

How to Write an LLVM Backend #5: Arithmetic Instructions

Dec 2, 2020

We will now put into practice what we discussed in the previous post about the Instruction Selection process in an LLVM backend. We will do so by looking at the concrete implementation of arithmetic instructions, like add, subtract and multiply, for the RISCV backend.

NOTE: The code discussed in this post can be found [here](#).

NOTE: Posts in this series:

1. [Introduction](#)
2. [Getting Started](#)
3. [Setting Up a New Backend](#)
4. [Configuring the Build System](#)
5. [Instruction Selection](#)
6. [Arithmetic Instructions](#)

RISCV's arithmetic instructions are relatively simple, so we only need to configure the target instruction selection and legalization phases from the Instruction Selection process. This requires three main code changes. First, we will describe the instruction set to LLVM using TableGen. Second, we will inform LLVM what arithmetic operations and types need to be legalized. And finally, we will add custom C++ code to enhance the target instruction selection phase.

NOTE: The general idea is broadly similar when implementing Instruction Selection for other architectures, but the specifics will be different. For example, your instruction set could have interesting/complex arithmetic instructions, like long multiplies or multiple-accumulate, that probably require extra code to be emitted properly. Also, we will not be customizing the optimization phases, but you can look at other backends, e.g. XCore and ARM, that have plenty of good examples.

NOTE: The majority of the code shown here is originally from LLVM's existing RISCV backend although it has been greatly simplified for the purposes of this tutorial.

TableGen

TableGen is the notation used to describe the domain-specific features of an instruction set architecture including its registers, instructions, calling convention, etc. The idea is that TableGen is a maintainable and human-readable description which is translated into C++ by the `tablegen` tool and compiled, along with the hand-coded C++ files from your backend, when building LLVM.

NOTE: I actually find TableGen pretty hard to read and even harder to code!

In short, TableGen is a declarative notation that has the following features:

- There are two main components: *records* and *classes*. Records are concrete definitions with a name and its associated fields of values. Classes are abstract records that can be used to make concrete records; they are a means to factor out common code and reduce duplication.
- Records can inherit from one or more classes as well as define their own fields.
- Fields have a name along with a value (or a list of values) of a type (like `bit` or `int`). [Here](#) is a list of the types.
- An `include` declaration is used to include TableGen code from a file into another much like C's `#include`.

The following is an example taken from the LLVM documentation [here](#) showing a very simple TableGen file. It contains a class `C` that defines a field `V` of type `bit` with the value `1`. There is also a declaration for a record `X` derived from class `C`.

```
class C {  
    bit V = 1;  
}  
  
def X : C;
```

Backends need to provide TableGen records declaring instructions, registers, etc. These records must inherit from internalized classes that `tablegen` *magically* handles to generate the appropriate C++ code. For example, a record declaring a register must inherit from the internalized `Register` class. In this post, we will look at the TableGen code for register and instruction declarations.

NOTE: The [TableGen documentation](#) does a pretty good job at explaining what it is and the syntax. However, there is little documentation regarding the internalized classes and other records packaged into LLVM that the backends should reuse; you have to look at the source code for that.

NOTE: Recall from previous posts that our CMake build files have commands to invoke `tablegen` with various arguments to generate C++ files with the `.inc` extension from the `.td` TableGen files. The generated files can be found in the build directory at `build/lib/Target/RISCV` after the build system issues the `tablegen` command. The arguments to `tablegen` are (very briefly) documented [here](#).

NOTE: TableGen can also be used to configure the Scheduling and Formation phase from the Instruction Selection process, but I will not look at that in this tutorial. Take a look at [this](#) video if you are interested.

Registers in TableGen

The register definition for RISCV is pretty straight-forward and can be found in `llvm/lib/Target/RISCV/RISCVRegisterInfo.td`. Lets dissect it to get a feel of how TableGen works.

At the top of the file we find the following class.

```
let Namespace = "RISCV" in {
  class RISCVReg<bits<5> Enc, string n, list<string> alt = []> : Register<n> {
    let HWEncoding{4-0} = Enc;
    let AltNames = alt;
  }
} // end Namespace
```

That code declares a class `RISCVReg` that inherits from the internalized `Register` class which is defined in `include/llvm/Target/Target.td`. The code is also telling us that we must provide up to three arguments when inheriting from this class:

- The encoding for the register `Enc` which is a 5-bit integer of type `bits<5>`.
- A string `n` indicating the human-readable name of the register like `x0`, `x1`, etc in RISCV.
- An optional list of alternative human-readable names for this register. For example, `x2` in RISCV can also be called `sp` i.e. the stack pointer. But note that this list is optional as a register can be declared with no alternative names, i.e. a list with no strings `[]`, if the `alt` argument is not provided.

Also, note that the `Register` class takes one argument: the string with the register's name. The remaining two arguments to `RISCVReg`, i.e. `Enc` and `alt`, are used in `let` statements to overwrite the internal `HWEncoding` and `AltNames` fields originally defined by the `Register` class – you can double-check this by reading the code for `Register` [here](#) and [here](#).

Moving on, we find the following register definitions in the file. Each line of code is a TableGen record that defines a single register. The records inherit from two classes: `RISCVReg` (that we

already discussed) and `DwarfRegNum`. `DwarfRegNum` is yet another internalized class defined [here](#) used to provide debug information for GCC and GDB.

```
def X0 : RISCWReg<0, "x0", ["zero"]>, DwarfRegNum<[0]>;  
...  
def X31 : RISCWReg<31,"x31", ["t6"]>, DwarfRegNum<[31]>;
```

The last few lines in the file define the Register Classes. There are two in RISCW: `GPR` which includes the general-purpose registers and `SP` for the stack pointer.

```
def GPR : RegisterClass<"RISCW", [i32], 32, (add  
  (sequence "X%u", 10, 17),  
  (sequence "X%u", 5, 7),  
  (sequence "X%u", 28, 31),  
  (sequence "X%u", 8, 9),  
  (sequence "X%u", 18, 27),  
  (sequence "X%u", 0, 4)  
)>;  
  
def SP : RegisterClass<"RISCW", [i32], 32, (add X2)>;
```

The records defining a Register Class inherit from the `RegisterClass` class which takes four arguments (and an optional fifth that we will not discuss):

- The namespace which is `RISCW` in our case. This matches the `Namespace` field that we overwrote in the `RISCWReg` class above.
- A list with the data types supported by the registers in this class. This is a list because registers in some architectures can operate with multiple types. For example, registers in some 64-bit machines can operate in 32-bit and 64-bit mode. RISCW is for 32-bit machines only so the registers always operate with 32 bits and their type is `i32`.
- The register alignment when these are stored or loaded from memory.
- A DAG indicating the previously defined registers in this class. And ...
 - I really meant DAG! Note that the `GPR` DAG has an `ADD` mode with 6 child `sequence` nodes. A `sequence` is an operation that takes a string format argument along with start and end values for the sequence. Each value in the sequence is replaced in the format to generate an element. So another way to write this DAG is `(add X10, X17, ..., X3, X4)`.
 - The DAG lists the registers in the order that the register allocator must prioritize their use. For example, the allocator will prefer using `x10` over `x4` from `GPR` if both are available.

The register classes are later used to configure the legalizer.

Instructions in TableGen

Computer architectures encode instructions using a set of formats. For example, the RISC-V architecture uses a 32-bit encoding with 4 base instruction formats.¹ The formats allow encoding instructions with various register operands and immediates of different ranges. Also, each format has a group of unique opcodes (or operation codes). Specific instructions are then encoded using a format along with an opcode; this enables the processor to distinguish instructions from each other and determine their operands.

TableGen instruction definitions in LLVM follow a similar approach to the way instruction set encodings are defined. Thus, the TableGen code is usually split in two components: formats and instructions.

Instruction Formats

The TableGen format code contains definitions of unique identifiers that indicate the format of an instruction. This is used by the C++ code to, for example, correctly emit an instruction's encoding. The RISC-V backend defines the identifiers for the formats like this:

```
class InstFormat<bits<5> val> {
  bits<5> Value = val;
}
def InstFormatPseudo : InstFormat<0>;
def InstFormatR      : InstFormat<1>;
...
def InstFormatOther  : InstFormat<17>;
```

NOTE: A LLVM backend's instruction format code is usually in the file

`llvm/lib/Target/<BACKEND>/<BACKEND>InstrFormats.td` while the actual instruction definitions are in `llvm/lib/Target/<BACKEND>/<BACKEND>InstrInfo.td`. But in complex backends, the instruction definitions can be split across multiple files, for example, the RISC-V backend has a different file for each extension; the files are called `llvm/lib/Target/RISCV/RISCVInstrInfo<EXT>.td` where `<EXT>` is an extension letter like M, A, etc. The RISC-V backend is relatively simple, so all the TableGen code for format and instruction definitions is stored in `llvm/lib/Target/RISCV/RISCVInstrFormats.td` and `llvm/lib/Target/RISCVInstrInfo.td` respectively.

NOTE: The RISC-V backend is based on RISC-V's. Both attempt to closely mirror the instruction formats from the RISC-V reference manual, but this is not a requirement. You can structure your TableGen code however it makes sense for your architecture and compiler.

The RISC-V backend defines records for the opcodes like this:

```

class RISCWOpcode<bits<7> val> {
    bits<7> Value = val;
}
def OPC_LOAD      : RISCWOpcode<0b0000011>;
def OPC_LOAD_FP   : RISCWOpcode<0b0000111>;
...
def OPC_SYSTEM    : RISCWOpcode<0b1110011>;

```

All instruction definitions must inherit from the internalized `Instruction` class. For convenience, the RISCW backend also defines a subclass of `Instruction`, namely `RWInst`, that overwrites a number of fields, like the `Size` and `TSFlags`, while adding extra fields. The `Instruction` class is actually very large with lots of knobs and bolts to configure, so I encourage you to read through LLVM's source code to see what fits your needs.

```

class RWInst<dag outs, dag ins, string opcodestr, string argstr,
            list<dag> pattern, InstFormat format>
    : Instruction {
    field bits<32> Inst;
    field bits<32> SoftFail = 0;
    let Size = 4;
    ...
    let TSFlags{4-0} = format.Value;
}

```

NOTE: A lot of the fields from the `Instruction` class are only useful for specific tasks that your backend might perform. For example, the `Inst` field and the opcode will only be used when emitting object code. So do not bother implementing what you do not need!

Once again for convenience, the RISCW backend defines subclasses of `RWInst` for each instruction format. Our actual instruction definitions will inherit from these "low-level" formats to avoid code duplication.

The format classes mostly overwrite the value of the `Inst` field according to the encoding of the relevant format. Also, there is a special `Pseudo` class that sets the `isPseudo` field from the `Instruction` class. These pseudo instruction are normally place-holders for operations like stack adjustments and function call returns as we will explore in later posts.

```

class Pseudo<dag outs, dag ins, list<dag> pattern, string opcodestr = "",
            string argstr = "">
    : RWInst<outs, ins, opcodestr, argstr, pattern, InstFormatPseudo>,
      Sched<[]> {
    let isPseudo = 1;
}

```

```

let isCodeGenOnly = 1;
}

class RWInstR<bits<7> funct7, bits<3> funct3, RISCWOpcode opcode, dag outs,
            dag ins, string opcodestr, string argstr>
    : RWInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> rs2;
    bits<5> rs1;
    bits<5> rd;

    let Inst{31-25} = funct7;
    ...
    let Opcode = opcode.Value;
}
...

```

There are a few things to highlight about these TableGen classes:

- The `funct*` and `opcode` arguments are used to form the unique opcode which is used to encode the instruction in binary.
- The `outs` and `ins` arguments are DAGs that specify the instruction's output and input operands respectively. The operands are generally registers or immediates, but can also be stack frame locations, global addresses, etc.
- The `opcodestr` is the instruction's mnemonic such as `ADD` for addition, `sub` for subtraction and so on.
- The `argstr` argument is a format string that tells LLVM how the instruction's operands are printed in assembly. For example, if the `outs` argument says that there is a single `$r1` register operand, the `ins` argument says that there is a single `$r2` register operand and the `argstr` has the format `"$r1, $r2"` then the assembly for this instruction will show the output operand first, i.e. right after the mnemonic, followed by a `,` and then the input operand.
- In `Pseudo`, there is also a `pattern` operand that tells LLVM what node patterns in the DAG can be replaced by this instruction during the target instruction selection phase. More on this later!

Instruction Definitions

For convenience, there is yet another TableGen class to facilitate instruction definitions depending on the operands and whether the operation is related to ALU, memory, etc. For example, we have the following `ALU_rr` class to define instructions that use the ALU, i.e. [Arithmetic and Logic Unit](#), which have three general-purpose register operands of type `GPR`: two are inputs and one is the output. Clearly, all `ALU_rr` instructions are of R format because the class inherits from `RWInstR`. Also, the definition of `ALU_rr` is inside a `let` block that

overwrites the `hasSideEffects`, `mayLoad` and `mayStore` fields from `Instruction` with the value 0; most of these fields are self-explanatory although I encourage you to read the code for more information.

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_rr<bits<7> funct7, bits<3> funct3, string opcodestr>
  : RWInstR<funct7, funct3, OPC_OP, (outs GPR:$rd), (ins GPR:$rs1, GPR:$rs2),
    opcodestr, "$rd, $rs1, $rs2">;
```

Defining instructions is actually quite simple when inheriting from `ALU_rr`. For example, here is the code for the `ADD`:

```
def ADD : ALU_rr<0b0000000, 0b000, "add">, Sched<[WriteIALU, ReadIALU, ReadIALU]>
```

The TableGen code is difficult to follow because most backends use a deep inheritance hierarchy to reduce code duplication. So you might want to take out pen and paper to expand some records by hand to see how this works. For example, expanding the first level of `ADD` looks as shown below. This longer version of `ADD` is valid TableGen code and we could replace the previous version of `ADD` with this one – LLVM would still build without any problems!

```
let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
def ADD : RWInstR<0b00000000, 0b000, OPC_OP, (outs GPR:$rd),
  (ins GPR:$rs1, GPR:$rs2), "add", "$rd, $rs1, $rs2">,
  Sched<[WriteIALU, ReadIALU, ReadIALU]>;
```

Defining instructions that have an immediate operand, instead of a register, is done much the same way, but the input operands must be adjusted. For example, the second input of the class `ALU_ri`, shown below, is a 12-bit immediate operand called `$imm12` of class `simm12`. The definition of `simm12` confirms this and tells us that operands of this class are of type `i32`. `simm12` also indicates what encoder, decoder, etc C++ methods are used to handle this operand for the various tasks that the backend might be required to perform. Finally, `simm12` inherits from class `ImmLeaf` which has a condition that must be satisfied for the immediate to match a leaf node in the Selection DAG during target instruction selection (more on this later!).

```
def simm12 : Operand<i32>, ImmLeaf<i32, [{return isInt<12>(Imm);}]> {
  let ParserMatchClass = SImmAsmOperand<12>;
  let EncoderMethod = "getImmOpValue";
  let DecoderMethod = "decodeSImmOperand<12>";
  let MCOperandPredicate = [{
    int64_t Imm;
```



```

    if (MCOp.evaluateAsConstantImm(Imm))
        return isInt<12>(Imm);
    return MCOp.isBareSymbolRef();
}];
let OperandType = "OPERAND_SIMM12";
let OperandNamespace = "RISCWOp";
}

...

let hasSideEffects = 0, mayLoad = 0, mayStore = 0 in
class ALU_ri<bits<3> funct3, string opcodestr>
    : RWInstI<funct3, OPC_OP_IMM, (outs GPR:$rd), (ins GPR:$rs1, simm12:$imm12)
        opcodestr, "$rd, $rs1, $imm12">,
        Sched<[WriteIALU, ReadIALU]>;

```

Some computer architectures, like RISC-V, have instructions that are aliases of other instructions. Generally, these are just for convenience and to improve the readability of assembly code. For example, RISC-V does not have a designated register-register move instruction, so `ADDI` with a 0 immediate is used instead. We can tell LLVM about our aliases with the `InstAlias` and `MnemonicAlias` classes like this:

```

def : InstAlias<"mv $rd, $rs", (ADDI GPR:$rd, GPR:$rs, 0)>;

def : MnemonicAlias<"move", "mv">;

```

Legalizer

The legalizer phases, both for types and operations, is configured with C++ code in your backend's `TargetLowering` subclass. In RISC-V, this class is `RISCVTargetLowering` and its code is in `lib/Target/RISCV/RISCVISelLowering.cpp` and `lib/Target/RISCV/RISCVISelLowering.h`. Most of our changes to support arithmetic instructions are actually in the constructor.

```

RISCVTargetLowering::RISCVTargetLowering(const TargetMachine &TM,
                                           const RISCVSubtarget &STI)
    : TargetLowering(TM), Subtarget(STI)
{
    addRegisterClass(MVT::i32, &RISCV::GPRRegClass);
    ...

    setSchedulingPreference(Sched::RegPressure);

```

```

...

setOperationAction(ISD::SRA_PARTS, MVT::i32, Custom);
...

setOperationAction(ISD::ROTL, MVT::i32, Expand);
...
}

```

There are a few things of remark:

- The call to `addRegisterClass` is telling LLVM our register class and the types of the registers. Recall that this information is used to legalize types.
- The call to `setSchedulingPreference` is used to configure the scheduling and formation phase. Here we are telling LLVM to use an algorithm that minimizes register pressure.
- The calls to `setOperationAction` are used to tell LLVM how to legalize an operation, i.e. a node in the Selection DAG. Recall that there are three options: expand, promote and custom. LLVM automatically handles the former two, for example, RISC-V does not have a bit-rotate instruction, so we asked the compiler to expand that operation into other operations, like bit-shifts and ors, that emulate the rotate. Operations labelled as custom are handled by C++ code in our `TargetLowering` subclass.

Custom operations generate a callback to the `TargetLowering`'s `LowerOperation` method whenever LLVM encounters a DAG node with that operation during legalization. For example, the `SHL_PARTS` operation is used to bit-shift left a 64-bit integer. We told LLVM that we would legalize this operation ourselves with the line `setOperationAction(ISD::SHL_PARTS, MVT::i32, Custom)` in the constructor, so we implemented a function `LowerShlParts` to achieve that. `LowerShlParts` essentially replaces the `SHL_PARTS` DAG node with other legal operation nodes performing 32-bit bit-shifts and ors. I encourage you to take a look at the code to see how this works!

NOTE: The operations that need legalizing are entirely dependent on the instruction set that your backend implements and how well your TableGen code pattern-matches DAG nodes to instructions (more on this in the next section!). If legalizing some operations is too complex or the emitted code is inefficient then your instruction set might be missing some instructions!

Target Instruction Selection

The target instruction selection phase matches DAG patterns provided in the backend's TableGen files to nodes in the Selection DAG. When a match is found, the matching nodes in the Selection DAG are replaced with nodes that have concrete machine (or pseudo) instructions. So the quality of your TableGen patterns is crucial to emitting good code, and in fact, you can expect to spend a hearty amount of time tweaking those patterns!

NOTE: The TableGen patterns are usually found towards the end of the

`llvm/lib/Target/<BACKEND>/<BACKEND>InstrInfo.td` file.

Pattern records inherit from LLVM's `Pat` class which takes two arguments. The first argument is a DAG with the pattern to match. The second argument is a DAG with machine instructions. When the pattern matches something in the DAG, the matched nodes are replaced with the DAG in the second argument.

Unsurprisingly, we are going to use a few classes to help us streamline our TableGen patterns. Here are two such classes:

```
class PatGprGpr<SDPatternOperator OpNode, RWInst Inst>
  : Pat<(OpNode GPR:$rs1, GPR:$rs2), (Inst GPR:$rs1, GPR:$rs2)>;
class PatGprSimm12<SDPatternOperator OpNode, RWInstI Inst>
  : Pat<(OpNode GPR:$rs1, simm12:$imm12), (Inst GPR:$rs1, simm12:$imm12)>;
```

`PatGprGpr` replaces nodes in the DAG with instructions that have two general-purpose register operands as inputs. For example, here is the declaration of a pattern for the `ADD` instruction:

```
def : PatGprGpr<add, ADD>;
```

`PatGprSimm12` replaces nodes in the DAG with instruction that have one general-purpose input operand and a 12-bit immediate. To achieve this, the second operand of the first and second arguments to `Pat` have `simm12` instead of `GPR`. Recall that `simm12` is a record that we discussed in the previous section of this post. It inherits from the class `Operand`, so it could be used in an instruction's definition. `simm12` also inherits from the `ImmLeaf` class, so it can be used in a pattern. The following is an example declaration of a pattern for the `ADDI` instruction that takes in an immediate operand:

```
def : PatGprSimm12<add, ADDI>;
```

NOTE: The `ImmLeaf` class is a pattern that is used to match immediates. Its argument are the type of the immediate, e.g. `i32`, `i16`, etc, and a predicate, i.e. a piece of C++ code, that checks a condition that must be satisfied for the pattern to match a node. For example, `simm12` has a the predicate `return isInt<12>(Imm)` to check whether an integer fits within 12 bits.

NOTE: Sometimes, the same instruction could be matched to more than one pattern in the Selection DAG. In this case, use LLVM's `PatFrag` class alongside `Pat` to avoid code duplication as shown in RISCW's [bit-shift patterns](#).

NOTE: The RISCW and RISCv backends declare instructions first and separately provide pattern records. However, other backends, like ARM or XCore, set up the subclasses of `Instruction`

differently such that they take in a pattern argument. This approach reduces lines of code because the patterns can be provided in the same record defining the instruction, but in my humble opinion, this makes the TableGen code harder to read (and explain!).

The second argument to `Pat` can also transform an operand, usually an immediate, when a match in the DAG has been found. To do so, we first need to declare an operation node that inherits from `SDNodeXForm`. The argument to `SDNodeXForm` takes as an argument a piece of C++ code implementing the desired transformation. Then we use our newly declared node in a pattern as shown in the example below. In this case, the `HI20` node extracts the 20 most significant bits of an immediate.

```
def HI20 : SDNodeXForm<imm, [{  
    return CurDAG->getTargetConstant(((N->getZExtValue()+0x800) >> 12) & 0xfffff,  
                                      SLoc(N), N->getValueType(0));  
}]>;  
  
def : Pat<(simm32hi20:$imm), (LUI (HI20 imm:$imm))>;
```

NOTE: The `SDNodeXForm` nodes do not insert actual instructions. They are normally used to transform immediates, so the operation can be fully resolved before the code is emitted.

As target instruction selection proceeds, LLVM calls the backend's `Select` method from the `SelectionDAGISel` subclass for every Selection DAG node processed. Thus backends can provide code, in the `Select` method, that implements complex pattern matching which is not easily expressed in TableGen. For example, RISCW's `Select` method includes code to replace the constant 0 with an access to the `x0` register.


NOTE: The `SelectionDAGISel` subclass is usually in a backend's `llvm/lib/Target/<BACKEND>/<BACKEND>ISelDAGToDAG.h` and `llvm/lib/Target/<BACKEND>/<BACKEND>ISelDAGToDAG.cpp` files. For example, the relevant code from the RISCW backend is in `llvm/lib/Target/RISCW/RISCWISelDAGToDAG.h` and `llvm/lib/Target/RISCW/RISCWISelDAGToDAG.cpp`.

Closing Thoughts

There are actually quite a few ways to implement an LLVM backend for an architecture, so make sure that you assess what fits your needs best before settling into any one approach. Also, keep in mind the following summary of TableGen classes that you will almost certainly need when writing your backend.



TableGen class	What is it for?
<code>Register</code>	Register declarations
<code>RegisterClass</code>	Register Class declarations
<code>Instruction</code>	Instruction declarations
<code>Operand</code>	Instruction operands
<code>ImmLeaf</code>	Pattern-matching immediates
<code>SDNodeXForm</code>	Transforming immediates after a match
<code>Pat</code>	Providing a pattern to match in the Selection DAG
<code>PatFrag</code>	Providing a pattern fragment with multiple patterns to match in the Selection DAG

Notes

1. A complete description of RISC-V's instruction formats is in Chapter 2 of the [reference manual](#). 

Source Code Artisan

Source Code Artisan
andresag1118@gmail.com

 [andresag01](#)
 [andresag](#)

A personal blog on computing topics