Chapter 2

Implementing new RISC-V Instructions with the LLVM Compiler Infrastructure

Gustavo Leite, Instituto de Computação, Unicamp (Email) (Lattes)
Carlos E. C. Barbosa, Instituto de Computação, Unicamp (Email)
Hervé Yviquel, Instituto de Computação, Unicamp (Email) (Lattes)
Sandro Rigo, Instituto de Computação, Unicamp (Email) (Lattes)

Abstract

RISC-V is an open and modular architecture standard whose design makes it a fertile ground for innovation in computer architecture. Its extensible ISA enables the introduction of domain-specific instructions, tailored for applications ranging from embedded systems and IoT to high-performance computing (HPC) and artificial intelligence (AI). As new extensions appear, such as those for vector and matrix operations, compiler support becomes a key enabler for their practical adoption. In this context, understanding how to extend a compiler backend is an essential skill for researchers and developers working on custom accelerators.

This minicourse presents a step-by-step introduction to adding new instructions to the RISC-V backend of the LLVM compiler infrastructure. Participants will learn the complete process of extending LLVM to support a custom matrix processing extension, including the definition of new instruction formats, encoding schemes, and register sets using the TableGen declarative language. The tutorial walks through the design of a small "xmatrix" extension that introduces 32 dedicated 512-bit matrix registers and a small set of load/store and arithmetic operations for 4×4 tiles. Through practical examples, attendees will see how to integrate these instructions into LLVM, assemble and disassemble them, and prepare the compiler for future integration with simulators, custom-chips and higher-level languages such as C/C++.

2.1. Introduction

Achieving maximum efficiency in high-performance computing requires a deep understanding of multiple layers of the system. Beyond the application itself, programmers must optimize for different levels of parallelism. For applications targeting machine clusters, this often involves understanding the cluster network's topology, link latency, and throughput. Within each node, engineers work with thread-level parallelism and must deal with thread synchronization problems. For each individual execution thread, there exists a third axis of instruction-level parallelism, where specialized extensions speed up specific operations. A prime example is Intel's Advanced Vector Extension (AVX), which introduces new vector registers and instructions for highly efficient vectorized code. Other vendors offer similar extensions, such as Arm's Scalable Vector Extension (SVE) and the RISC-V Vector Extension (RVV). More recently, hardware support for matrix operations has become common with extensions like Intel's Advanced Matrix Extension (AMX) and Arm's Scalable Matrix Extension (SME).

RISC-V is a relatively new architecture in the market, and its strength lies in the ISA standard being open-source and extensible. That means that any vendor building RISC-V hardware can implement their own instructions. RISC-V for high-performance computing is still in its early days, and currently, there is no ratified standard for matrix operations similar to Intel's AMX and ARM's SME mentioned before. Our research group has been working on building a prototype matrix accelerator for RISC-V. That involves proposing new ISA extensions, developing simulators, optimized software kernels that will explore those new instructions, and also retargeting a compiler to generate code using the new ISA extension. In this minicourse, we address this last step. The goal is to show how one can expand an existing RISC-V LLVM compiler backend, including new matrix specialized instructions.

This document is organized as follows: Section 2.2 provides background on the RISC-V instruction set and the LLVM compiler infrastructure. Section 2.3 presents the extension that will be implemented, while Section 2.4 provides the steps to write it using TableGen syntax. Section 2.5 demonstrates the new instructions in action. And finally, Section 2.6 provides concluding remarks and suggestions for next steps.

2.2. Background

This section covers the background necessary to carry out the implementation of the Matrix Extension for RISC-V. In Section 2.2.1 the basics of the RISC-V instruction set are covered, while Section 2.2.2 introduces the necessary tools to work with LLVM.

2.2.1. RISC-V

RISC-V is an open standard Instruction Set Architecture (ISA). It defines a set of registers and instructions that operate on those registers. Because RISC-V is free to use, hardware vendors can implement their processors according to the specification and automatically leverage the existing tools around it, such as compilers, linkers, operating systems, and more.

As the name suggests, RISC-V follows the Reduced Instruction Set Computer

(RISC) philosophy: it provides a simpler set of instructions that can be implemented efficiently in hardware. Consider Intel's x86 instruction set. Most instructions accept operands either from memory or from registers, sometimes with a variety of addressing modes. RISC-V, on the other hand, defines separate sets of instructions to access memory and to perform arithmetic computations. Thus, operands can only come from a register or be immediately encoded in the instruction itself. Besides that, RISC-V also reserves opcodes in the standard for custom, processor-specific instructions that are not tied to any particular ratified extension. The main use case for custom instructions is to add control instructions or optimized ALU instructions that have different semantics from the ones already defined in the standard.

RISC-V was created with extensibility in mind. The instruction set is organized into modular components called extensions, which hardware vendors can optionally implement. Every RISC-V processor must support a minimal core known as the Base Integer Instruction Set, which is uniquely identified by the letter "I". For example, the I instruction set does not even include multiplication instructions, which are instead provided by the "M" (for multiplication) extension. Similarly, the "F" and "D" extensions introduce single- and double-precision floating-point operations, respectively. Additional extensions cover vector processing, atomic operations, and bit manipulation, among others. Detailed descriptions of these modules can be found in the official RISC-V specifications.

Due to this extensibility and flexibility, RISC-V has rapidly gained traction across a wide range of computing domains, from deeply embedded IoT devices and secure hardware enclaves, to automotive controllers, high-performance computing (HPC) clusters, and datacenter-scale AI accelerators. As the ecosystem grows, so does the set of ISA profiles and custom instruction sets tailored for these use cases. The recent RISC-V profiles initiative (e.g., RVA22, RVA23, etc.) defines standardized combinations of extensions to facilitate software portability while still embracing specialization.

This flexibility at the ISA level, however, must be matched by equivalent flexibility in the software toolchain, particularly in the compiler backend. Robust compiler support is essential to make custom instructions usable in real-world applications and to ensure code generation fully exploits the capabilities of the underlying hardware.

Instruction Encoding

The RISC-V Base Instruction Set defines instructions to be 32-bits long, grouped in six instruction types, shown in Figure 2.1. It is possible to note that the opcode always occupies the least significant seven bits of the instruction. The remaining bits (7-31) are interpreted differently depending on the instruction type.

Following is a description for each kind of instruction:

R-type – Represents instructions that take operands from registers `rs1`/`rs2` and write the output to another register `rd`. Fields `funct3` and `funct7` control which operation is performed (i.e., add, sub, or, and, etc). Examples include arithmetic and logic instructions such as `add`, `xor`, and `mul`, etc. This is the most common instruction type.

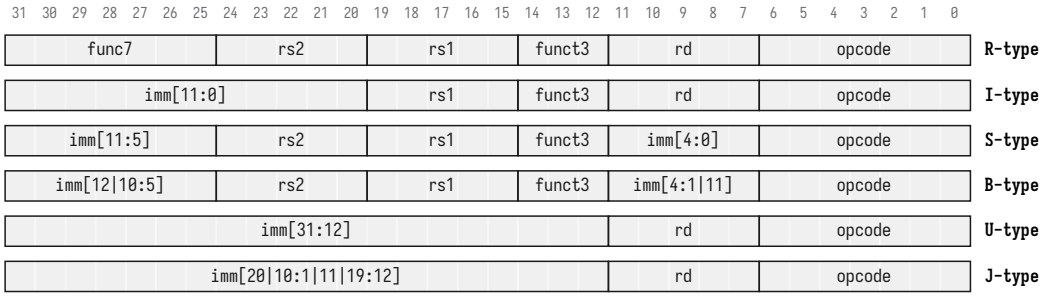

Figure 2.1. RISC-V base instruction formats.

I-type – Represents instructions that take one operand from a register `rs1` and another immediate operand. The output is a register `rd`. The funct3 `field` controls what the instruction computes. Examples include arithmetic instructions such as `addi`, `xori`, and `muli`, etc. This instruction format is also used to represent load instructions in which the immediate value is an offset from the address stored in the source register.

S-type – Represents store operations: the value to be stored is given by the second source register, the base address of the store is in the first source register, and there is an immediate offset from the base address.

B-type – Represents branch instructions. Compare the values in registers `rs1` and `rs2` , and add the immediate value to the program counter if they are equal. The field `funct3` encodes which comparison function to use (equal, not equal, less than, or greater than or equal).

U-type – Represents upper immediate instructions. Takes the immediate value and writes it to the upper part of register `rd`.

J-type – Represents the jump and link instruction. It saves the address of the next instruction (Program Counter + 4) into the destination register (rd), then updates the program counter by adding the immediate value, effectively performing a jump.

Opcodes

As mentioned earlier, RISC-V instructions use the lower seven bits to represent the opcode. Bits 0 and 1 of the opcode are always 1. Therefore, it is possible to represent 32 opcodes using the remaining five bits. The reason for this restriction is to differentiate 32-bit instructions from the compressed 16-bit instructions defined in the Compressed Instruction Set (“C” Extension). In this standard, the opcode occupies only two bits and can assume values 00, 01, or 10. In this manner, the hardware can unequivocally distinguish normal instructions from compressed instructions by just checking the first two bits. Table 2.1 shows how opcodes are encoded, taking the form `XXYYY11`, where `XX` is

mapped to the rows and YYY is mapped to the columns. The opcodes with YYY = 111 are currently reserved and have been omitted from the table.

	000	001	010	011	100	101	110
00	LOAD	LOAD-FP	*custom-0*	MEM	IMM	AUIPC	IMM32
01	STORE	STORE-FP	*custom-1*	AMO	REG	LUI	REG32
10	MADD	MSUB	NMSUB	NMADD	REG-FP	REG-V	*custom-2*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	REG-VE	*custom-3*

Table 2.1. RISC-V Opcodes.

Note the presence of four custom opcodes in the table. Such opcodes cannot be used by standard extensions and are reserved for vendor-specific instructions. The programmer can select any one of them to implement the custom matrix instructions.

2.2.2. LLVM

Compilers are generally implemented as a series of transformations on a source program. Common software engineering practice divides this pipeline into three stages:

The Frontend is responsible for parsing the input program, performing semantic analysis, and translating the code into an intermediate representation (IR).

The Middle-end takes the program in this intermediate form, runs a series of analyzes and transformation passes, and translates it to a lower-level representation that is very close to assembly language.

The Backend takes the output from the previous step, performs target-specific transformations (instruction selection, register allocation, etc) and generates executable or object code for the target architecture.

This layered structure, particularly the use of an intermediate representation, is crucial because it allows the frontend and backend to be decoupled. It becomes possible to compile many high-level languages (e.g., C/C++, Zig, Rust) into the same IR and then use a single backend to target multiple target architectures. Figure 2.2 visually represents this idea. In particular, LLVM is a framework for building compilers that implements a powerful intermediate representation called LLVM IR, along with several analysis and transformation passes, commonly referred to as “optimizations”.

In this context, LLVM has become the compiler infrastructure of choice for RISC-V development, offering a modular and retargetable backend that aligns well with RISC-V’s philosophy. Being able to integrate new instructions into LLVM allows hardware designers, researchers, and software developers to:

- Prototype and evaluate ISA changes rapidly;
- Generate optimized code for new instructions;

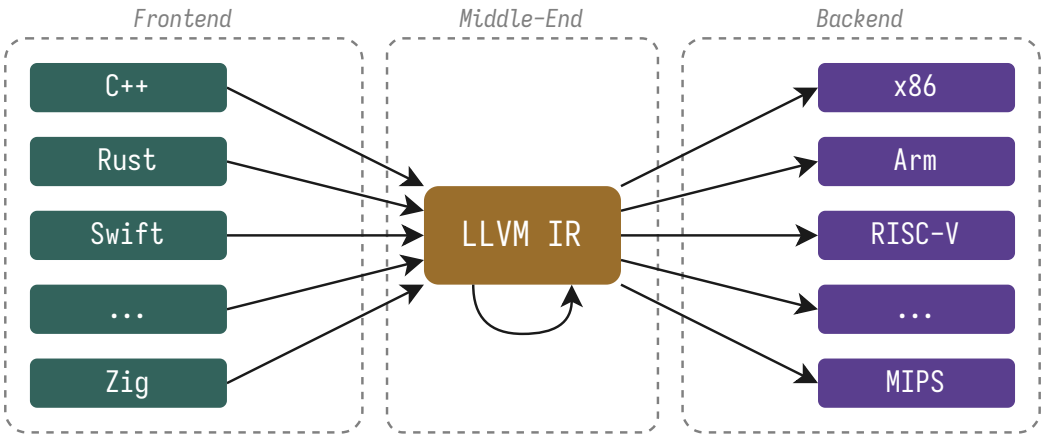

Figure 2.2. Overview of the compilation steps.

- Provide toolchain support for new RISC-V profiles or custom accelerators.

Focusing on how to extend the LLVM backend for RISC-V by adding new instructions, the modifications will occur in the backend, after the program is translated out of LLVM IR. The representation used in the LLVM backend is called Machine IR. However, it will not be necessary to interact with this representation since the backend can be extended by using a domain-specific language called TableGen.

2.2.3. TableGen

TableGen is a declarative language largely employed in the LLVM project. It allows programmers to define records of structured information that are used during the build process. For example, instead of writing an assembler by hand, this language shall be used to declare how the instructions are encoded (both in assembly and binary form) and then allow the C++ implementation to be generated automatically. This frees the programmer from writing boilerplate code and makes it much simpler to change the instructions later if the need arises. The syntax of TableGen is presented below.

TableGen Syntax

A record is a uniquely named list of typed key-value pairs, referred to as properties. Listing 2.1 provides an example record. The keyword `def` begins the declaration in line 1, followed by the record name `MyRecord` and a block delimited by curly braces. The record holds two properties: `Key1` of string type and value “Value 1” (line 2), and `Key2` of integer type with value 2 (line 3).

Nevertheless, records are rarely defined in this explicit manner. Instead, one can define a class that lists all properties a record should possess, much like a template, and then instantiate a record from it. Listing 2.2 shows a TableGen class in action. The `class` keyword is followed by its name `Car` and a list of typed parameters inside angle brackets

```
def MyRecord {
    string Key1 = "Value 1";
    int Key2 = 2;
}
```

Listing 2.1. TableGen record example.

(line 1). The body of the class is specified inside curly braces as a list of properties (lines 2-4). Now, instantiating a record from this "Car" class consists of using the `def` keyword as before, followed by the record name, a colon, and a list of classes from which the record is built (line 7). Note that classes themselves can also be derived from other classes.

```
class Car<string model, string color> {
    string Model = model;
    string Color = color;
}

def MyKombi : Car<"Kombi", "white">;
```

Listing 2.2. TableGen class example.

It is very common to define multiple records that share the values of some fields but not others. Consider the `Car` class from the previous example. Suppose that one would like to define a list of N car models in both black and white colors, thus $2N$ records are required. TableGen provides a pattern for efficiently encoding such variations with a multiclass, as shown in Listing 2.3.

It starts with the `multiclass` keyword, followed by a name and a list of parameters (line 1). Lines 2 and 3 define inner records for the multiclass. Note that both use the same model name but with different colors. Finally, three car models are defined in lines 6-8: Brasilia, Kombi, and Fusca from the `ColoredCar` multiclass. In order to distinguish between instantiating from a regular class, the keyword `defm` is used when instantiating from multi-classes. Writing this code results in 6 records being defined, where the name of the outer record is concatenated with the inner record, for example: `BrasiliaBlack`, `BrasiliaWhite`, `KombiBlack`, and so on.

```
multiclass ColoredCar<string model> {
    def Black : Car<model, "black">;
    def White : Car<model, "white">;
}

defm Brasilia : ColoredCar<"Brasilia">;
defm Kombi    : ColoredCar<"Kombi">;
defm Fusca    : ColoredCar<"Fusca">;
```

Listing 2.3. TableGen multiclass example.

Another common pattern in TableGen is to specify a property that is shared by the entire file or a large portion of it. Continuing with the previous example, suppose that now

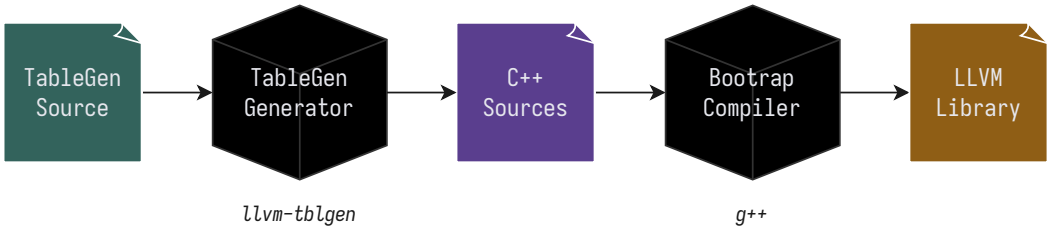

Figure 2.3. TableGen build process.

car records include a "Manufacturer" property. Even though multi-classes could be used for this purpose, an alternative syntax is shown in Listing 2.4. The `let` keyword introduces a comma-separated list of key-value pairs of properties, followed by the keyword `in`, and a block in curly braces (line 1). Inside the block, the records are defined normally, which effectively adds a "Manufacturer" property with the value "VW" to all of them at once (lines 2-4). While this feature might seem redundant for multi-class scenarios, both serve distinct purposes, as shown in the next section.

```
1 let Manufacturer = "VW" in {
2     defm Brasilia : ColoredCar<"Brasilia">;
3     defm Kombi    : ColoredCar<"Kombi">;
4     defm Fusca    : ColoredCar<"Fusca">;
5 }
```

Listing 2.4. TableGen let/in syntax example.

The last syntax relevant to this course is when we would like to create a sequence of records using an index in increasing (or decreasing) order. Listing 2.5 presents the `foreach` operator. It behaves like a for loop using the `Index` induction variable. The range of the loop goes from 0 to 3 (non-inclusive), increasing by 1 on each iteration. The symbol # is the concatenation operator. A record is defined in the body of the for-each loop with the name "Car" concatenated with the current index. The model name also uses the concatenation operator. In the end, the code shown in this example will create three records: `Car0`, `Car1`, and `Car2`.

```
1 foreach Index = !range(0, 3, 1) in {
2     def Car # Index : Car<"Model " # Index, "black">;
3 }
```

Listing 2.5. TableGen foreach example.

This concludes the brief introduction to TableGen. The language includes many additional constructs not covered in this tutorial. For a complete reference, check the TableGen Programmer's Reference[1] in the LLVM documentation.

[1] https://llvm.org/docs/TableGen/ProgRef.html

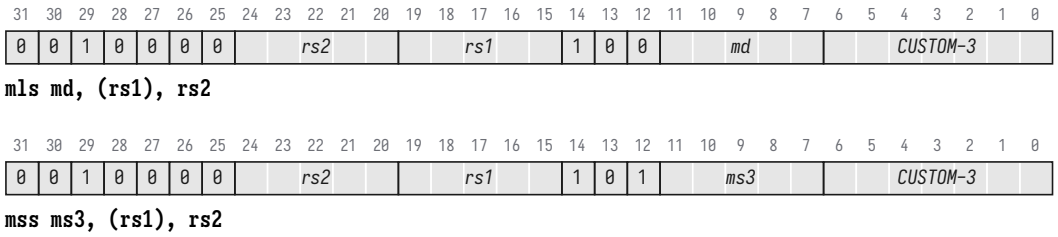

Figure 2.4. Memory (load/store) instruction encoding for the "xmatrix" extension.

As mentioned before, TableGen records do nothing on their own. They are instead used during the build process of LLVM itself to generate C++ code. This process is depicted in Figure 2.3. The TableGen source files are fed as input to a tool called `llvm-tblgen`, which expands all of the class instantiations into records and then filters these records through a "generator". EAlthough the internal workings of generators are beyond the scope of this course, we can assume that they produce a collection of C++ source files containing the implementation of the assembler, disassembler, and other tools. Still during the build process, those C++ source files are compiled using the bootstrap[2] compiler (typically `g++`) and the code end up in a statically linked library for LLVM.

2.3. RISC-V Matrix Extension

The proposed extension, called "xmatrix", adds 32 new matrix registers `m0-m31` with length of 512 bits, interpreted as a 4x4 matrix of 32-bit elements. The extension also includes memory, arithmetic, and move operations that utilize the matrix registers. The assembler must produce matrix instructions only when the "xmatrix" feature is enabled; otherwise, the code should be rejected with an error. Instructions from this extension shall use the *custom-3* opcode (`1111011`) from Table 2.1, chosen arbitrarily from the custom opcodes.

2.3.1. Memory Instructions

First of all, there must be a way to read matrix registers to and from main memory. Therefore, the instructions "matrix load" and "matrix store" are defined according to the encoding in Figure 2.4.

The **Matrix Load with Row-Stride** (`mls`) instruction reads a matrix register from memory, starting at a specified base address. The destination matrix register is encoded using five bits in the field `md` (`m0-m31`). The base address comes from the source operand in `rs1` , which encodes a scalar register. Also, there is an additional source operand called the leading dimension of the matrix (or equivalently, "row-stride") in the scalar register `rs2`. Figure 2.5 demonstrates how this instruction works.

The diagram shows the memory as a sequence of elements, each with 32-bits. The base address points to the first element that should be read. The load operation will read a single row of four elements and store it in the first row of the matrix register. Then,

[2]Bootstrapping is the process of using one compiler to build the source of another compiler.

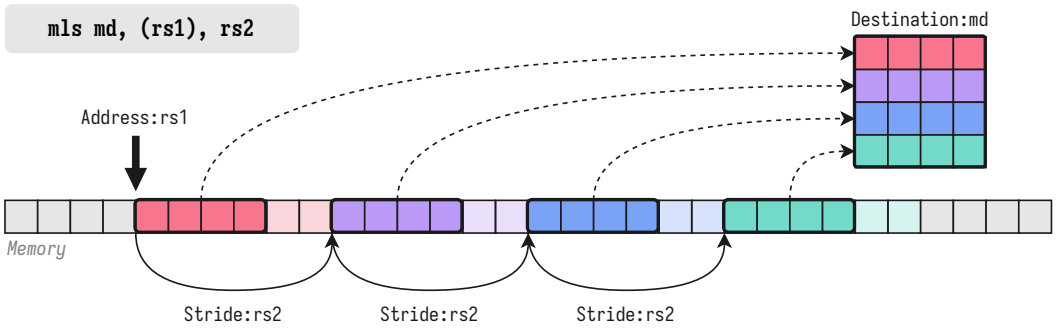

Figure 2.5. Matrix Load with Row-Stride (`mls`) instruction.

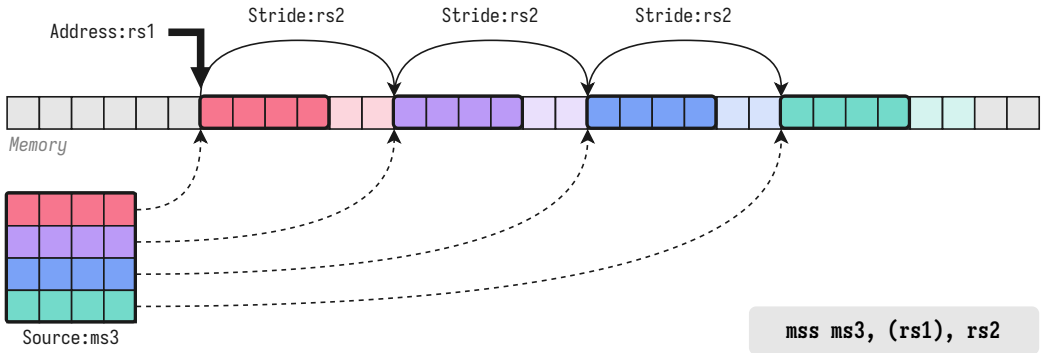

Figure 2.6. Matrix Store with Row-Stride (`mss`) instruction.

instead of reading the second row at an offset of four elements from the base address, it will actually jump "row-stride" elements forward before reading the next row. The process repeats until all rows of the destination register are filled.

The row-stride allows programmers to read a small 4×4 tile that is inside a larger $N \times M$ matrix. Without the stride, it would be necessary to first pack the corresponding elements of the tile sequentially in a separate region of memory using scalar operations and then execute a contiguous load. Notably, this strategy is highly inefficient in practice.

A contiguous load can be simulated when the row-stride matches the length of a row in the matrix register (i.e., `rs2 = 4`). Additionally, the row-stride can also be zero (the same four elements will be replicated to each row) or even negative (the rows are read in reverse order).

Similar to the load instruction, the **Matrix Store with Row-Stride** (`mss`) instruction writes a matrix register to memory, starting at a base address. It takes three source operands: the matrix register to be stored in `ms2`; the base address in memory from register `rs1`; and the row-stride in `rs2`. Its behavior is illustrated by Figure 2.6.

The first row of the matrix register is written to memory starting from the base address. Then, the pointer is incremented by the leading dimension, arriving at the address

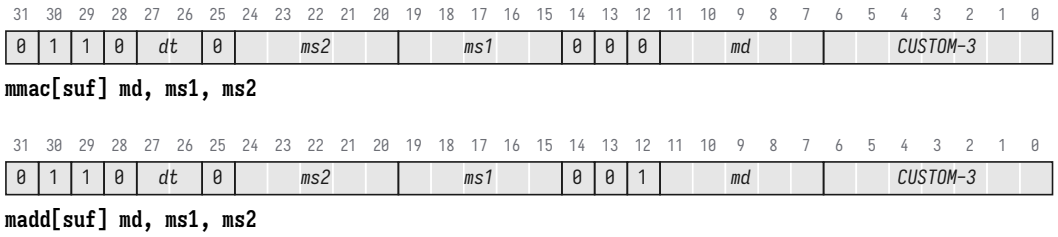

Figure 2.7. Arithmetic instructions encoding in the "xmatrix" extension.

where the second row will be stored. This process is repeated until the entire register has been written to memory. Similarly, the store instruction, like the load instruction, also accepts a null or negative stride.

2.3.2. Arithmetic Instructions

With the ability to read and write to matrix registers from memory, attention is focused on arithmetic instructions. Two groups of instructions are shown in Figure 2.7. The first group is the multiply-and-accumulate instruction, `mmac`. It is a group because there are three different data-types that this instruction can operate on: signed, unsigned, and floating point. Therefore we add a suffix `s`, `u`, or `f` respectively to each instruction. Likewise, another instruction that adds two matrices `madd` is defined for each data-type.

Bits 26 and 27 of the instruction encode the data-type being `00` for unsigned integer elements, `01` for signed integer elements, and `10` for single-precision floating point elements.

2.3.3. Movement Instructions

Finally, the instruction `mmv.xm` writes the value of a scalar register to all positions of a matrix register as shown in Figure 2.8. This operation is particularly useful for initialization, for example clearing a matrix register by moving the contents of `x0` (always zero). This pattern is so common, in fact, that a pseudo-instruction "`mzero md`" is defined to represent "`mmv.xm md, x0`".

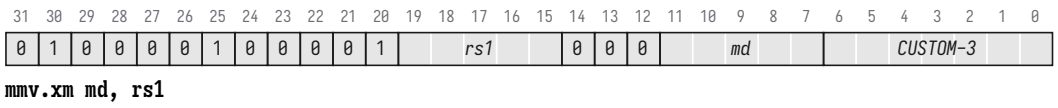

Figure 2.8. Movement instruction enconding in the "xmatrix" extension.

2.4. Implementation

In this section, it will be demonstrated how to implement the extension defined in the previous section. In Section 2.4.1, a new extension is declared to contain the new registers and instructions. Section 2.4.2 shows how to declare the new matrix registers. Finally,

the instructions themselves are defined in Sections 2.4.3, 2.4.4, and 2.4.5.

2.4.1. Feature

In order to define a new RISC-V extension, we must create two records in the file `RISCVFeatures.td`. This file is shown in Listing 2.6. The first record (lines 3-4) declares a new `RISCVExtension` that takes the major and minor versions of the specification as integers and a string with the extension name. In this particular case, the record name matters and must begin with the prefix `FeatureVendorX`. It will automatically add a new architecture flag "xmatrix" on the command-line.

```
1 // File: llvm/lib/Target/RISCV/RISCVFeatures.td
2 def FeatureVendorXMatrix
3     : RISCVExtension<0, 1, "Matrix Extension">;
4
5 def HasVendorXMatrix
6     : Predicate<"Subtarget->hasVendorXMatrix()">,
7       AssemblerPredicate<(all_of FeatureVendorXMatrix),
8                          "'XMatrix' (Matrix Extension)">;
```

Listing 2.6. RISC-V Matrix Extension declaration.

The second record (lines 6-9) defines predicates that indicate whether the feature is enabled or not. It is created from two classes: `Predicate` , which takes a C++ expression that performs the test; and `AssemblerPredicate` , which takes a TableGen predicate followed by a name. Although redundant, both predicate classes must be instantiated since they are used in different parts of the generation.

2.4.2. Registers

After declaring the new feature, it is possible to define the registers in Listing 2.7. The operator `foreach` is employed here. For each iteration, define a register (`RISCVReg`) with an index and a name (line 2). The # symbol means concatenation in TableGen. Therefore, registers are named "m0", "m1", etc.

```
1 // File: llvm/lib/Target/RISCV/RISCVRegisterInfo.td
2 foreach Index = !range(0, 32, 1) in {
3     def M#Index : RISCVReg<Index, "m"#Index>;
4 }
5
6 def MR : Registerclass<[v16i32, v16f32], (sequence "M%u", 0, 31), 1>;
```

Listing 2.7. Matrix registers declaration.

A register class is also defined in line 6. It takes, as the first parameter, a list of types that can be stored in such a register. In this case, a vector of 16 (4x4) 32-bit integers or 16 single precision floating point elements. The second parameter is a list of strings containing register names. In particular, it expands to a list of all matrix registers.

2.4.3. Memory Instructions

A new instruction can be defined in TableGen by instantiating a record from the `RVInst` class, as shown in Listing 2.8. This class has the following type parameters:

- `outs` specifies which operands are output from the instruction, along with the register class they belong to;
- `ins` specifies which operands are input to the instruction, along with their register class;
- `opcodestr` is a string with the assembly mnemonic that identify this instruction.
- `argstr` is a format string for the operands, which placeholders starting with the character "$";
- `pattern` is a list of patterns (i.e., record created from class `Pattern`) that converts an LLVM IR operation to Machine IR. This parameter shall be left empty; and
- `format` specifies which format is used by this instruction (R-type, I-type, etc from Figure 2.1).

Starting at line 1, a new subclass `MatrixInstLoad` is created from the `RVInst` class (line 2). Our new class takes the same parameters mentioned above and "forwards" them to the base class. It defines new operands `md`, `rs1`, and `rs2`, all encoded with five lines (lines 4-6). Next, the instruction encoding is declared. For each bit range in the `Inst` property we write which value it should take (lines 8-13). For example, the first seven bits (6-0) correspond to the opcode, thus we assign the value of the `OPC_CUSTOM_3` (line 13). This record has been previously defined in LLVM and its value is exactly the same as described in Table 2.1. According to the instruction encoding presented earlier, bits 31-25 are constant, therefore we directly assign the value `0b0010000` to it (line 8).

With the class properly coded, a new record called `MLS` is created (lines 16-19). We first wrap the record definition in a `let`/`in` block that sets a few required properties (line 16). The predicate previously defined that identifies the matrix extension is used to indicate that this instruction is only available when the feature is enabled. Additionally, we indicate that this instruction has no side effects (i.e. does not change a register that is not explicitly identified in the format), it may load from memory but never stores.

The instruction itself uses the mnemonic "mls" and its arguments are formatted like "md, rs1, rs2" (line 17). Pay close attention to the outs/ins declarations (line 17). This syntax is known as a DAG type and encode a list of operands with their type attached to it. We say the output operand (`outs`) from this instruction is the `md` register that should be one of those in the `MR` register class defined in the previous section. Another DAG value is used to specity the inputs `ins`: register `rs1` from `GPRMemZeroOffset` and register `rs2` from the `GPR` register class. The `GPR` is a register class that includes all general purpose registers `x0-x31`, while the `GPRMemZeroOffset` class includes all general

```
// File: llvm/lib/Target/RISCV/RISCVInstrInfo.td
class MatrixInstLoad<string opcodestr, string argstr, dag outs, dag ins
    >
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> md;
    bits<5> rs1;
    bits<5> rs2;

    let Inst{31-25} = 0b0010000;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b100;
    let Inst{11-7}  = md;
    let Inst{6-0}   = OPC_CUSTOM_3.Value;
}

let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 1,
    mayStore = 0 in {
    def MLS : MatrixInstLoad<"mls", "$md, $rs1, $rs2",
            (outs MR:$md), (ins GPRMemZeroOffset:$rs1, GPR:$rs2)>;
}

class MatrixInstStore<string opcodestr, string argstr, dag outs, dag
    ins>
    : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> ms3;
    bits<5> rs1;
    bits<5> rs2;

    let Inst{31-25} = 0b0010000;
    let Inst{24-20} = rs2;
    let Inst{19-15} = rs1;
    let Inst{14-12} = 0b101;
    let Inst{11-7}  = ms3;
    let Inst{6-0}   = OPC_CUSTOM_3.Value;
}

let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 0,
    mayStore = 1 in {
    def MSS : MatrixInstStore<"mss", "$ms3, $rs1, $rs2",
            (outs), (ins MR:$md, GPRMemZeroOffset:$rs1, GPR:$rs2)>;
}
```

Listing 2.8. TableGen classes for memory instructions.

purpose registers but are used as a memory offset operand. This causes this operand to be wrapped in parenthesis, like the scalar load instruction `ld x3,(x4)`.

The store follows a similar pattern, a subclass of `RVInst` called `MatrixInstStore` is created to contain the encoding of all 32 bits (lines 21-33). The record `MSS` is instantiated with the corresponding mnemonic, argument format string and the outs/ins DAG patterns (lines 36-37). Note however that store operations do not produce values, therefore all operands are inputs and the outputs are empty. Another observation is that we use the same predicate as the load and also identify that this instruction may store to memory.

2.4.4. Arithmetic Instructions

In order to specify arithmetic instructions with TableGen, it is useful to first define constants for the "funct3" and "dt" fields (Listing 2.9). A class that holds a string of 3-bits is defined (lines 2-4) and then the constants for `mmac` and `madd` are defined according to the encoding (lines 5-6). A class that takes a 2-bit string as a parameter is declared to encode the data-type (lines 8-10). Records for each data type are declared according to the specification (lines 11-13).

```
1  // File: llvm/lib/Target/RISCV/RISCVInstrFormats.td
2  class MatrixAluFunct3<bits<3> value> {
3      bits<3> Value = value;
4  }
5  def MMacFunct3 : MatrixAluFunct3<0b000>;
6  def MAddFunct3 : MatrixAluFunct3<0b001>;
7
8  class MatrixDataType<bits<2> value> {
9      bits<2> Value = value;
10 }
11 def MUnsigned : MatrixDataType<0b00>;
12 def MSigned   : MatrixDataType<0b01>;
13 def MFloat    : MatrixDataType<0b10>;
```

Listing 2.9. Matrix instruction fields.

Similar to the memory instructions, a new class is created in Listing 2.10 from the `RVInst` base class (line 2). It takes the same parameters are the classes for memory instruction but also the bits corresponding to funct3 and the ones corresponding to the data-type field. The bits of the instruction are declared according to the specification (lines 8-15). Since all arithmetic instructions are very similar, another class is created in line 18. It indicates how the arguments of the instruction are formatted in assembly, that is "$md, $ms1, $ms2" in the argument format string. It also encodes that register "md" is an output and registers "ms1" and "ms2" are inputs, all of them from the class of matrix registers (line 19).

Now it is time to encode the variations regarding the data-type using a multiclass (line 21). It takes the opcode and funct3 fields as parameters and defines three inner records, one for each data-type, passing along the correct constants defined earlier (lines 22-24). Finally, instruction records themselves are declared by passing the corresponding opcode and funct3 bits (lines 27-30). Because of the multiclass, the result is a total of six

```
// File: llvm/lib/Target/RISCV/RISCVInstrInfo.td
class MatrixInstALUBase<string opcodestr, string argstr,
    MatrixAluFunct3 funct3, MatrixDataType dt, dag outs, dag ins>
        : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
    bits<5> md;
    bits<5> ms1;
    bits<5> ms2;

    let Inst{31-28} = 0b0110;
    let Inst{27-26} = dt.Value;
    let Inst{25}    = 0b0;
    let Inst{24-20} = ms2;
    let Inst{19-15} = ms1;
    let Inst{14-12} = funct3.Value;
    let Inst{11-7}  = md;
    let Inst{6-0}   = OPC_CUSTOM_3.Value;
}

class MatrixInstALU<string opcodestr, MatrixAluFunct3 funct3,
    MatrixDataType dt>
        : MatrixInstALUBase<opcodestr, "$md, $ms1, $ms2", funct3, dt, (
    outs MR:$md), (ins MR:$ms1, MR:$ms2);

multiclass MatrixInstALU_SUF<string opcodestr, MatrixAluFunct3 funct3>
    {
    def S : MatrixInstALU<opcodestr,     funct3, MSigned>;
    def U : MatrixInstALU<opcodestr#"u", funct3, MUnsigned>;
    def F : MatrixInstALU<opcodestr#"f", funct3, MFloat>;
}

let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 0,
    mayStore = 0 in {
    defm MADD : MatrixInstALU_SUF<"madd", MAddFunct3>;
    defm MMAC : MatrixInstALU_SUF<"mmac", MMacFunct3>;
}
```

Listing 2.10. Arithmetic matrix instruction declaration.

new records being defined: `MMACS`, `MMACU`, `MMACF`, `MADDS`, `MADDU`, and `MADDF`. No arithmetic instruction perform a load, store, or have any side-effects.

2.4.5. Movement Instructions

The final move instruction is given in Listing 2.11. The pattern closely resembles the other instructions: define a new class `MatrixInstMov` (line 2-3), declare the operands (lines 4-5), declare the encoding (lines 7-11), and finally instantiate the record (line 15). We also define a pseudo-instruction `mzero` as an anonymous record that inherits from `InstAlias` prividing the assembly format as a string and a DAG pattern to substitute for (line 16). Similar to arithmetic instructions, movement instructions neither load, nor store to memory and have no side effects.

```
1  // File: llvm/lib/Target/RISCV/RISCVInstrInfo.td
2  class MatrixInstMov<string opcodestr, string argstr, dag outs, dag ins>
3      : RVInst<outs, ins, opcodestr, argstr, [], InstFormatR> {
4      bits<5> md;
5      bits<5> rs1;
6
7      let Inst{31-20} = 0b010000100001;
8      let Inst{19-15} = rs1;
9      let Inst{14-12} = 0b000;
10     let Inst{11-7}  = md;
11     let Inst{6-0}   = OPC_CUSTOM_3.Value;
12 }
13
14 let Predicates = [HasVendorXMPE], hasSideEffects = 0, mayLoad = 0,
       mayStore = 0 in {
15     def MMV_XM : MatrixInstMov<"mmov.xm", "$md, $rs1", (outs MR:$md), (
       ins GPR:$rs1)>;
16     def : InstAlias<"mzero $md", (MMV_XM MR:$md, X0)>;
17 }
```

Listing 2.11. Movement instruction declaration.

2.5. Testing

With the implementation of the new instructions complete, we now proceed to compile the modified LLVM sources (Section 2.5.1) and test the assembler and disassembler using a dummy program (Section 2.5.2). This section concludes with a discussion on an optimized microkernel for general matrix multiplication (Section 2.5.3).

2.5.1. Compiling LLVM

The LLVM project uses CMake to generate the Makefiles required for building its libraries and tools, including Clang.

Listing 2.12 shows the command that should be executed in the root directory of the LLVM repository. The first command at line 1 generates the build system. By default, only the LLVM libraries (without Clang) are compiled, and the only backend enabled by default is the architecture of the host, most likely x86 or Arm. Thus, we explicitly enable Clang for the RISC-V backend. The second command on line 6 actually builds the

```
$ cmake -S . -B build \
    -DCMAKE_BUILD_TYPE=Debug \
    -DLLVM_ENABLE_PROJECTS="clang" \
    -DLLVM_TARGETS_TO_BUILD="RISCV"

$ cmake --build build -j
```

Listing 2.12. Compile LLVM with CMake.

libraries and the executable. Due to the size of LLVM, this process can take anywhere from 20 minutes to a few hours, depending on the host hardware specifications.

2.5.2. Assembling and Disassembling

To verify the correctness of the assembler and disassembler, consider a dummy function that simply multiplies register `m1` by `m2` and stores the result in `m0`, as shown in Listing 2.13 (lines 3-4). Clang can be used to assemble this function into object code using the command on lines 7-8. The flag "`-target riscv64`" specifies that RISC-V should be the target architecture, and "`-march=rv64g_xmatrix`" specifies that the target architecture is 64-bits, with the "G" extension enabled (short for "IMAFD" extensions) and also "xmatrix". Finally, we invoke `llvm-objdump` with the `-d` flag to disassemble the previously assembled object file and confirm that it reproduces the original assembly.

```
# Dummy assembly file
$ cat dummy.s
dummy:
    mmacf m0, m1, m2

# Assemble the code into and object file
$ build/bin/clang \
    -target riscv64 -march=rv64g_xmatrix -c -o dummy.o dummy.s

# Disassemble the object file
$ build/bin/llvm-objdump -d dummy.o
dummy.o:        file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 <dummy>:
       0: 6a208077      mmacf   m0, m1, m2
```

Listing 2.13. Assemble and disassemble dummy code using Clang.

2.5.3. Matrix Multiplication Microkernel

OpenBLAS is a software package that implements highly optimized kernels to compute the most common numerical linear algebra routines. It is organized into three levels:

1. The first level includes operations on vectors, such as vector scaling (`SCAL`), dot product (`DOT`), and vector normalization (`NRM2`).

2. The second level gathers matrices through vector operations, such as general matrix-vector multiplication (`GEMV`) and general rank update (`GER`).

3. The third and last level includes matrix by matrix operations, with the most important being the general matrix-matrix multiplication (`GEMM`).

From all the routines available, matrix multiplication is arguably the most ubiquitous operation. It is used in applications across numerous domains, such as engineering, geophysics, and deep learning. In particular, the essence of Large Language Models (LLMs) is several matrix multiplication kernels interspersed with other operations, such as activation functions (ReLU, GeLU, etc.) and tensor normalization (LayerNorm, BatchNorm, etc). Therefore, being able to speed up this routine using hardware-level acceleration, such as the matrix extension that is being developed in this course, is of utmost importance to enable inference locally on custom built RISC-V hardware.

The GEMM kernel, however, not only multiplies two matrices but also accumulates the result into a third matrix. In practice, given matrix A with size $m \times k$, matrix B with size $k \times n$, and matrix C with size $m \times n$, it computes:

$$C = \alpha AB + \beta C \tag{1}$$

where α and β are real-valued scalars that control the sign and scaling of the terms being added. The BLAS package implements several versions of each routine for different data-types. Thus, we look at `SGEMM` which operates on matrices with single-precision floating-point elements.

To improve cache locality, the kernel computes the result matrix in tiles of fixed size, which is usually tied to the size and number of registers available. The tiles are first laid out contiguously in memory in a separate buffer, an operation called packing. For reasons that will become clear later, matrix A tiles are also transposed during packing. Then, tiles from A and B are multiplied together and accumulated into the corresponding tile in C. We shall call this operation on individual tiles a "micro-kernel". A micro-kernel with a tile size of 4x4 is presented below.

Listing 2.14 shows how this can be implemented in Assembly language, assuming $\alpha = 1$ and $\beta = 0$. First, the accumulator matrix register `m0` is cleared (line 2), and the index variable on k is initialized in scalar register `t0` (line 3). Register `t1` stores the size of a single row from a matrix register in bytes, which equals 16 (line 4). Inside the loop beginning at line 7, one matrix register is loaded from A (lines 8 and 9), and another from B (lines 10 and 11). Both registers are multiplied and accumulated into `m0` (line 13). The loop induction variable is incremented (line 14), and the loop exits if it is greater than or equal to k; otherwise, it iterates again (line 14). The micro-kernel ends by writing the accumulator to memory (line 17) and returning.

Even though the kernel shown above is correct, it is possible to hide memory latencies by having loads, stores, and MACs happen concurrently. The idea is to make better use of the processor pipeline by dispatching as many operations as possible so that all functional units are busy at the same time. This can be achieved by choosing

```
usgemm_naive:
    mzero m0
    li t0, 0            # i = 0
    li t1, 16           # Row size (bytes)
    sll a0, a0, 2       # K *= sizeof(float)

.loop:
    add t2, a3, t0      # A + i
    mls m20, (t2), a6
    add t3, a2, t0      # B + i
    mls m25, (t3), a5
    mmacf m0, m20, m25
    add t0, t0, t1      # i += 4
    blt t0, a0, .loop   # if i >= K, exit loop

.loop.end:
    mss m0, (a1), a4
    srl a0, a0, 2       # K /= sizeof(float)
    ret
```

Listing 2.14. Naive outer product micro-kernel.

a tile larger than the register size. Consider the two mutually exclusive strategies for implementing a highly-parallel GEMM micro-kernel. The first step computes the inner product between a row i of A and a column j of B, and accumulates it in element i, j of C. Note that one element of C is computed at a time. The second approach, however, computes the outer product between column i of A and row j of B, yielding a new matrix of size $m \times n$ that must be accumulated into C. No solution is better than the other in general. Choosing the best approach depends on the details of each architecture. For this RISC-V matrix extension in particular, we adopt an outer product micro-kernel.

Figure 2.9 introduces the layout of a single tile of each matrix in a GEMM operation. The idea is to choose the largest tile size of C such that all temporaries can be kept in registers rather than spilled to memory. That way, one can load 4 matrix registers from A along the m dimension, and 6 matrix registers from B along the n dimensions, resulting in $4 \cdot 6 = 24$ outer products, each stored in their own register. It sums up to a total of 34 registers, which is larger than the 32 available. However, it is possible to allocate only two registers that hold elements from A and reuse them accordingly. This brings the total to exactly 32 registers. Therefore, we choose a tile size of 16x24 elements, grouped in registers of 4x4 elements. This way, tile $C_{i,j}$ is the result of the operation $A_j B_i + C_{i,j}$.

The assembly source code for the improved micro-kernel is too large to be reproduced in full therefore we present the pseudo-code in Listing 2.15. The full code is available online[3]. We start by setting a constant `ROW_LEN` to be equal to the number of elements of a single row in a matrix register (4). Since this is an outer-product kernel, the outermost loop iterates over the k dimension (line 2). Next, iteration occurs inside the tile for matrix C, going over elements `acc[i,j]` that correspond to elements $C_{i,j}$ from Figure 2.9 (lines 3-4). The body of the loop nest loads a matrix register from A (`ma`), a matrix

[3] https://gist.github.com/leiteg/a5f4eb1fdd3fc7f2cca947b8c7e2be77

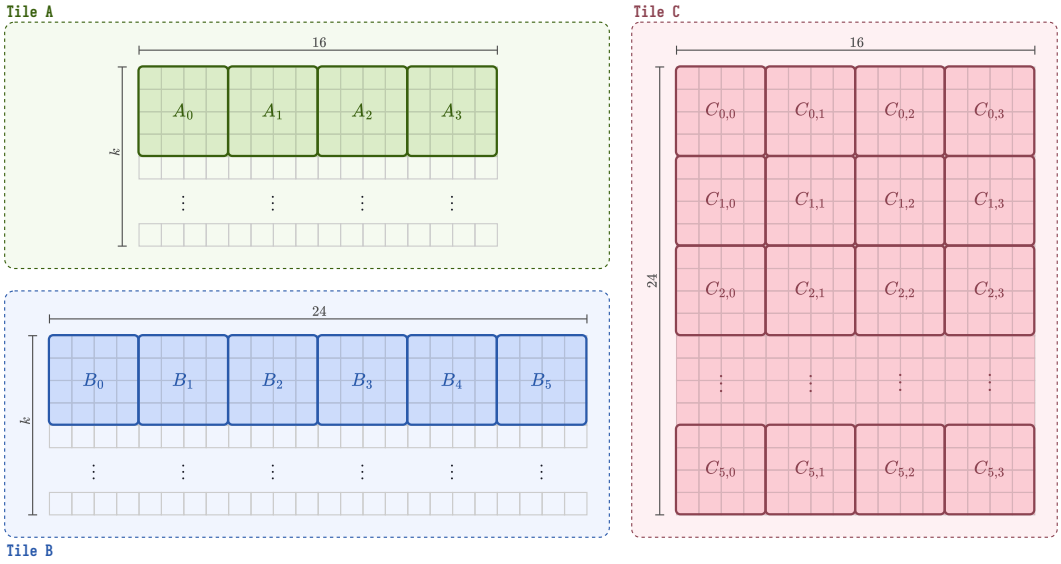

Figure 2.9. Outer product micro-kernel using the RISC-V Matrix extension (Tile A is green, tile B is blue, and tile C is red).

```
ROW_LEN = 4
for l in range(k, step=4):
    for j in range(6): # unroll
        for i in range(4): # unroll
            ma = mls(A[k * 16 + i * ROW_LEN], ROW_LEN)
            mb = mls(B[k * 24 + j * ROW_LEN], ROW_LEN)
            acc[i, j] = mmac(ma, mb, acc[i, j])
for j in range(6): # unroll
    for i in range(4): # unroll
        mss(acc[i, j], C[j * ldc + i * ROW_LEN], ldc)
```

Listing 2.15. Pseudo-code for the improved micro-kernel.

register from *B* (`mb`), multiply them together and accumulates on the corresponding accumulator register that is assumed to be initialized with zero (lines 5-7). As discussed above, there are enough registers to compute all iterations of the two innermost loops without the need to spill to memory. For this reason we unroll the loops on `i` and `j`. At this point, the outer product of matrices *A* and *B* is computed all that is left is storing the values back to memory (lines 8-12). This loop nest is also unrolled for improved performance.

2.6. Concluding Remarks

This tutorial has presented how to implement a matrix processing extension for RISC-V using the LLVM compiler infrastructure. The resulting toolchain allows programmers to write kernels in assembly that use matrix instructions and assemble them into object code. Programmers can also disassemble an object file back into instructions in textual format.

Compilers are often treated as opaque black boxes and many programmers assume that experimenting with them is too difficult and not worthwhile. It is true that compilers are generally large and complex to understand holistically. However, in practice, much software engineering is utilized to tame the complexity and keep the design modular. This modularity makes it possible for curious learners to peer into the individual stages of the compilation process and develop a deeper understanding incrementally. The objective of this course was precisely to show that extending a compiler is both feasible and conceptually approachable when done methodically.

2.6.1. Next Steps

Hardware development naturally progresses at a slower pace than software development. Before tape-out, processors must undergo extensive testing through simulators and FPGAs, as hardware bugs cannot be fixed once the circuit is fabricated. Software development, on the other hand, is incremental and iterative. Bugs can be patched, features refined, and new versions deployed with relative ease. Therefore, a good next step is to implement a simulator capable of executing the new matrix instructions. Free and open-source projects like QEMU[4] and GEM5[5] provide a good starting point for implementing emulators.

But the work inside the compiler itself rarely stops at the assembler. Programmers prefer to write their computational kernels in a high-level language like C/C++. The interested student can use the Clang C/C++ compiler shipped with the LLVM project and extend it with new built-in functions. In general, for each new instruction, there is a corresponding built-in function that, once called in C, becomes a single instruction in the executable. However, the process is not so straightforward. Remember that the code is first transformed into LLVM IR before being compiled into machine code. Thus, it is necessary to add new intrinsic operations to the IR to bridge the gap between built-ins and the final instructions. Both built-ins and intrinsics are specified using the already familiar TableGen syntax. In this case, however, the programmer is required to write some additional C++ code to define how these representations are translated within the compiler.

[4] https://www.qemu.org/
[5] https://gem5.org