLAZEAsmPrinter.cpp`
7. `MBLAZEMCTargetDesc.cpp`

They represent the minimum, except for the target description files, the the target must have.

4.4.1 TargetInfo

The first one holds the information of the backend that is displayed when `llc --version` command is launched and this is the implementation:

```
1 Target &llvm::getTheMBLAZETarget() {
2   static Target TheMBLAZETarget;
3   return TheMBLAZETarget;
4 }
5
6 extern "C" LLVM_EXTERNAL_VISIBILITY void LLVMInitializeMBLAZETargetInfo() {
7     RegisterTarget<Triple::mblaze> X(getTheMBLAZETarget(), "mblaze",
8     "mblaze 32 bit", "MBLAZE");
9 }
```

4.4.2 TargetMachine

To understand how `MBLAZETargetMachine.cpp` there is the need to talk about `LLVMTargetMachine`. It is designed as a base class for targets implemented with the LLVM target-independent code generator. The `LLVMTargetMachine` class should be specialized by a concrete target class that implements the various virtual methods. LLVMTargetMachine is defined as a subclass of TargetMachine in `include/llvm/Target/TargetMachine.h`. The implementation of `MBLAZETargetMachine` must have access methods to obtain objects that represent target components. These methods are named *get*Info*, and are intended to obtain the instruction set (*getInstrInfo*), register set (*getRegisterInfo*), stack frame layout (*getFrameInfo*), and similar information. Some version before

this methods were implemented directly into `MBLAZETargetMachine.cpp/h`, but now they are defined inside `MBLAZESubtarget.h` and so in the header file is described how `MBLAZETargetMachine` retrieve all the necessary information from `MBLAZESubtarget`:

```
class MBLAZETargetMachine : public LLVMTargetMachine {
...
public:
  MBLAZETargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                      StringRef FS, const TargetOptions &Options,
                      Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                      CodeGenOpt::Level OL, bool JIT);
  ~MBLAZETargetMachine() override;

  const MBLAZESubtarget *getSubtargetImpl() const { return &Subtarget; }
  const MBLAZESubtarget *getSubtargetImpl(const Function &) const override {
    return &Subtarget;
  }
```

The implementation of this class inside the *cpp* is in the following code snippets:

```
namespace {
class MBLAZEPassConfig : public TargetPassConfig {
public:
  MBLAZEPassConfig(MBLAZETargetMachine &TM, PassManagerBase &PM)
      : TargetPassConfig(TM, PM) {}

  MBLAZETargetMachine &getMBLAZETargetMachine() const {
    return getTM<MBLAZETargetMachine>();
  }

  bool addInstSelector() override;
  void addPreEmitPass() override;
  void addPreRegAlloc() override;
};

}

TargetPassConfig *MBLAZETargetMachine::createPassConfig(PassManagerBase &PM) {
  return new MBLAZEPassConfig(*this, PM);
}
...
```

4.4.3 Selection DAG Implementation

The selection DAG phase is implemented in `MBLAZEDAGToDAGISel.cpp` in the class with the same name. Since most of the functionality is generated from

target description, at first is possible to just override the `Select()` function and forward it to the `SelectCode` function. Because this class is a machine function pass, the name for the pass need to be provided. The main bulk of the implementation comes from the generated file, that is included in the middle of the class:

```
class MBLAZEDAGToDAGISel : public SelectionDAGISel {
const MBLAZESubtarget *Subtarget;
public:
        MBLAZEDAGToDAGISel(MBLAZETargetMachine &TM, CodeGenOpt::Level OptLevel)
        : SelectionDAGISel(TM, OptLevel) {}

        StringRef getPassName() const override {
        return "MBLAZE DAG->DAG Pattern Instruction Selection";
}

#include "MBLAZEGenDAGISel.inc"

        void Select(SDNode *Node) override {
                SelectCode(Node);
        }
};
```

At the end of the file the function to create the pass in this file:

```
/// This pass converts a legalized DAG into a MBLAZE-specific DAG, ready for
/// instruction scheduling.
FunctionPass *llvm::createMBLAZEISelDag(MBLAZETargetMachine &TM,
                                        CodeGenOpt::Level OptLevel) {
  return new MBLAZEDAGToDAGISel(TM, OptLevel);
}
```

4.4.4 Lowering Phase

In this phase is involved `MBLAZEISelLowering.cpp/h` file and it defines the lowering methods for the element of the DAG. Starting from the definition of nodes like (inside `MBLAZEISelLowering.h`):

```
enum NodeType : unsigned {
  // Start the numbering where the builtin ops and target ops leave off.
  FIRST_NUMBER = ISD::BUILTIN_OP_END,

  // Branch and link (call)
  BL,

  // Jump and link (indirect call)
  JL,

```

```
  //RETURN
  RET,
  ...
```

Now the lowering need to be implemented. Look at the *RET* node: it is defines inside the header, but before being defined here it is defined inside `MBLAZEInstrInfo.td` as:

```
// Call return
def ret       : SDNode<"MBLAZEISD::RET", SDTNone,
                       [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
```

The lowering is done by this method `SDValue MBLAZETargetLowering::LowerReturn` inside `MBLAZEISelLowering.cpp`; this method has a lot of arguments (all defined inside the `TargetLowering` superclass), but the most important ones are *Outs* vector, which holds the description of the return argument and the *OutVals* vector, which holds the DAG nodes for the return values:

```
SDValue
MBLAZETargetLowering::LowerReturn(SDValue Chain, CallingConv::ID CallConv,
                                  bool IsVarArg,
                                  const SmallVectorImpl<ISD::OutputArg> &Outs,
                                  const SmallVectorImpl<SDValue> &OutVals,
                                  const SDLoc &dl, SelectionDAG &DAG) const {
                                  ...
```

To make a proper decision about how to handle the return is better to analyze it with a specific class: `CCState`. The result of the analysis define how the return i going to be managed:

1. **void function**: directly return without doing anything more
2. **not void function**: otherwise, there is loop over all the return arguments. For each return argument, there is have an instance of the `CCValAssign` class, which described how has to be handled the argument.

```
  for (unsigned i = 0, e = RVLocs.size(); i != e; ++i) {
    CCValAssign &VA = RVLocs[i];
    ...
```

At the end of this analysis is possible to call the return:

```
 return DAG.getNode(MBLAZEISD::RET, dl, MVT::Other, RetOps);
```

This approach is a standard in most of backend, so is possible to take the structure of the method from another target and then replace the information about the nodes with the ones has been created. The operation flows remains the same: get the node, analyse it, lower it.

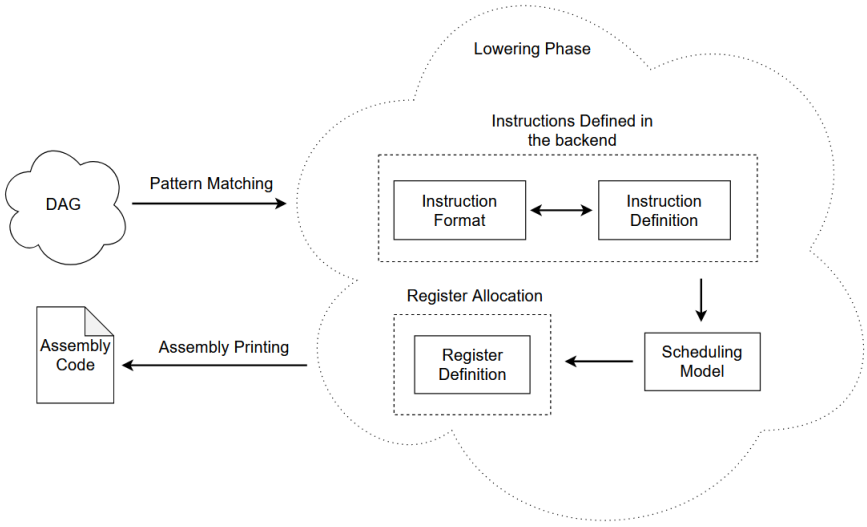


Figure 29: Instructions workflow: from DAG to lowering

4.4.5 Assembly Printing

If all the previous passes went right now is possible to implement the assembly code generation. This is done by combining informations that were previously defined inside:

1. `MBLAZEInstrInfo.td` that has instruction definitions.
2. `MBLAZEAsmInfo.cpp/h` which contains target-specific values for *TargetAsmInfo* properties and sometimes new implementations for methods.
3. `MBLAZEAsmPrinter.cpp` implements the *AsmPrinter* class performing the LLVM-to-assembly conversion.
4. `MBLAZEInstPrinter.cpp/h` which holds all the method that need to be invocated to print information about target instructions.

The second file defines some more information that will be used by the *AsmPrinter* and has this definition of the header:

```
1   #include "llvm/MC/MCAsmInfoELF.h"
2
3  namespace llvm {
4
5  class Triple;
6
7  class MBLAZEMCAsmInfo : public MCAsmInfoELF {
8    void anchor() override;
```

```
9
10  public:
11    explicit MBLAZEMCAsmInfo(const Triple &TT);
12  };
```

and of its *cpp* file:

```
1   MBLAZEMCAsmInfo::MBLAZEMCAsmInfo(const Triple &TT) {
2     SupportsDebugInformation = true;
3     Data16bitsDirective = "\t.short\t";
4     Data32bitsDirective = "\t.word\t";
5     Data64bitsDirective = nullptr;
6     ZeroDirective = "\t.space\t";
7     CommentString = ";";
```

This are just informations that are going to be used to generate assembly code. There is the symbol of the comments, if debug information is supported or not, etc. ...

To print assembly code two other files need to be added. Inside `MBLAZEAsmPrinter` file the `runOnMachineFunction()` method is invoked, which first prints out the header of the function and then processes its basic blocks.

```
1   class MBLAZEAsmPrinter : public AsmPrinter {
2     MBLAZEMCInstLower MCInstLowering;
3
4   public:
5     explicit MBLAZEAsmPrinter(TargetMachine &TM,
6                               std::unique_ptr<MCStreamer> Streamer)
7         : AsmPrinter(TM, std::move(Streamer)),
8           MCInstLowering(&OutContext, *this) {}
9
10    StringRef getPassName() const override { return "MBLAZE Assembly Printer"; }
11    void emitInstruction(const MachineInstr *MI) override;
12
13    bool runOnMachineFunction(MachineFunction &MF) override;
14  };
```

Instead in `MBLAZEInstPrinter` there is the definition of each method that print information about some particular aspect of the target: register name (*printRegName*), operands (*printOperand*), etc. ... This is the configuration of the header file:

```
1   namespace llvm {
2
3   class MBLAZEInstPrinter : public MCInstPrinter {
4   public:
5     MBLAZEInstPrinter(const MCAsmInfo &MAI, const MCInstrInfo &MII,
6                       const MCRegisterInfo &MRI)
```

```
7         : MCInstPrinter(MAI, MII, MRI) {}
8
9   // Autogenerated by tblgen.
10  std::pair<const char *, uint64_t> getMnemonic(const MCInst *MI) override;
11  void printInstruction(const MCInst *MI, uint64_t Address, raw_ostream &O);
12  static const char *getRegisterName(unsigned RegNo);
13
14  void printRegName(raw_ostream &OS, unsigned RegNo) const override;
15  void printInst(const MCInst *MI, uint64_t Address, StringRef Annot,
16                                                    const MCSubtargetInfo &STI, raw_os
```

and this is the main *cpp* file:

```
1  void MBLAZEInstPrinter::printRegName(raw_ostream &OS, unsigned RegNo) const {
2    OS << StringRef(getRegisterName(RegNo)).lower();
3  }
4
5  void MBLAZEInstPrinter::printInst(const MCInst *MI, uint64_t Address,
6                                    StringRef Annot, const MCSubtargetInfo &STI,
7                                    raw_ostream &O) {
8    printInstruction(MI, Address, O);
9    printAnnotation(O, Annot);
10 }
```

The arguments of this method are often related with the other part of the backend, so take a closer look at this schema to have an idea of what class brought us here:

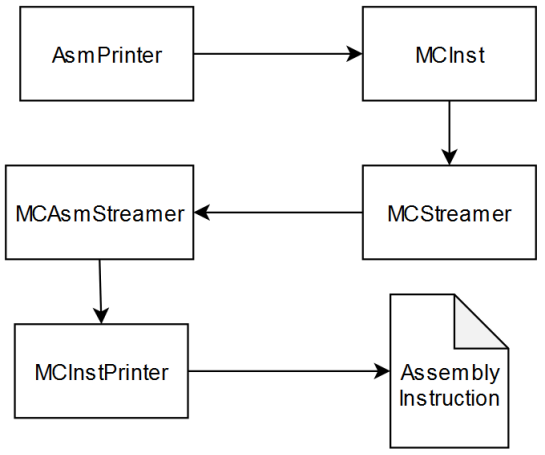


Figure 30: Assembly printing path

The *MCStreamer* class processes a stream of MCInst instructions to emit

them to *MCAsmStreamer*. At this point the method used to print assembly instruction are called from *MCInstPrinter* and this is the end of assembly production.

4.5 Summary

Following this list of instructions the reader should have understood the basic steps that are needed to write a custom backend. There is a more detailed guide in the next chapter, in which there will be more code snippets and explanation of the actual implementation of *MBLAZE* inside LLVM project.

5 Technical Implementation of the Backend

5.1 Introduction

This chapter will be focused on the actual implementation of some of the method and files that have been seen in the previous chapter. There will be no more explanation in depth about how the backend or llvm project work, but just some hint if needed.

5.2 Adding MBLAZE to Triple Class

As it was illustrated the backend is based on a *triple* system and in order to be able to use *MicroBlaze* the *mblaze* member need to added to the `ArchType` enumeration in `llvm/include/llvm/ADT/Triple.h`.

```
class Triple {
public:
    enum ArchType{
        ...
        mblaze,         // MBLAZE
        ...
        }
    }
}
```

After that `llvm/lib/Support/Triple.cpp` must be extended in the`getArchTypeName()` method by inserting the new backend:

```
StringRef Triple::getArchTypeName(ArchType Kind) {
  switch (Kind) {
    ...
    case mblaze:          return "mblaze";
    ...
    }
}
```

All the following method need to be expanded with the same information:

- `getArchTypePrefix(ArchType Kind)`
- `getArchTypeForLLVMName(StringRef Name)`
- `ArchType parseArch(StringRef ArchName)`
- `ObjectFormatType getDefaultFormat(const Triple &T)`
- `getArchPointerBitWidth(llvm::Triple::ArchType Arch)`
- `get32BitArchVariant()`

5.3 Adding MBLAZE to Clang frontend

If the user wants to build the target with `-DLLVM_ENABLE_PROJECTS="clang"` option, it is important to register the new target within the frontend. Some lined of the code need to be added at `clang/lib/Basic/CMakeLists.txt`:

```
1       Targets/MBLAZE.cpp
```

Then open `clang/lib/Basic/Targets.cpp` and add:

```
1       #include "Targets/MBLAZE.h"
```

Then in the `switch (Triple.getArch())` the case of MicroBlaze target need to be inserted:

```
1     case llvm::Triple::mblaze:
2         return new MBLAZETargetInfo(Triple, Opts);
```

Now is time to create the following file`LLVM_13_MBLAZE/clang/lib/Basic/Targets/MBLAZE.cpp` as follows:

```
1  #include "MBLAZE.h"
2  #include "clang/Basic/MacroBuilder.h"
3  #include "llvm/ADT/StringSwitch.h"
4
5  using namespace clang;
6  using namespace clang::targets;
7
8  const char *const MBLAZETargetInfo::GCCRegNames[] = {
9    // Integer registers
10   "R0",  "R1",  "R2",  "R3",  "R4",  "R5",  "R6",  "R7",
11   "R8",  "R9",  "R10", "R11", "R12", "R13", "R14", "R15",
12   "R16", "R17", "R18", "R19", "R20", "R21", "R22", "R23",
13   "R24", "R25", "R26", "R27", "R28", "R29", "R30", "R31",
14 };
15
16 ArrayRef<const char *> MBLAZETargetInfo::getGCCRegNames() const {
17   return llvm::makeArrayRef(GCCRegNames);
18 }
19
20 void MBLAZETargetInfo::getTargetDefines(const LangOptions &Opts,
21                                         MacroBuilder &Builder) const {
22   // Define the __MBLAZE__ macro when building for this target
23   Builder.defineMacro("__MBLAZE__");
24 }
```

The information about registers are not always the same, depending on the implementation of the registers in the backend. The file that is been just edited are often identical through all the backends, so is possible to copy the configuration

from another file and make some tweaks to make it compile correctly for another target.

Now the header file `LLVM_13_MBLAZE/clang/lib/Basic/Targets/MBLAZE.h` need to be defined as follows:

```
#ifndef LLVM_CLANG_LIB_BASIC_TARGETS_MBLAZE_H
#define LLVM_CLANG_LIB_BASIC_TARGETS_MBLAZE_H

#include "clang/Basic/TargetInfo.h"
#include "clang/Basic/TargetOptions.h"
#include "llvm/ADT/Triple.h"
#include "llvm/Support/Compiler.h"

namespace clang {
namespace targets {

class LLVM_LIBRARY_VISIBILITY MBLAZETargetInfo : public TargetInfo {
  static const char *const GCCRegNames[];

public:
  MBLAZETargetInfo(const llvm::Triple &Triple, const TargetOptions &)
    : TargetInfo(Triple) {
    // Description string has to be kept in sync with backend string at
    // llvm/lib/Target/MBLAZE/MBLAZETargetMachine.cpp
    resetDataLayout("e"
                    // ELF name mangling
                    "-m:e"
                    // 32-bit pointers, 32-bit aligned
                    "-p:32:32"
                    // 64-bit integers, 64-bit aligned
                    "-i64:64"
                    // 32-bit native integer width i.e register are 32-bit
                    "-n32"
                    // 128-bit natural stack alignment
                    "-S128"
    );
    SuitableAlign = 128;
    WCharType = SignedInt;
    WIntType = UnsignedInt;
    IntPtrType = SignedInt;
    PtrDiffType = SignedInt;
    SizeType = UnsignedInt;
  }

  void getTargetDefines(const LangOptions &Opts,
```

```
                        MacroBuilder &Builder) const override;

  ArrayRef<const char *> getGCCRegNames() const override;

  BuiltinVaListKind getBuiltinVaListKind() const override {
    return TargetInfo::VoidPtrBuiltinVaList;
  }

...
};

} // namespace targets
} // namespace clang

#endif // LLVM_CLANG_LIB_BASIC_TARGETS_MBLAZE_H
```

After having finished the header implementation then there are some other lines of code that need to be added in the `clang/lib/Driver/CMakeLists.txt` file:

```
ToolChains/MBLAZE.cpp
```

The last thing to add two more files in the following path `clang/lib/Driver/ToolChains/`:

1. `MBLAZE.cpp`
2. `MBLAZE.h`

The following code snippets is a stub of the most important function, if the reader is looking for more information about clang and how it works he has to read this documentation: `https://clang.llvm.org/docs/index.html`

`MBLAZE.cpp`'s code:

```
#include "MBLAZE.h"
#include "CommonArgs.h"
#include "clang/Driver/Compilation.h"
#include "clang/Driver/Driver.h"
#include "clang/Driver/Options.h"
#include "llvm/Option/ArgList.h"

using namespace clang::driver;
using namespace clang::driver::toolchains;
using namespace clang;
using namespace llvm::opt;

MBLAZEToolChain::MBLAZEToolChain(const Driver &D, const llvm::Triple &Triple,
                                 const ArgList &Args)
```

```
    : ToolChain(D, Triple, Args) {
  // ProgramPaths are found via 'PATH' environment variable.
}

bool MBLAZEToolChain::isPICDefault() const { return true; }

bool MBLAZEToolChain::isPIEDefault() const { return false; }

bool MBLAZEToolChain::isPICDefaultForced() const { return true; }

bool MBLAZEToolChain::SupportsProfiling() const { return false; }

bool MBLAZEToolChain::hasBlocksRuntime() const { return false; }
```

and this is `MBLAZE.h`'s code:

```
#ifndef LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_MBLAZE_H
#define LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_MBLAZE_H

#include "Gnu.h"
#include "clang/Driver/ToolChain.h"

namespace clang {
namespace driver {
namespace toolchains {

class LLVM_LIBRARY_VISIBILITY MBLAZEToolChain : public ToolChain {
public:
  MBLAZEToolChain(const Driver &D, const llvm::Triple &Triple,
                  const llvm::opt::ArgList &Args);

  bool isPICDefault() const override;
  bool isPIEDefault() const override;
  bool isPICDefaultForced() const override;
  bool SupportsProfiling() const override;
  bool hasBlocksRuntime() const override;
 ...
} // end namespace toolchains
} // end namespace driver
} // end namespace clang

#endif // LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_MBLAZE_H
```

5.4 ELF Definition

The Executable and Linkable Format (ELF) is a common standard file format for executables, object code, shared libraries and core dumps. First published in the System V Application Binary Interface specification, and later in the Tool Interface Standard, it was quickly accepted among different vendors of Unixsystems. In 1999 it was chosen as the standard binary file format for Unix and Unix-like systems on x86 by the x86open project.

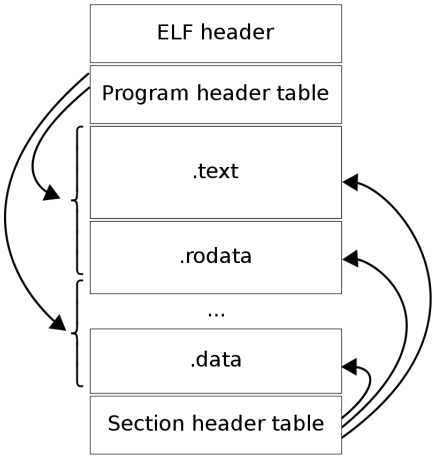


Figure 31: ELF View

To support ELF compilation *relocations* for the backend need to be created: *"Conceptually, a relocation describes how to calculate the address, for example, as an offset to a known address. If we resolve the relocations into addresses, like the linker and dynamic loader do, then we can execute the object code. Running the static compiler to compile IR code into an object file in memory, performing a link step on the in-memory object file, and then running the code gives us a JIT compiler. The JIT implementation in the LLVM core libraries is based on this idea"*. From *Learn LLVM 12, Kai Nacke, page 261, first paragraph* [3]

Opening `llvm/include/llvm/BinaryFormat/ELFRelocs/` the `MBLAZE.def` need to be created as follows:

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

ELF_RELOC(R_MBLAZE_NONE,                0)
ELF_RELOC(R_MBLAZE_32,                  2)
ELF_RELOC(R_MBLAZE_HI16,                5)
ELF_RELOC(R_MBLAZE_LO16,                6)
```

```
ELF_RELOC(R_MBLAZE_GPREL16,             7)
ELF_RELOC(R_MBLAZE_LITERAL,             8)
ELF_RELOC(R_MBLAZE_GOT16,               9)
ELF_RELOC(R_MBLAZE_PC16,               10)
ELF_RELOC(R_MBLAZE_CALL16,             11)
ELF_RELOC(R_MBLAZE_GPREL32,            12)
ELF_RELOC(R_MBLAZE_PC24,               13)
ELF_RELOC(R_MBLAZE_GOT_HI16,           22)
ELF_RELOC(R_MBLAZE_GOT_LO16,           23)
ELF_RELOC(R_MBLAZE_RELGOT,             36)
ELF_RELOC(R_MBLAZE_TLS_GD,             42)
ELF_RELOC(R_MBLAZE_TLS_LDM,            43)
ELF_RELOC(R_MBLAZE_TLS_DTP_HI16,       44)
ELF_RELOC(R_MBLAZE_TLS_DTP_LO16,       45)
ELF_RELOC(R_MBLAZE_TLS_GOTTPREL,       46)
ELF_RELOC(R_MBLAZE_TLS_TPREL32,        47)
ELF_RELOC(R_MBLAZE_TLS_TP_HI16,        49)
ELF_RELOC(R_MBLAZE_TLS_TP_LO16,        50)
ELF_RELOC(R_MBLAZE_GLOB_DAT,           51)
ELF_RELOC(R_MBLAZE_JUMP_SLOT,          127)
```

To handle correctly ELF there are some more information that need to be added inside `llvm/include/llvm/BinaryFormat/ELF.h`. Close to 130th line of code, there is a list of machine architectures. There are some other informations that need to be inserted the information about the target: `EM_MBLAZE = 999 //MBLAZE`. This is a list of value that define the architecture; the general advice is to give a great value, so if it happens to pull from main stream branch is almost impossible to have conflict in the project.

Always in the same file ELF Relocation type needs to defined:

```
567.    // ELF Relocation types for MBLAZE
568.    enum {
569.        #include "ELFRelocs/MBLAZE.def"
570.    };
```

There are some more information about the target that are mandatory to add in `llvm/include/llvm/Object/ELFObjectFile.h`:

```
    case ELF::EM_MBLAZE:
      return "elf32-mblaze";
```

to the `getFileFormatName()` method. Also in the method `getArch()` is necessary to add the following lines:

```
case ELF::EM_MBLAZE:
    return Triple::mblaze;
```

Last the relocations definition that is defined here `llvm/lib/Object/ELF.cpp` has to be used in the `getELFRelocationTypeName()` method as follows:

```
case ELF::EM_MBLAZE:
switch (Type) {
    #include "llvm/BinaryFormat/ELFRelocs/MBLAZE.def"
    default:
        break;
    }
break;
```

5.5 MBLAZE

This is the core of the target description files: inside of it are included all other description files. Every time that will be added new `.td` files they are going to be included inside of it. So at first `MBLAZE.td` looks like:

```
include "llvm/Target/Target.td"
...
include "MBLAZERegisterInfo.td"
include "MBLAZEInstrInfo.td"
include "MBLAZECallingConv.td"
```

The order in which they are included is really important; if the order changes there will have some problem about the definition of instructions and classes. Now the file need to be completed with a definition of *InstrInfo* and the information of the supported processor:

```
def MBLAZEInstrInfo : InstrInfo;

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

def : Proc<"generic", []>;

def MBLAZE : Target {
  let InstructionSet = MBLAZEInstrInfo;
}
```

If there are have any subtarget features this is the first place in which they are defined. This is an example of a subtarget feature taken from ARC:

```
def FeatureNORM
    : SubtargetFeature<"norm", "Xnorm", "true",
                       "Enable support for norm instruction.">;
```

This is done in the same way for every subtarget features that the backend could have.

5.6 Instructions Formats

From MicroBlaze Documentation[1] the user has access to information about the two main formats of instruction and implement it. This is the type TA, TB, TAR, TBR:

```
class TA<bits<6> op, bits<11> flags, dag outs, dag ins, string asmstr,
         list<dag> pattern> :
         InstMBLAZE<4, outs, ins, asmstr, pattern>
{
  bits<5> rd;
  bits<5> ra;
  bits<5> rb;

  let Inst{0-5} = op;
  let Inst{6-10}  = rd;
  let Inst{11-15} = ra;
  let Inst{16-20} = rb;
  let Inst{21-31} = flags;
}

class TAR<bits<6> op, bits<11> flags, dag outs, dag ins, string asmstr,
          list<dag> pattern> :
          TA<op, flags, outs, ins, asmstr, pattern>
{
  bits<5> rrd;
  bits<5> rrb;
  bits<5> rra;

  // let Form = FRRRR;

  let rd = rrd;
  let ra = rra;
  let rb = rrb;
}


class TB<bits<6> op, dag outs, dag ins,string asmstr, list<dag> pattern>
  : InstMBLAZE<4, outs, ins, asmstr, pattern>
{
  bits<5>  rd;
  bits<5>  ra;
  bits<16> imm16;

  let Inst{0-5} = op;
  let Inst{6-10}  = rd;
  let Inst{11-15} = ra;
```

```
  let Inst{16-31} = imm16;
}

class TBR<bits<6> op, dag outs, dag ins, string asmstr, list<dag> pattern>
  : TB<op, outs, ins, asmstr, pattern> {
  bits<5>  rrd;
  bits<16> rimm16;
  bits<5>  rra;

//  let Form = FRIR;

  let rd = rrd;
  let ra = rra;
  let imm16 = rimm16;
}
```

All implemented inside the `MBLAZEInstrFormats.td`.

5.7 Instruction Info

This are the subgroup that was defined starting from the main enconding type. The following are the arithmetic subgroup that are going to generare the ADD and SUB instructions:

```
class Arith<bits<6> op, bits<11> flags, string instr_asm> :
            TA<op, flags, (outs GPR32:$dst), (ins GPR32:$b, GPR32:$c),
               !strconcat(instr_asm, "   $dst, $b, $c"),
               []>;

class ArithI<bits<6> op, string instr_asm> :
             TB<op, (outs GPR32:$dst), (ins GPR32:$b, i32imm:$c),
                !strconcat(instr_asm, "   $dst, $b, $c"),
                []>;

class ArithR<bits<6> op, bits<11> flags, string instr_asm> :
            TAR<op, flags, (outs GPR32:$dst), (ins GPR32:$b, GPR32:$c),
                !strconcat(instr_asm, "   $dst, $c, $b"),
                []>;

class ArithRI<bits<6> op, string instr_asm> :
             TBR<op, (outs GPR32:$dst), (ins i32imm:$b, GPR32:$c),
                 !strconcat(instr_asm, "   $dst, $c, $b"),
                 []>;
```

This are the actual instruction derived from the the previous subgroups:

```
def ADD_rrr : Arith<0x00, 0x000, "add">;
...
```

```
def ADD_rrlimm : ArithI<0x08, "addi">;

def SUB_rrr : ArithR<0x05, 0x000, "rsubk">;
def SUB_rrlimm : ArithRI<0x09, "rsubi">;
...
```

Another subgroup is the logic that create the instructions related with the logical operations (AND, OR, XOR):

```
class Logic<bits<6> op, bits<11> flags, string instr_asm> :
            TA<op, flags, (outs GPR32:$dst), (ins GPR32:$b, GPR32:$c),
               !strconcat(instr_asm, "   $dst, $b, $c"),
               []>;

class LogicI<bits<6> op, string instr_asm> :
             TB<op, (outs GPR32:$dst), (ins GPR32:$b, uimm16:$c),
                !strconcat(instr_asm, "   $dst, $b, $c"),
                []>;
 ...

def OR_rrr : Logic<0x20, 0x000, "or">;
def OR_rrlimm : LogicI<0x28, "ori">;
...

def AND_rrr : Logic<0x21, 0x000, "and">;
def AND_rrlimm : LogicI<0x29, "andi   ">;
...

def XOR_rrr : Logic<0x22, 0x000, "xor">;
def XOR_rrlimm : LogicI<0x2A, "xori">;
...
```

Then there is the subgroup of branch conditional instructions and its instruction definition:

```
class BranchC<bits<6> op, bits<5> br, bits<11> flags, string instr_asm> :
             TA<op, flags, (outs),
                (ins GPR32:$a, GPR32:$b),
                !strconcat(instr_asm, "   $a, $b"),
                []> {
 let rd = br;
}
...
  def BGE    :  BranchC<0x27, 0x05, 0x000, "bge    ">;
  def BEQ_S : BranchC<0x27, 0x00, 0x000, "beq_s">;
  def BNE_S : BranchC<0x27, 0x01, 0x000, "bne_s">;
  def BGT_S : BranchC<0x27, 0x04, 0x000, "bgt_s">;
```

```
  def BGE_S : BranchC<0x27, 0x05, 0x000, "bge_s">;
  def BLT_S : BranchC<0x27, 0x02, 0x000, "blt_s">;
  def BLE_S : BranchC<0x27, 0x03, 0x000, "ble_s">;
```

And finally there are the load and store subgroups with instructions definitions:

```
class LoadM<bits<6> op, bits<11> flags, string instr_asm> :
            TA<op, flags, (outs GPR32:$dst), (ins MEMrs9:$addr),
               !strconcat(instr_asm, "   $dst, $addr"),
               []>;

class LoadMI<bits<6> op, string instr_asm> :
             TB<op, (outs GPR32:$dst), (ins MEMii:$addr),
                !strconcat(instr_asm, "   $dst, $addr"),
                []>;

 ...

 class StoreM<bits<6> op, bits<11> flags, string instr_asm> :
             TA<op, flags, (outs), (ins GPR32:$dst, memrr:$addr),
                !strconcat(instr_asm, "   $dst, $addr"),
                []>;


class StoreMI<bits<6> op, string instr_asm> :
              TB<op, (outs), (ins GPR32:$dst, memri:$addr),
                 !strconcat(instr_asm, "   $dst, $addr"),
                 []>;

 ...
// load 32-bit
def LD : LoadM<0x32, 0x000, "lw">;
...
def LD_limm : LoadMI<0x3A, "lwi">;

// load 16-bit
def LDH : LoadM<0x31, 0x000, "lhu">;
...
def LDH_limm : LoadMI<0x39, "lhui">;

// load 8-bit
def LDB : LoadM<0x30, 0x000, "lbu">;
...
def LDB_limm : LoadMI<0x38, "lbui">;
 ...

```

```
41  // store 32-bit
42  def ST_rs9 : StoreM<0x36, 0x000, "sw">;
43  def ST_limm : StoreMI<0x3E, "swi">;
44
45  // store 16-bit
46  def STH_rs9 : StoreM<0x36, 0x000, "sw">;
47  def STH_limm : StoreMI<0x3E, "swi">;
48
49  // store 8-bit
50  def STB_rs9 : StoreM<0x36, 0x000, "sw">;
51  def STB_limm : StoreMI<0x3E, "swi">;
```

5.8 Registers

Defining register meanings define a general class in which there is the hardware encoding of the registers

```
class MBLAZEReg<string n, list<string> altNames> : Register<n, altNames> {
  field bits<5> HwEncoding;
  let Namespace = "MBLAZE";
}
```

and then a sub-class that will help us defining the effective registers:

```
class Core<int num, string n, list<string>altNames=[]> : MBLAZEReg<n, altNames> {
  let HWEncoding = num;
}

def R0 : Core< 0, "%r0">, DwarfRegNum<[0]>;
def SP : Core<1, "%sp", ["%r1"]>, DwarfRegNum<[1]>;
def R2 : Core< 2, "%r2">, DwarfRegNum<[2]>;
def R3 : Core< 3, "%r3">, DwarfRegNum<[3]>;
...
```

5.9 Calling Convention

Information about this part are written in the architecture documentations [1], so this is the *MicroBlaze* implementation:

```
def RetCC_MBLAZE : CallingConv<[
  // i32 are returned in registers R0, R4, R2, R3
  CCIfType<[i32, i64], CCAssignToReg<[R3, R4]>>,

  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<[i64], CCAssignToStack<8, 4>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;
```

This definition define in which registers are stored the returning values, while the next code's snippet defines which registers are going to be used to handle arguments:

```
def CC_MBLAZE : CallingConv<[
  // Promote i8/i16 arguments to i32.
  CCIfType<[i8, i16], CCPromoteToType<i32>>,

  // The first 8 integer arguments are passed in integer registers.
  CCIfType<[i32, i64], CCAssignToReg<[R0, R4, R2, R3, R4, R5, R6, R7]>>,

  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<[i64], CCAssignToStack<8, 4>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;

def CSR_MBLAZE : CalleeSavedRegs<(add (sequence "R%u", 2, 13), GP, FP)>;
```

5.10 TargetMachine and Subtarget

Every backend needs at least one implementation of a *TargetMachine* class and `MBLAZETargetMachine` class holds a lot of information that are used during code generation phase and it is also useful for the subtarget class. In this sense can be a good idea to see the implementation of this files:

- `TargetTargetMachine.cpp/h`
- `TargetSubtarget.cpp/h`

and stub all the method that are required. The general structure of targetmachine file have a main class:

```
MBLAZETargetMachine::MBLAZETargetMachine(const Target &T, const Triple &TT,
                                         StringRef CPU, StringRef FS,
                                         const TargetOptions &Options,
                                         Optional<Reloc::Model> RM,
                                         Optional<CodeModel::Model> CM,
                                         CodeGenOpt::Level OL, bool JIT)
    : LLVMTargetMachine(T, computeDataLayout(TT), TT, CPU, FS, Options,
                        RM.getValueOr(Reloc::Static),
                        getEffectiveCodeModel(CM, CodeModel::Small), OL),
      TLOF(std::make_unique<TargetLoweringObjectFileELF>()),
      Subtarget(TT, std::string(CPU), std::string(FS), *this) {
  initAsmInfo();
}
```

that is is defined as follows in its header file:

```
namespace llvm {

class TargetPassConfig;

class MBLAZETargetMachine : public LLVMTargetMachine {
  std::unique_ptr<TargetLoweringObjectFile> TLOF;
  MBLAZESubtarget Subtarget;

public:
  MBLAZETargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                      StringRef FS, const TargetOptions &Options,
                      Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                      CodeGenOpt::Level OL, bool JIT);
                      ~MBLAZETargetMachine() override;

  const MBLAZESubtarget *getSubtargetImpl() const { return &Subtarget; }
  const MBLAZESubtarget *getSubtargetImpl(const Function &) const override {
    return &Subtarget;
```

```
                    }
            ...
            };
  }
```

For what concerns the `MBLAZESubtarget` file, it holds a lot of information about the target and eventually its subtargets. In `MBLAZESubtarget.cpp` can defined a generic function that generate the subtarget information. Even if the target doesn't have any feature or subtarget, is mandatory to have it.

```
#include "MBLAZEGenSubtargetInfo.inc"

MBLAZESubtarget::MBLAZESubtarget(const Triple &TT, const std::string &CPU,
                                 const std::string &FS, const TargetMachine &TM)
    : MBLAZEGenSubtargetInfo(TT, CPU, /*TuneCPU=*/CPU, FS), FrameLowering(*this),
      TLInfo(TM, *this) {}
```

As is shown in the code snippet is important to add the file that *tablegen* generates at the beginning of this file. These files that have been implemented are an extension of what *table gen* produce, starting from target description files. For what concern the its header file is really important to stub all the virtual methods that llvm expects. This is the implementation:

```
public:
  /// This constructor initializes the data members to match that
  /// of the specified triple.
  MBLAZESubtarget(const Triple &TT, const std::string &CPU, const std::string &FS,
                  const TargetMachine &TM);

  /// Parses features string setting specified subtarget options.
  /// Definition of function is auto generated by tblgen.
  void ParseSubtargetFeatures(StringRef CPU, StringRef TuneCPU, StringRef FS);

  const MBLAZEInstrInfo *getInstrInfo() const override { return &InstrInfo; }
  const MBLAZEFrameLowering *getFrameLowering() const override {
    return &FrameLowering;
  }

  const MBLAZETargetLowering *getTargetLowering() const override {
    return &TLInfo;
  }

  const MBLAZERegisterInfo *getRegisterInfo() const override {
    return &InstrInfo.getRegisterInfo();
  }

  const SelectionDAGTargetInfo *getSelectionDAGInfo() const override {
```

```
25        return &TSInfo;
26      }
```

5.11 Selection DAG implementation

The selection DAG is implemented by the developer in `<Target>DAGtoDAGIsel` class, inside the file that has the same name. In this very first implementation is enough to override the `Select()` function and forward it to `SelectCode` function; also because most of the implementation of the selection DAG is done by *table gen*. This is an example of implementation:

```
1     class MBLAZEDAGToDAGISel : public SelectionDAGISel {
2     public:
3         MBLAZEDAGToDAGISel(MBLAZETargetMachine &TM, CodeGenOpt::Level OptLevel)
4             : SelectionDAGISel(TM, OptLevel) {}
5
6       void Select(SDNode *N) override;
7       bool SelectAddrModeImm(SDValue Addr, SDValue &Base, SDValue &Offset);
8
9
10        StringRef getPassName() const override {
11    return "MBLAZE DAG->DAG Pattern Instruction Selection";
12  }
13
14  #include "MBLAZEGenDAGISel.inc"
15  };
16
17  ...
18
19  FunctionPass *llvm::createMBLAZEISelDag(MBLAZETargetMachine &TM,
20                                          CodeGenOpt::Level OptLevel) {
21  return new MBLAZEDAGToDAGISel(TM, OptLevel);
22  }
```

5.12 Configure Target Lowering

Supposing that the DAG phase went right it is mandatory to define the lowering of the instructions chosen in the DAG. Without this information the DAG matching phase does not works at all. Inside `<Target>ISelLowering.h` there is the enumeration of node types:

```
  namespace MBLAZEISD {

  enum NodeType : unsigned {
  // Start the numbering where the builtin ops and target ops leave off.
  FIRST_NUMBER = ISD::BUILTIN_OP_END,

  // Branch and link (call)
  BL,

  // Jump and link (indirect call)
  JL,

...

  // Return
  RET,

  IRET
};

} // end namespace MBLAZEISD
```

It can be expand further based on necessity. The instructions need to be lowered and there are some functions that make this possible:

```
 class MBLAZETargetLowering : public TargetLowering {
public:
  explicit MBLAZETargetLowering(const TargetMachine &TM,
                                const MBLAZESubtarget &Subtarget);
                                ...
SDValue PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI) const override;
 SDValue LowerReturn(SDValue Chain, CallingConv::ID CallConv, bool isVarArg,
                      const SmallVectorImpl<ISD::OutputArg> &Outs,
                      const SmallVectorImpl<SDValue> &OutVals, const SDLoc &dl,
                      SelectionDAG &DAG) const override;
```

In this code snippets is shown the implementation of the return call. It is a good idea to take a closer look also at the implementation in the following link: `https://github.com/MaxBubblegum47/LLVM_13_MBLAZE/blob/main/llvm/lib/Target/MBLAZE/MBLAZEISelLowering.cpp` The core idea is that if there is a node, in particular: return, jump and link, branch operations, has be to defined in the

header and check if there is the right lowering. All the arguments that the `LowerReturn` takes as input are defined by its superclass, but the most interesting is `Outs` that is the vector containing the description of the return argument and `OutVals` that holds the return values of the DAG.

5.13 Assembler

In this part will completed the last step in order to generate assembly code. The `<Target>AsmPrinter` machine pass (associated with `<Target>AsmPrinter.cpp` file) is responsible for this task. Here the `MachineInstr` is further lowered into a `MCInst` to be fully compatible with the target architecture.

At first is possible to limit the implementation and just overriding `emitInstruction()`. If more operands need to be implemented, the developer need to look at other backends and implement method that may help him. This is the declaration:

```
 namespace {

class MBLAZEAsmPrinter : public AsmPrinter {
  MBLAZEMCInstLower MCInstLowering;

public:
  explicit MBLAZEAsmPrinter(TargetMachine &TM,
                            std::unique_ptr<MCStreamer> Streamer)
      : AsmPrinter(TM, std::move(Streamer)),
        MCInstLowering(&OutContext, *this) {}

  StringRef getPassName() const override { return "MBLAZE Assembly Printer"; }
  void emitInstruction(const MachineInstr *MI) override;
};
```

In the implementation of `<Target>AsmPrinter` method is possible to specify a behaviour to every operands that will encountered, but is also possible to only get the operand without knowing what it is and then handle it inside the `instlowering`:

```
void MBLAZEAsmPrinter::emitInstruction(const MachineInstr *MI) {
  SmallString<128> Str;
  raw_svector_ostream O(Str);

  switch (MI->getOpcode()) {
  case MBLAZE::DBG_VALUE:
    llvm_unreachable("Should be handled target independently");
    break;
  }

  MCInst TmpInst;
  MCInstLowering.Lower(MI, TmpInst);
  EmitToStreamer(*OutStreamer, TmpInst);
}
```

The aim of `<Target>MCInstLower` class is to handle the various operand type and it has not superclass. The full implementation can be seen in the source code at `https://github.com/MaxBubblegum47/LLVM_13_MBLAZE`

In the end this method has to be included. It is used during the initialization phase:

```
// Force static initialization.
extern "C" LLVM_EXTERNAL_VISIBILITY void LLVMInitializeMBLAZEAsmPrinter() {
  RegisterAsmPrinter<MBLAZEAsmPrinter> X(getTheMBLAZETarget());
}
```

5.14 Machine Code

The `MC` layer is responsible of emitting machine code in two possible differents forms:

- Textual
- Binary

This part takes place inside the `MCTargetDesc` folder and in `<Target>MCTargetDesc.cpp` file. Inside of this file have to be defined several classes:

- `<Target>MCInstrInfo`
- `<Target>MCRegisterInfo`
- `<Target>MCSubtargetInfo`
- `<Target>MCAsmInfo`
- `<Target>MCInstPrinter`

Each of this class holds information about some aspects of out backend; they can be implemented with few lines of code as follows:

```
1  static MCInstrInfo *createMBLAZEMCInstrInfo() {
2    auto *X = new MCInstrInfo();
3    InitMBLAZEMCInstrInfo(X);
4    return X;
5  }
6
7  ...
8
9  extern "C" LLVM_EXTERNAL_VISIBILITY void LLVMInitializeMBLAZETargetMC() {
10 ...
11   // Register the MC instruction info.
12   TargetRegistry::RegisterMCInstrInfo(TheMBLAZETarget, createMBLAZEMCInstrInfo);
```

Before declaring the the class and then calling it inside of `LLVMInitializeMBLAZETargetMC()`. This operation is called registration. Not every class that in the backend need to be registered. For example: if there is no need to have any textual printing of the assembler's output, is possible to be omitted `MCInstPrinter`.

Inside the header file have to be included some of the auto generated file that *table gen* have created:

```
1  #include "llvm/Support/DataTypes.h"
2
3  namespace llvm {
4
5  class Target;
6
```

```
} // end namespace llvm

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

// Defines symbolic names for the MBLAZE instructions.
#define GET_INSTRINFO_ENUM
#include "MBLAZEGenInstrInfo.inc"

#define GET_SUBTARGETINFO_ENUM
#include "MBLAZEGenSubtargetInfo.inc"
```

5.15 Creating the CMakeLists

To build the target as experimental (this is done for target in developing) the following information need to be added in the `llvm/CMakeLists.txt` file:

```
set(LLVM_EXPERIMENTAL_TARGETS_TO_BUILD "MBLAZE"  ... )
```

Is also necessary to provide a `llvm/lib/Target/MBLAZE/CMakeLists.txt` file in order to build the target. The file has the following configuration:

```
add_llvm_component_group(MBLAZE)

set(LLVM_TARGET_DEFINITIONS MBLAZE.td)

tablegen(LLVM MBLAZEGenAsmWriter.inc -gen-asm-writer)
tablegen(LLVM MBLAZEGenCallingConv.inc -gen-callingconv)
tablegen(LLVM MBLAZEGenDAGISel.inc -gen-dag-isel)
tablegen(LLVM MBLAZEGenDisassemblerTables.inc -gen-disassembler)
tablegen(LLVM MBLAZEGenInstrInfo.inc -gen-instr-info)
tablegen(LLVM MBLAZEGenRegisterInfo.inc -gen-register-info)
tablegen(LLVM MBLAZEGenSubtargetInfo.inc -gen-subtarget)

add_public_tablegen_target(MBLAZECommonTableGen)

add_llvm_target(MBLAZECodeGen
  MBLAZEAsmPrinter.cpp
  MBLAZEBranchFinalize.cpp
  MBLAZEExpandPseudos.cpp
  MBLAZEFrameLowering.cpp
  MBLAZEInstrInfo.cpp
  MBLAZEISelDAGToDAG.cpp
  MBLAZEISelLowering.cpp
  MBLAZEMachineFunctionInfo.cpp
  MBLAZEMCInstLower.cpp
  MBLAZEOptAddrMode.cpp
  MBLAZERegisterInfo.cpp
  MBLAZESubtarget.cpp
  MBLAZETargetMachine.cpp

  LINK_COMPONENTS
  Analysis
  AsmPrinter
  CodeGen
  Core
  MC
  SelectionDAG
  Support
```

```
38    Target
39    TransformUtils
40    MBLAZEDesc
41    MBLAZEInfo
42
43    ADD_TO_COMPONENT
44    MBLAZE
45  )
46
47  add_subdirectory(Disassembler)
48  add_subdirectory(MCTargetDesc)
49  add_subdirectory(TargetInfo)
```

In this CMakeLists.txt there are more *cpp* files than the ons that have been illustrated in this chapter; they are standard files that can be seen at that link [28]. They are standard files and they make use of templates that are common through all the backends. In each subdirectories there is a CMakeLists similar to this, but with fewer *cpp* and without *tablegen* invocation:

```
1   add_llvm_component_library(LLVMMBLAZEInfo
2     MBLAZETargetInfo.cpp
3
4     LINK_COMPONENTS
5     Support
6
7     ADD_TO_COMPONENT
8     MBLAZE
9     )
```

This is the `CMakeLists.txt` that is present inside the `TargetInfo` folder.

5.16 Build Configuration

To build LLVM is mandatory to have a `build` folder in the following path: `LLVM\_13\_MBLAZE/llvm/`and after have positioned the terminal inside of it two differents command need to be launched:

1. `cmake -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="MBLAZE" ..`
2. `make`

The compilation of LLVM is not fast, but the are some tweak that can improve substantially the build-time: at first is possible to remove all the backends from the list of the default build editing `llvm-project/llvm/CMakeLists.txt`

```
# List of all targets to be built by default:
set(LLVM_ALL_TARGETS
  AArch64
  AMDGPU
  ...
  XCore
  )
```

Then another important thing is to change the build type. If the user do not specifies it, the build will be in *Debug* mode by default. Building in *Debug* mode takes more time than other type of build, like for example *Release*. To enable this type of build the user need to add the option `-DLLVM_BUILD_TYPE=Release`. If *Release* mode is enable the user takes advantage of another interesting option that use all cpu-cores. The only thing to is to add the following line of code: `-j $(nproc)` along with *make*. Last there is the opportunity to use a different linker to gain better performance, as is reported in this link. So the final commands that will make possible more quickly are:

1. `cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_USE_LINKER=gold -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="MBLAZE" ..`
2. `make -j $(nproc)`

6 Evaluations

6.1 Achievements

In this section will be shown with a matrix all the capabilities and instructions encoding that have been achieved and what is missing, considering what is defined inside the MicroBlaze official documentation [1]:

Implemented	Not Implemented
add	andn
addi	andni
and	beqi
andi	bgei
beq	bgti
bge	blei
bgt	blti
ble	bnei
blt	br
bne	bri
lbu	brk
lbui	brki
lhu	bs
lhui	bsi
lw c	lz
lwi	cmp
or	fadd
ori	frsub
rsub	fmul
rsubi	fdiv
rtsd	fcmp
sb	fit
sbi	fint
sh	fsqrt
shi	get
sw	getd
swi	idiv
xor	imm
xori	lwx

Implemented	Not Implemented
R0	mbar
R1	mfs
R2	msrclr
R3	msrset
R4	mts
R5	mts
R6	mul
R7	mulh
R8	mulhu
R9	mulhsu
R10	muli
R11	pcmpbf
R12	pcmpeq
R13	put
R14	putd
R15	rtbd
R16	rtid
R17	rted
R18	sext16
R19	sext18
R20	sra
R21	src
R22	srl
R23	swapb
R24	swaph
R25	swx
R26	wdc
R27	wic
R28	MSR
R29	EAR
R30	BTR
R31	FSR
SP	EDR
GP	SLR
Calling Convention	SHR
	PID
	ZPR
	TLBLO
	TLBHI
	TLBX
	TLBSX
	PVR

The word written in UPPER CASE are registers, the others are instructions.

6.2 Benchmarks

The implementation of MicroBlaze backend it is fairly complete in term of features and using Clang frontend to produce an intermediate representation with the command: `clang -s -emit-llvm gemm.c`, is possible to generate assembly code using the next command: `llc -mtriple=mblaze`. All basic capabilities are present:

1. Arithmetical Operations
2. Logic Operations
3. Memory Operations
4. Branch Operation

All this operations include operate with and without immediate values.

To benchmark the build it was used PolyBench. It is a collection of benchmarks in which was chosen as main benchmark the `gemm` test from the *Blas Routines* group. This is the input of the problem:

1. α, β: scalars
2. `A`: $P \times Q$ matrix
3. `B`: $Q \times R$ matrix
4. `C`: $P \times R$ matrix

and this is the expected output:

1. `Cout`: $P \times R$ array, where `Cout` $= \alpha AB + \beta C$

This is the implementation in C language:

```
1  * POLYBENCH/GPU-OPENMP
2   *
3   * This file is a part of the Polybench/GPU-OpenMP suite
4   *
5   * Contact:
6   * William Killian <killian@udel.edu>
7   *
8   * Copyright 2013, The University of Delaware
9   */
10 #include <stdio.h>
11 #include <unistd.h>
12 #include <string.h>
13 #include <math.h>
14
15 /* Include polybench common header. */
16 #include <polybench.h>
```

```c

/* Include benchmark-specific header. */
/* Default data type is double, default size is 4000. */
#include "gemm.h"


/* Array initialization. */
static
void init_array(int ni, int nj, int nk,
                DATA_TYPE *alpha,
                DATA_TYPE *beta,
                DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj),
                DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk),
                DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj))
{
  int i, j;

  *alpha = 32412;
  *beta = 2123;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++)
      C[i][j] = ((DATA_TYPE) i*j) / ni;
  for (i = 0; i < ni; i++)
    for (j = 0; j < nk; j++)
      A[i][j] = ((DATA_TYPE) i*j) / ni;
  for (i = 0; i < nk; i++)
    for (j = 0; j < nj; j++)
      B[i][j] = ((DATA_TYPE) i*j) / ni;
}


/* DCE code. Must scan the entire live-out data.
   Can be used also to check the correctness of the output. */
static
void print_array(int ni, int nj,
                 DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj))
{
  int i, j;

  for (i = 0; i < ni; i++)
    for (j = 0; j < nj; j++) {
        fprintf (stderr, DATA_PRINTF_MODIFIER, C[i][j]);
        if ((i * ni + j) % 20 == 0) fprintf (stderr, "\n");
    }
  fprintf (stderr, "\n");
}
```

```
63
64
65  /* Main computational kernel. The whole function will be timed,
66     including the call and return. */
67  static
68  void kernel_gemm(int ni, int nj, int nk,
69                   DATA_TYPE alpha,
70                   DATA_TYPE beta,
71                   DATA_TYPE POLYBENCH_2D(C,NI,NJ,ni,nj),
72                   DATA_TYPE POLYBENCH_2D(A,NI,NK,ni,nk),
73                   DATA_TYPE POLYBENCH_2D(B,NK,NJ,nk,nj))
74  {
75    int i, j, k;
76    #pragma scop
77    #pragma omp parallel
78    {
79      /* C := alpha*A*B + beta*C */
80      #pragma omp for private (j, k)
81      for (i = 0; i < _PB_NI; i++)
82        for (j = 0; j < _PB_NJ; j++)
83          {
84            C[i][j] *= beta;
85            for (k = 0; k < _PB_NK; ++k)
86              C[i][j] += alpha * A[i][k] * B[k][j];
87          }
88    }
89    #pragma endscop
90  }
91
92
93  int main(int argc, char** argv)
94  {
95    /* Retrieve problem size. */
96    int ni = NI;
97    int nj = NJ;
98    int nk = NK;
99
100   /* Variable declaration/allocation. */
101   DATA_TYPE alpha;
102   DATA_TYPE beta;
103   POLYBENCH_2D_ARRAY_DECL(C,DATA_TYPE,NI,NJ,ni,nj);
104   POLYBENCH_2D_ARRAY_DECL(A,DATA_TYPE,NI,NK,ni,nk);
105   POLYBENCH_2D_ARRAY_DECL(B,DATA_TYPE,NK,NJ,nk,nj);
106
107   /* Initialize array(s). */
108   init_array (ni, nj, nk, &alpha, &beta,
```

```
109                 POLYBENCH_ARRAY(C),
110                 POLYBENCH_ARRAY(A),
111                 POLYBENCH_ARRAY(B));
112
113   /* Start timer. */
114   polybench_start_instruments;
115
116   /* Run kernel. */
117   kernel_gemm (ni, nj, nk,
118                alpha, beta,
119                POLYBENCH_ARRAY(C),
120                POLYBENCH_ARRAY(A),
121                POLYBENCH_ARRAY(B));
122
123   /* Stop and print timer. */
124   polybench_stop_instruments;
125   polybench_print_instruments;
126
127   /* Prevent dead-code elimination. All live-out data must be printed
128      by the function call in argument. */
129   polybench_prevent_dce(print_array(ni, nj,  POLYBENCH_ARRAY(C)));
130
131   /* Be clean. */
132   POLYBENCH_FREE_ARRAY(C);
133   POLYBENCH_FREE_ARRAY(A);
134   POLYBENCH_FREE_ARRAY(B);
135
136   return 0;
137 }
```

6.3 Problems and Solutions

The are still some instruction that need to be better implemented, like: `mul` and its variants, while others like `mov` need to be implemented in a proper way and the code need to be cleaned from unused parts and formatting. In particular inside `MBLAZEInstrInfo.td`. For what concerns the lowering part there are some modification that need to be done in order to be more compliant to the documentation, starting from the name of the instructions that are lowered. There are also a lot of special registers that need to be implemented like: `MSR`, `EAR`, `ESR`, ... and also some general purpose registers need to be better arranged in order to be complete the calling convention correctly. Another thing is that the version of the backend is the 13.0.1, but from the beginning of the project llvm developers are arrived at 15 version. So another task will be to implement the backend inside the new version and check if there are some structural changes inside llvm that may be invalidated some part of the backend.

7 Acknowledgements

Dedico i miei più sentiti ringraziamenti ai miei genitori, Lucia Americo e Giovanni Stigliano, i quali hanno fatto in modo che potessi sostenere gli studi universitari e mi hanno sempre spinto ai confini della mia curiosità. Sono stati il primo riflesso del mondo nei miei occhi, il primo assaggio di cosa sia la vita; nel loro essere umani, quindi imperfetti in tutto, mi hanno trasmesso un modo di vivere che funge da lente per interpretare una realtà che, mai come oggi, appare indecifrabile.

Dedico questi ringraziamenti anche ai miei amici: quelli di una vita, i Bomportesi, con cui ho trascorso avventure incredibili in un piccolo paese che confina con tutto il mondo, perché tale è la eterogeneità che ci contraddistingue. Questi ringraziamenti sono dedicati anche ai nuovi amici, quelli nati nel corso dell'università; sono stati il milgior esempio di impegno e passione nei confronti dell'Informatica, nonché massimo esempio di altruismo didattico nei confronti di chi risultava essere manchevole di molte nozioni importanti e necessitava dei giusti consigli.

In particolare, ci tengo a ringraziare Lorenzo Mazzella, perché questa tesi è nata da un suo invito ad approdare in un laboratorio di cui avevo sentito echeggiare il nome nei corridoi dell'edificio di Matematica, ma di cui sapevo ben poco. Conscio di quanto fossi incerto sul modo in cui avrei svolto il tirocinio, mi ha incosapevolmente traghettato verso quello che è poi si è rivelato essere l'oggetto della mia tesi di ricerca. Se non fosse stato per Lorenzo ora avrei meno capelli bianchi, ma sarei sicuramente più ignorante rispetto a molti aspetti dell'informatica, che altrimenti non avrei avuto mai modo di esplorare.

Dedico un grazie anche agli insegnanti che mi hanno accompagnato in questo percorso didattico. Le bocciature decise, gli elogi, le critiche costruttive, hanno forgiato la mia vita da studente positivamente. Voglio ringraziare in particolare Andrea Marongiu, che mi ha accompagnato, con la sua conoscenza dei compilatori e la sua pazienza, durante il progetto e successivamente come relatore della tesi. Il suo ottimismo mi ha accompagnato per un anno, vincendo infine tutti i momenti di sconforto che mi hanno attanagliato durante il progetto.

Questi ultimi ringraziamenti li dedico al giovane me che 4 anni fece l'incauta scelta di scegliere un percorso di studi nei confronti del quale fosse totalmente disarmato; ho compiuto questo percorso armato solo della mia curiosità ed è bastata. L'augurio più grande che faccio al me del domani è quello di non perdere mai la curiosità.

MaxBubblegum47

Inferno XXVI, 112-126, Dante

"O frati," dissi, "che per cento milia
perigli siete giunti a l'occidente,
a questa tanto picciola vigilia

d'i nostri sensi ch'è del rimanente
non vogliate negar l'esperïenza,
di retro al sol, del mondo sanza gente.
Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza".

Li miei compagni fec'io sì aguti,
con questa orazion picciola, al cammino,
che a pena poscia li avrei ritenuti;

e volta nostra poppa nel mattino,
de' remi facemmo ali al folle volo,
sempre acquistando dal lato mancino.

References

[1] Xilinx (2019), *MicroBlaze Processor Reference Guide*

[2] Christoph Erhardt (2009), *Design and Implementation of a TriCore Backend for the LLVM Compiler Framework*

[3] Kai Nacke (2021), *Learn LLVM 12*

[4] Bruno Cardoso Lopes and Rafel Auler (2014), *Getting Started with LLVM Core Libraries*

[5] Mayur Pandey and Suyog Sard (2015), *LLVM Cookbook*

[6] Chen Chung-Shu (2022), *Tutorial: Creating an LLVM Backend for the Cpu0 Architecture*

[7] Anton Korobeynikov (2009 LLVM Developers), *Tutorial: Building backend in 24 hours*

[8] J. Paquette - F. Hahn (2019 LLVM Developers), *Getting Started With LLVM: Basics*

[9] A. Bradbury (2019 LLVM Developers' Meeting), *Maturing an LLVM backend: Lessons learned from the RISC-V Target*

[10] Min-Yih Hsu (2021 LLVM Dev Mtg), *How to write a TableGen backend*

[11] M. Braun (2017 LLVM Developers' Meeting), *Welcome to the back-end: The LLVM machine representation*

[12] E. Christopher - J. Doerfert (2019 LLVM Developers' Meeting), *Introduction to LLVM*

[13] Elliot Vulgarise (2021), *Ma backend sur llvm*

[14] A. Bradbury (2018 LLVM Developers' Meeting), *LLVM backend development by example (RISC-V)*

[15] Johannes Doerfert, Argonne National Laboratory (2018), *A Compiler's View of OpenMP*

[16] James Beyer - Jeff Larkin(April 4-7, 2016), *Targeting GPUs with OpenMP 4.5 Device Directives*

[17] Oscar Hernandez - Tom Scogland - Colleen Bertoni - JaeHyuk Kwack - Johannes Doerfert - Vivek Kale (2020), *OpenMP 5.0/5.1 Tutorial*

[18] Gianluca Bellocchi - Alessandro Capotondi - Francesco Conti - Andrea Marongiu (2021), *RISC-V-based FPGA Overlay to Simplify Embedded Accelerator Deployment*

[19] Andreas Kurth - Pirmin Vogel - Alessandro Capotondi - Andrea Marongiu - Luca Benini (18 Dec 2017), *HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA*

[20] Vittoriano Muttillo - Giacomo Valente - Fabio Federici - Luigi Pomante - Marco Faccio - Carlo Tieri - Serenella Ferri (2016), *A design methodology for soft-core platforms on FPGA with SMP Linux, OpenMP support, and distributed hardware profiling system*

[21] John Demme (2016), *A Compiler Infrastructure for FPGA and ASIC Development*

[22] Marius Knaust (2019), *OpenMP to FPGA Offloading Prototype Using the Intel FPGA SDK for OpenCL*

[23] Olaf Krzikalla (European LLVM Conference Paris, 2013), *Performing Source-to-Source Transformations with Clang*

[24] Universidade Federal de Minas Gerais – Department of Computer Science – Programming Languages Laboratory, *INTRODUCTION TO LLVM*

[25] Keith D. Cooper and Linda Torczon (2012), *Engineering a Compiler*

[26] A. V. Aho - M.S. Lam - R. Sethi - J.D. Ullman - Carlo Brandolese - William Fornaciari (2019), *Compilatori Principi, tecniche e strumenti*

[27] Bjarne Stroustrup (2015), *C++*

[28] `https://llvm.org/docs/WritingAnLLVMBackend.html`

[29] `https://hpc-wiki.info/hpc/Building_LLVM/Clang_with_OpenMP_Offloading_to_NVIDIA_GPUs`

[30] `https://discourse.llvm.org/`

[31] `https://lists.llvm.org/pipermail/llvm-dev/2015-September/090630.html`

[32] `https://llvm.liuxfe.com/post/62627`

[33] `https://blog.csdn.net/fuhanghang/article/details/113888527`

[34] `https://en.wikipedia.org/wiki/Just-in-time_compilation`

[35] `https://github.com/cavazos-lab/PolyBench-ACC/blob/master/OpenMP/linear-algebra/kernels/gemm/gemm.c`

[36] `https://en.wikipedia.org/wiki/Field-programmable_gate_array`

[37] `https://en.wikipedia.org/wiki/Hardware_description_language`

[38] `https://pulp-platform.org/hero.html`