6

RISC-V Assembly Language

In this chapter, we will introduce the RISC-V assembly language, which is newer than ARM and x86. The ISA has a very interesting history. There are both legal and technical reasons for designing a new RISC ISA as late as 2010. Recall that ISA design was considered to be a dead area long before. Furthermore, a massive SW-HW ecosystem is needed to sustain an ISA, which is hard to design from scratch. Nevertheless, there was a requirement in 2010 for a RISC ISA that could be freely used by everybody and incorporated a lot of the technical know-how that had been generated in the past two decades of computer architecture research.

Any modern ISA should be compatible with all kinds of devices starting from IoT devices to mobile phones to laptops to servers. Many of these devices didn't exist when conventional ISAs were designed. This means that it should have 32, 64 and 128-bit variants, support a relatively larger number of registers, and have extensive support for atomics and floating point numbers.

Let us understand the situation that prevailed in 2010. Designing a simple RISC processor had become relatively very easy. The architectures were well understood, EDA tools were reasonably mature and enough computational power was available to even amateur designers. Hence, putting together a small RISC core became feasible for fabless companies and academic groups. It is important to note that circa 2010, there was a widespread consensus that RISC is the way to go for new cores and x86-like CISC ISAs were not an option. This is because of the 1000+ instructions in CISC ISAs and the resulting decoding complexity. Decoders are power hungry and given that these ISAs were meant to run on low-power embedded systems, a CISC ISA was not a feasible option.

However, just designing a RISC processor is not enough. It needs to be fabricated as well. A lot of chip fabrication companies (fabs) increased their capacity at that time, and it also became possible to design reasonably high-performance chips at older technology nodes. Hence, fabrication became quite inexpensive. Along with these technology-level tailwinds, many fabless companies rapidly came up in different parts of the world. They either wanted to design their bespoke processors or wanted to integrate their custom cores with third-party accelerators

in SoCs. They always had an option to use or modify existing RISC cores designed by ARM and MIPS or design their own processors that are compatible with their ISAs. The advantage of using existing technologies is that their toolchain can be used. This includes the compilers, operating systems, libraries and *binutils*.

Despite so much of software support, many processor developers decided to forego the advantages of using existing ISAs primarily for legal reasons. There were a fair amount of restrictions and licensing requirements for using such legacy ISAs while even designing new processors from scratch. Furthermore, licensing their cores and using them in SoCs was also proving to be an expensive proposition to many developers. There was thus a need to create a new ISA from scratch and also create the full ecosystem to support it. The idea was to have extremely lenient licensing requirements such that the barrier to entry is reduced to a minimum.

In 2010, the RISC-V project began in Berkeley. It was initially supported by various academic groups. Later on the RISC-V technical documents were released under a Creative Commons license in 2015. Currently, the RISC-V foundation maintains this ISA and publishes regular updates.

6.1 RISC-V Machine Model

6.1.1 RISC-V Base ISAs and Extensions

RISC-V is not actually one instruction set, but it is a collection of instruction sets that incrementally build on top of each other. There are a set of baseline versions (common cores). Developers build on top of a common core by adding extensions. Given that the philosophy of RISC-V is to run on all kinds of machines, the notion of having different core ISAs and extensions aligns with it well. Let us look at the common base ISAs and extensions in Table 6.1.

Name	Description
ISA base versions	
RV32	32-bit ISA
RV32E	32-bit ISA (embedded version)
RV64	64-bit ISA
RV64E	64-bit ISA (embedded version)
RV128	128-bit ISA
Extensions	
E	embedded version
I	base integer ISA
M	integer multiplication/division instructions
A	atomic instructions
F	single-precision floating point
D	double-precision floating point
V	vector instructions

Table 6.1: RISC-V base versions and extensions

RISC-V has three base versions: RV32, RV64 and RV128. The term "RV" is a short form

of RISC-V. The numbers 32, 64 and 128 indicate the bit width, respectively. The 'E' suffix has a special place. It indicates the embedded version that uses a reduced number of registers. For example, RV32E assumes only 16 integer registers as opposed to 32 integer registers in the regular version. It is important to note that unlike other ISAs (such as ARM Thumb), the instruction sizes remain the same. This simplifies compiler and processor design.

Let us now focus on the extensions. Most versions of the ISAs support basic integer instructions. They are thus named RV32I, RV64I and RV128I, respectively. Then, there are a bunch of extensions that can be added based on users' requirements. The list of extensions has become quite large as of 2024 (around 20-30). Most of the common ones correspond to floating point instructions, atomic operations, cryptographic primitives, memory barriers, etc. For example, RV32IMA means that integer instructions (I), multiplication/ division instructions (M) and atomic instructions (A) are supported.

These extensions themselves can have version numbers: a major version number and a minor version number. This is because the specifications keep changing as the ISAs are under development. For example, RV32I1p3 means that the major version number is 1 and the minor version is 3. The separator 'p' is used to separate the major and minor version numbers.

The extensions can be grouped into packages the same way we bundle together add-ons in flight or hotel deals. The 'G' suffix that represents general-purpose computing, combines the base integer instruction set, additional integer instructions, floating point instructions and basic synchronization primitives. This is considered to be an essential set of instructions in a multi-core setup. RV32G is thus a general-purpose RISC-V ISA. It is so happening that the number of extensions is continuously increasing, and we are running out of letters !!!

This is why the 'Z' series was introduced, where the extension name (suffix) should start with 'Z' and be followed by a word that describes the extension. For example, 'Zfa' refers to additional floating point instructions.

Let us now recall the fact that the embedded version of RISC-V does not reduce the instruction width; it instead reduces the number of available registers by 50%. On the lines of ARM Thumb, RISC-V does have a compressed format. In this case, the ISA specifier is 'C'. In such contexts, most often we use the ISA RV32GC (general-purpose and compressed). The compressed instructions have the following limitations.

- The width of the instructions is 16 bits.
- Every compressed 16-bit instruction corresponds to a 32-bit RV32 instruction.
- They access a limited number of registers (typically limited to 8 registers in the 16-bit version).
- Limited opcode support.
- Immediate values that can be encoded are also commensurately smaller.

Akin to any other compressed ISA, they lead to reduced code size, better usage of the i-cache and lower power consumption in terms of fetching and decoding instructions.

32, 64 and 128-bit Formats

The greatness of RISC-V is that regardless of the width of the data path, the instruction size (or width) remains the same, i.e., 32 bits. This makes the process of compilation and decoding the instructions easy. The only thing that varies across the three different ISA formats is the set of supported instructions, the register width and the size of the memory address.

However, there are some exceptions to this general rule. There are some RISC-V instructions whose size can be more than 32 bits. The restriction here is that the size needs to be a multiple of 16 bits – it cannot be an arbitrary size. We have already seen the C-format instructions that are 16 bits in length. However, we can have longer formats: 48 or 64 bits. Such instructions (at the moment) are not a part of the standard set of instructions, but there are extensions that require these longer instructions. Some of these extensions are vector instructions, bit-manipulation instructions and cryptographic extensions. The instruction format has bits to indicate if the instruction is longer than 32 bits or not. The current version of the standard (v20240411) [ris,] can theoretically support instruction lengths up to 176 bits.

We need to note that at the moment (in 2024) such extensions are in different stages of ratification. For example, there is discussion of ratifying a more general set of 48-bit and 64-bit instructions. Similarly, the 128-bit format of the ISA is also not fully finalized yet. In other words, the standards are not fully frozen yet.

Such debates during the course of ISA design are very common. There is always a pull between the RISC and CISC sides. The RISC side wants regularity and elegance, whereas the CISC side wants more instructions and more complexity.

Important Point 7 *The RISC-V ISA is a RISC ISA. It is however not small and simple like SimpleRisc . Instead, it has a base set of instructions, a set of extensions, 48 and 64-bit instruction lengths (may get fully frozen in the future) and different ISA variants including a compressed 16-bit form. This sounds more like "CISC". However, there is still a lot of regularity in the ISA: there are a few instructions formats, instructions are mostly 32 bits in length and the base ISA is very "RISC like". Such trade-offs are inevitable in designing any modern ISA that needs to support a wide range of devices: embedded processors to supercomputers.*

6.1.2 View of Registers

RV32I contains 32 registers that are each 32-bits wide. The registers are named $x0$ to $x31$. $x0$ is hardwired to zero (refer to Table 6.2). There is a dedicated *pc* register that exposes the program counter. The architecture per se does not define a fixed calling convention. However, over time a convention has emerged and developers are mostly using the calling convention shown in Table 6.2. Note that *saved registers* are preserved across function calls (callee saved), whereas temporary registers (caller saved) are not preserved.

RISC-V uses a standard memory addressing model (similar to other RISC machines). It started out being a little-endian ISA, but now big-endian and bi-endian modes are also supported.

Register	Mnemonic	Description
$x0$	zero	Hard-wired to zero
$x1$	ra	Return address
$x2$	sp	Stack pointer
$x3$	gp	Global pointer
$x4$	tp	Thread pointer (thread-local storage)
$x5-7$	t0-2	Temporary registers
$x8$	s0/fp	Saved register/frame pointer
$x9$	s1	Saved register
$x10-11$	a0-1	Function arguments/return values
$x12-17$	a2-7	Function arguments
$x18-27$	s2-11	Saved registers
$x28-31$	t3-6	Temporary registers

Table 6.2: RISC-V registers and their assembler mnemonics

Given the machine model, let us now explain the instructions supported by the RV32I ISA. Note that this is our fourth chapter on low-level assembly languages. Hence, the treatment will be brief.

6.2 Integer Instructions

While describing the semantics of instructions, we use the same convention as ARM and *SimpleRisc* . $rs1$ is the first source register, $rs2$ is the second source register, rd is the destination register and imm represents an immediate value. In some cases, we may require to use a third source register $rs3$.

6.2.1 Moving Values to Registers

Semantics	Example	Explanation
addi rd, rs1, imm	addi x1, x0, 5	x1 ← 0 + 5
add rd, rs1, rs2	add x1, x2, x3	x1 ← x2 + x3

Table 6.3: Loading values into registers

The most basic operation in any assembly language is to load a value into a register. We typically transfer the contents from another register or from the immediate field of an instruction. We need a counterpart of the *mov* instruction in RISC-V. The relevant instructions are shown in Table 6.3. RISC-V does not have a dedicated *mov* instruction; instead, we add an immediate to the *zero* register and store the result in the destination register.

Specifically, the *addi* instruction can be used to load a signed 12-bit immediate to the destination register. In this case, its usage is somewhat unconventional. As the example shows, the immediate is added to the contents of $x0$ (*zero* register), which is hardwired to 0. Effectively,

the immediate gets transferred to the destination register. An advantage of such a mechanism is that we need not have a dedicated *mov* instruction. The *add* instruction and its variants can be used to load immediates. Similarly, we can set the immediate to 0 and transfer the contents of the source register to the destination register. This simulates a regular register *mov* instruction. We can alternatively use the regular *add* instruction to achieve this. We can set the second register operand to *zero*. The net effect is that the contents are transferred to the destination register. The *add* instruction otherwise does the same as its counterparts in ARM and *SimpleRisc* .

Loading Values Directly into Registers

A major issue with the *addi* instruction is that the immediate is limited to 12 bits. Loading a full 32-bit value thus requires several instructions. RISC-V therefore provides the *lui* instruction that loads a 20-bit immediate into the upper 20 bits of a register – the immediate is effectively left-shifted by 12 positions. The semantics of this instruction is shown in Table 6.4 (also refer to Example 91).

Semantics	Example	Explanation
lui rd, imm	lui x1, 5	x1 $\leftarrow$ 5 << 12
li rd, imm	li x1, 0xABCD1234	x1 $\leftarrow$ 0xABCD1234

Table 6.4: Loading values directly into registers

Example 91
Write a RISC-V assembly program to add 409932 + 409823.
Answer:

```
.main:
    lui  t0, 100          # t0 = 4096 * 100 = 409600
    addi t0, t0, 332      # t0 = t0 + 332
    lui  t1, 100          # t1 = 4096 * 100 = 409600
    addi t1, t1, 223      # t1 = t1 + 223
    add  t2, t0, t1       # t2 = t0 + t1
```

It is evident from Example 91 that loading a 32-bit value into a register requires two instructions. Even though the ISA has this limitation, most RISC-V assemblers support the assembler directive *li* that directly loads a 32-bit value into a register. The assembler replaces the directive with two assembly instructions: *addi* and *lui*. The code in Example 91 can be compressed using the *li* assembler directive (refer to Example 92).

Example 92
Write a RISC-V assembly program to add 409932 + 409823 using the li assembler directive.
Answer:

```
.main:
    li   t0, 409932      # t0 = 409932
    li   t1, 409823      # t1 = 409823
    add  t2, t0, t1      # t2 = t0 + t1
```

6.2.2 Add and Subtract Instructions

Semantics	Example	Explanation
add rd, rs1, rs2	add x1, x2, x3	x1 ← x2 + x3
addi rd, rs1, imm	addi x1, x2, 5	x1 ← x2 + 5
sub rd, rs1, rs2	sub x1, x2, x3	x1 ← x2 - x3

Table 6.5: Arithmetic instructions: add and subtract

Table 6.5 shows the general form of the *add* and *sub* instructions in RISC-V. They have the same general format as *SimpleRisc add* and *sub* instructions, respectively. The generic format is *inst rd, rs1, rs2/imm.*

Example 93
Write a RISC-V assembly program to compute 4 + 5 - 19.
Answer:

```
    addi t0, zero, 4      # load 4 into t0
    addi t1, zero, 5      # load 5 into t1
    add  t2, t0, t1       # t2 = t0 + t1
    addi t2, t2, -19      # subtract 19 from t2
```

Example 94
Write an assembly program to swap two numbers stored in $x1$ and $x2$.

Answer:

```
    add x3, x0, x1      # x3 = x1
    add x1, x0, x2      # x1 = x2
    add x2, x0, x3      # x2 = x3 (old x1)
```

6.2.3 Multiplication and Division Instructions

Semantics	Example	Explanation
mul rd, rs1, rs2	mul x1, x2, x3	x1 ← x2 × x3
div rd, rs1, rs2	div x1, x2, x3	x1 ← x2 / x3
rem rd, rs1, rs2	rem x1, x2, x3	x1 ← rem (x2 / x3)

Table 6.6: Multiplication and division instructions

Table 6.6 shows the multiplication and division instructions. They are a part of the 'M' extension. The reason for including them in an extension is to enable the creation of really low-end and low-power implementations that do not require such instructions.

The multiplication instruction has some complications. The product requires 64 bits, which means that it will not fit in a single register. The default implementation thus places the lower 32 bits in the destination register. However, sometimes there is a need to store the full 64-bit product – this will require two registers. The default *mul* instruction computes the lower 32 bits. The *mulh* and *mulhu* instructions can next be used to store the upper 32 bits for signed × signed and unsigned × unsigned multiplication, respectively. Even though we require two separate instructions now, micro-architectures can fuse them dynamically. They can identify two consecutive multiplication instructions where one instruction computes the lower 32 bits and the next instruction computes the upper 32 bits. This sequence can be identified dynamically, and a single multiplication will only be required.

Example 95
Write an assembly program to multiply 3 with -17 and save the result in t3.

Answer:

```
addi t1, zero, 3        # t1 = 3
addi t2, zero, -17      # t2 = -17
mul  t3, t1, t2         # t3 = t1 * t2
```

Example 96
Compute 12^3 *+ 1 and save the result in t4.*

Answer:

```
# load the registers with required values
addi t1, zero, 1    # t1 = 1
addi t2, zero, 12   # t2 = 12
addi t3, zero, 12   # t3 = 12
```

```
#perform the arithmetic operations
mul t3, t2, t2     # t3 = 12 * 12
mul t3, t3, t2     # t3 = 12 * 12 * 12
add t4, t3, t1     # 12^3 + 1
```

The division instruction *div* is comparatively simpler. In the RV32 variant, it requires 32-bit dividends and divisors. The quotient is stored in the destination register. The rounding is *towards zero*. Let us explain rounding using a few examples.

Division operation	Quotient	Remainder
$4 \div 3$	1	1
$4 \div (-3)$	-1	1
$(-4) \div 3$	-1	-1
$(-4) \div (-3)$	1	-1

We see that rounding towards zero also means that the sign of the remainder is the same as the sign of the dividend. The remainder instruction works on similar lines. It computes the remainder of the division operation (rounding towards zero).

Akin to the multiplication operations, the division and remainder operation work in the same manner. When they are issued back to back, micro-architectures are expected to *fuse* them. They compute a single division operation and store the results in two registers – one register for the quotient and one for the remainder, respectively.

This is an example of a scenario where the ISA has deliberately been under-designed. Instead of having an instruction that stores to two 32-bit registers, the programmer or compiler are expected to invoke these instructions consecutively. It is the job of the hardware to dynamically identify such sequences and *fuse* them. This transfers the responsibility of ensuring efficiency to hardware at the cost of keeping the ISA simple.

Example 97
Write a RISC-V assembly program to divide -50 by 3. Store the quotient in t2 and remainder in t3.

Answer:

```
addi t0, zero, -50  # t0 = -50
addi t1, zero, 3    # t1 = 3
div t2, t0, t1      # quotient in t2
rem t3, t0, t1      # remainder in t3
```

Semantics	Example	Explanation
and rd, rs1, rs2	and x1, x2, x3	x1 ← x2 AND x3
andi rd, rs1, imm	andi x1, x2, 6	x1 ← x2 AND 6
or rd, rs1, rs2	or x1, x2, x3	x1 ← x2 OR x3
ori rd, rs1, imm	ori x1, x2, 9	x1 ← x2 OR 9
xor rd, rs1, rs2	xor x1, x2, x3	x1 ← x2 XOR x3
xori rd, rs1, imm	xori x1, x2, 7	x1 ← x2 XOR 7
sll rd, rs1, rs2	sll x1, x2, x3	x1 ← x2 << x3
srl rd, rs1, rs2	srl x1, x2, x3	x1 ← x2 >> x3
sra rd, rs1, rs2	sra x1, x2, x3	x1 ← x2 >>> x3
slli rd, rs1, imm	slli x1, x2, 3	x1 ← x2 << 3
srli rd, rs1, imm	srli x1, x2, 3	x1 ← x2 >> 3
srai rd, rs1, imm	srai x1, x2, 3	x1 ← x2 >>> 3

Table 6.7: Logical and shift instructions

6.2.4 Logical and Shift Instructions

Table 6.7 shows a list of some prominent logical and shift instructions. The primary logical instructions are *and*, *or* and *xor*. We can attach an 'i' suffix to these instructions to accept an immediate value as the second source operand. The format is otherwise the same as the *add* and *sub* instructions.

Example 98
Write a RISC-V assembly program to compute the bitwise OR of A and B. Let $A = 4$ and $B = 1$.

Answer:

```
addi t1, zero, 4   # t1 = 4
ori  t2, t1, 1     # bitwise OR of 4 and 1
```

Akin to other ISAs, RISC-V has three shift instructions: shift left logical (*sll*), shift right logical (*srl*) and shift right arithmetic (*sra*). They have their variants where the second source is an immediate. They are *slli*, *srli* and *srai*, respectively.

Example 99
Write RISC-V assembly code to compute 50/4.

Answer:

```
addi t0, zero, 50     # t0 = 50
srai t1, t0, 2        # t1 = 50/4
```

Example 100

Write RISC-V assembly code to compute $t1 = t2 + t3 \times 4$.

Answer:

```
addi t3, zero, 5      # t3 = 5
addi t2, zero, 7      # t2 = 7
slli t4, t3, 2        # t4 = t3 * 4
add t1, t2, t4        # t1 = t2 + t3 * 4
```

6.3 Control Transfer Instructions

6.3.1 Conditional Branches

Unlike *SimpleRisc* and ARM, RISC-V does not have a *flags* register that stores the result of the last comparison. The arguments for the comparison are typically specified directly in the branch instruction itself along with the branch target.

Set-less-than (slt) Instruction

However, sometimes there is a need to store the result of a comparison. RISC-V thus provides a flexible mechanism to achieve this. In a conventional RISC ISA, the *flags* register is implicit, whereas it is more explicit in RISC-V. Such a class of instructions is shown in Table 6.8.

Semantics	Example	Explanation
slt rd, rs1, rs2	slt x1, x2, x3	if (x2 < x3) set x1 to 1
slti rd, rs1, imm	slt x1, x2, 5	if (x2 < 5) set x1 to 1
sltu rd, rs1, rs2	slt x1, x2, x3	if (x2 $<_{unsigned}$ x3) set x1 to 1
sltui rd, rs1, imm	slt x1, x2, 5	if (x2 $<_{unsigned}$ 5) set x1 to 1

Table 6.8: The *slt* family of instructions. The destination register is by default set to 0.

Table 6.8 shows the *slt* family of instructions. They compare the values of two registers, or a register and an immediate. If the first source operand is less than the second source operand, then the destination register's value is set to 1. Otherwise, it remains 0. The conditional branch

instructions can then directly compare this register with *zero* and decide the outcome of the branch instruction: taken or not-taken.

Example 101
Write RISC-V assembly code to set $t2$ if $2 < 5$.

Answer:

```
addi t0, zero, 2      # t0 = 2
addi t1, zero, 5      # t1 = 5
slt t2, t0, t1        # t2 = (t0 < t1)
```

Example 102
Add two long 64-bit values stored in $\langle t1, t0\rangle$ and $\langle t3, t2\rangle$. Store the result in $\langle t5, t4\rangle$.
Answer:

```
# initialize the registers
addi t2, zero, -1
addi t3, zero, 2
addi t0, zero, 1
addi t1, zero, 0

# add <t5,t4> = <t1,t0> + <t3,t2>
add t4, t0, t2     # add lower 32 bits
add t5, t1, t3     # add upper 32 bits
sltu t6, t4, t0    # t6 stores the carry

add t5, t5, t6     # add the carry
```

Branch Instructions

Semantics	Example	Explanation
beq rs1, rs2, label	beq x1, x2, .foo	Branch to the . foo label if x1 = x2
bne rs1, rs2, label	bne x1, x2, .foo	Branch to the .foo label if x1 $\neq$ x2
bge rs1, rs2, label	bge x1, x2, .foo	Branch to the .foo label if x1 $>$ x2
blt rs1, rs2, label	blt x1, x2, .foo	Branch to the .foo label if x1 $<$ x2
bgeu rs1, rs2, label	bgeu x1, x2, .foo	Similar to bge, considers unsigned values.
bltu rs1, rs2, label	bltu x1, x2, .foo	Similar to blt, considers unsigned values.

Table 6.9: Conditional branch Instructions

The conditional branch instructions in RISC-V are shown in Table 6.9. The instructions take two register arguments and compare them. The result of the comparison is immediately used to decide the direction of the branch.

Table 6.9 shows the *beq*, *bne*, *bge* and *blt* instructions that have their usual meanings. The third argument is a label that represents the branch target. Along with these signed comparison instructions, RISC-V has comparison instructions to compare unsigned integers: *bgeu* and *bltu*. Recall that ARM also has similar instructions that are implemented with the help of custom flags.

Example 103

Write a RISC-V assembly program to compute the factorial of a positive number (> 1) stored in a1. Save the result in a0.

Answer:

```
.main:
      addi a0, zero, 1       # prod = 1
      addi t0, zero, 1       # index = 1
.loop:
      mul a0, a0, t0       # prod = prod * index
      addi t0, t0, 1       # index ++
      bge a1, t0, .loop    # loop condition

      # a0 stores the factorial
```

Example 104

Write an assembly program to add the numbers from 1 to 10. Store the result in s0.

Answer:

```
.main:
      addi t0, zero, 1      # initialize t0 to 1
      addi s0, zero, 0      # result (s0) = 0
      addi t1, zero, 10     # loop end value

.loop:
      add s0, s0, t0        # add to the result
      addi t0, t0, 1        # increment the counter
      bge t1, t0, .loop     # loop condition

# s0 has the sum
```

Example 105

Write an assembly program to test if a number stored in a1 is prime or not. Save the Boolean result in a0.

Answer:

```
   # input in a1, return value in a0
.main:
   addi t0, zero, 2          # starting divisor

.loop:
   rem t1, a1, t0            # find the remainder (t1)
   beq t1, zero, .notPrime

   addi t0, t0, 1            # increment the divisor
   bne t0, a1, .loop         # loop back

   addi a0, zero, 1          # number is prime
   jal x0, .end

.notPrime:
   addi a0, zero, 0

.end:
   # a0 contains the result
```

Example 106

Write an assembly program to find the number of ones in a 32-bit number stored in a1.

Answer:

```
.main:
      addi t0, zero, 0       # counter, t0 = 0
      addi t1, zero, 32      # maximum possible ones
      addi t2, zero, 1       # t2 = 1
      addi a0, zero, 0       # will contain the result (a0 = 0)

.loop:
      andi t3, a1, 1         # check the LSB of the argument a1
      srli a1, a1, 1         # shift the argument by 1 step
      beq  t3, t2, .inc      # jump to .inc if the LSB is 1

.lret:
      addi t0, t0, 1          # increment the counter
```

```
        beq t1, t0, .end        # exit the loop
        jal zero, .loop         # loop back

.inc:
        addi a0, a0, 1          # increment the count of 1s
        jal zero, .lret         # resume the next iteration

.end:
        # a0 contains the result
```

Example 107

Write an assembly program to check if a natural number stored in a1 is a perfect square or not. Save the Boolean result in a0.

Answer:

```
.main:
        # input number in a1
        addi a1, zero, 101
        addi a0, zero, 0     # assuming result (a0) = false
        addi t1, zero, 1     # counter (t0) = 1

.loop:
        mul t2, t1, t1        # square -> compare
        beq t2, a1, .square   # It is a square
        addi t1, t1, 1        # increment the counter
        blt a1, t2, .end
        jal zero, .loop       # loop back

.square:
        addi a0, a0, 1        # result = 1

.end:
    # result in a0
```

6.3.2 Unconditional Branches

The unconditional branch/jump instructions of RISC-V are shown in Table 6.10. The most commonly used instruction is *jal* – it functions both as a function call instruction as well as a regular unconditional jump instruction. In both cases, the control jumps to the PC pointed to by the *label*. Akin to other ISAs, while encoding the instruction, the label is translated to a PC-relative offset. The jump can take place within a region of ± 1 MB. The *jal* instruction

Semantics	Example	Explanation
jal rd, label	jal x1, func	Jump to the func label and store the return address in $x1$
jalr rd, rs1, offset	jalr x1, x2, 20	Jump to the address $x2 + 20$ and store the return address in $x1$

Table 6.10: Jump instructions in x86

additionally stores the return address (pc+4) in the first source register ($x1$ in the example). Note that if the first source register is equal to $x0$ ($zero$), then the return address is not stored. jal in this case acts as a regular unconditional jump that does not store the return address.

The $jalr$ instruction augments jal with one additional register argument. Consider the example: $jal\, x1$, $x2$, 20. In this case, we add the offset 20 to the contents of $x2$ and jump to the resulting address. The return address is stored in $x1$. Similar to jal, we do not store the return address if the first source register is $x0$. The $jalr$ instruction can be used to implement a function return instruction. All that we have to do is to jump to the PC whose value is $0(ra)$ (contents of the register $ra + 0$).

Example 108
Write a RISC-V assembly program that has a function call.

Answer:

Listing 6.1: C code

```
int foo() {
   return 2;
}
void main() {
   int x = 3;
    int y = x + foo();
}
```

Listing 6.2: RISC-V code

```
.foo:     #callee
   addi a0, zero, 2     # a0 = 2
   jalr zero, 0(ra)     # return inst.

.main:
   addi s0, zero, 3     # s0 = 3
   jal ra, .foo         # jump to .foo
   add s1, s0, a0       # y = x + foo()

#  s1 contains the result
```

Example 109
Write a RISC-V assembly program to compute x^n and store the result in a0. x is passed through a1 and n is passed through a2.

Answer:

```
.power:
    addi a0, zero, 1      # a0 will contain the result
    add t1, zero, a2      # t1 = n
    beq t1, zero, .end    # check (n == 0)

.loop:
    mul a0, a0, a1        # result *= x
    addi t1, t1, -1       # decrement n
    bne t1, zero, .loop

    jalr zero, 0(ra)      # return

.main:
    addi a1, zero, 7      # x = 7
    addi a2, zero, 3      # n = 3

    jal ra, .power        # call the power function

# the result is in a0
```

6.3.3 Load and Store Instructions

Semantics	Example	Explanation
lw rd, imm(rs1)	lw x1, 32(sp)	x1 ← mem[sp + 32]
sw rs2, imm(rs1)	sw x1, 32(sp)	mem[sp+32] ← x1
la rd, label	la x1, pi	x1 ← address(pi)

Table 6.11: Load and store instructions. Note that *la* is an assembler directive.

Table 6.11 shows the load and store instructions in RISC-V. We only show the 32-bit versions of these instructions. The *lw* instruction loads 32-bit values from memory that is specified in the base-offset format. On similar lines, the *sw* instruction stores the value of a register to memory. Note that the store instruction takes two register source operands, and it has its separate format. The store operation has always been an exception in such respects. RISC-V defines a special format for it, which accepts two register-based source operands and an immediate.

Example 110
Write an assembly program to load $a0$ with the contents of the memory address $sp - s0 \times 4 - 12$.

Answer:

```
.main:
    slli s0, s0, 2      # s0 = s0 * 4
    add s0, s0, 12      # s0 = s0 + 12
    sub t0, sp, s0      # t0 = sp - s0
    lw a0, 0(t0)        # load the value of mem[t0] in a0
```

Example 111
Write an assembly program to create a copy of a 10-element array. Assume the starting address of the original array is stored in $a1$ and that of destination array is stored in $a2$.

Answer:

```
.main:
    addi t1, zero, 0    # counter(t1) = 0
    addi t2, zero, 10   # number of iterations

.loop:
    lw t0, 0(a1)        # load an element from the source array
    sw t0, 0(a2)        # store an element in the destination array

    addi a1, a1, 4      # get the address of the next element: src array
    addi a2, a2, 4      # destination array
    addi t1, t1, 1      # increment the counter
    bne t1, t2, .loop   # loop back
```

Example 112
Write a RISC-V assembly program to compute the sum of the elements in a 10-element array. Assume that the base address of the array is stored in $a1$. Store the result in $a0$.

Answer:

Listing 6.3: C code

```
void addNumbers(int a[10]) {
```

```
    int idx;
    int sum = 0;
     for (idx = 0; idx < 10; idx++){
         sum = sum + a[idx];
     }
}
```

Listing 6.4: RISC-V code

```
.main:
    addi t0, zero, 0      # index = 0
    addi a0, zero, 0      # result = 0
    addi t1, zero, 10     # limit = 10

.loop:
    lw t2, 0(a1)      # load an element in t2
    add a0, a0, t2    # update the result

    addi a1, a1, 4    # traverse the array
    addi t0, t0, 1    # index ++
    bne t0, t1, .loop

 # result in a0
```

Example 113
Write a RISC-V assembly program to compute the factorial of a number (stored in a1) using recursion. Store the result in a0.

Answer:

```
.fact:
    # check if n (in a1) is 0 or 1
    addi t1, zero, 1     # t1 = 1
    bge t1, a1, .ltone   # if (a1 == 1) jump to .ltone

    # need to make a recursive call
    add t0, a1, zero     # t0 = a1 (=n)
    addi a1, a1, -1      # a1 = n - 1

    # store the state
    addi sp, sp, -8      # sp = sp - 8
    sw ra, 0(sp)         # store ra and t0
    sw t0, 4(sp)         # on the stack

    # recursive call
```

```
    jal ra, .fact

    # restore the state of the stack
    lw t0, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 8

    # compute the result
    mul a0, a0, t0      # fac(n) = n * fac(n-1)
    jalr zero, 0(ra)    # return

.ltone:
    addi a0, zero, 1    # result is 1
    jalr zero, 0(ra)    # return

.main:
   addi a1, zero, 5     # compute 5!
   jal ra, .fact        # Call the factorial function
   # result in a0
```

The *la* Assembler Directive

There is often a need to load values to memory before a program starts to execute. A need arises when we use built-in constants and initialize global or static variables. Using the *li* instruction, it is always possible to load 32-bits to a given memory address. However, it is possible to design a more elegant solution that in practice will translate to multiple assembly instructions. It will nevertheless make the job of the assembly programmer much easier. The *la* directive achieves this.

Let us consider an example. A constant *val* needs to be defined as a label. The specific way of defining it is as follows: *val: .word 17*. A 32-bit integer constant requires the *.word* directive and a floating point constant requires the *.float* directive. It is then succeeded by the value of the constant.

The *la* directive can be used to load the address of a constant into a register. Subsequently, a regular load instruction can be used to read the value of the constant (refer to Example 114).

Example 114
Define a constant val that is initialized to 17. Store its value in register s0 after loading it from memory.

Answer:

```
val: .word 17

la t1, val
lw s0, 0(t1)
```

6.4 Floating Point Instructions

Let us now look at the floating point instructions in RISC-V. In 2017, the 'F' and 'D' extensions were introduced for single precision and double precision floating point operations, respectively. This part of the instruction set is quite conventional and is similar to other RISC ISAs. Floating point numbers are stored in the regular IEEE 754 format.

6.4.1 View of Registers

The RISC-V floating point model has 32 floating point registers. Their names range from $f0$ to $f31$. Unlike integer registers, no register is hardwired to 0. There is however a register usage convention (akin to integer registers).

Register	Mnemonic	Description
f0-7	ft0-7	Temporary registers
f8-9	fs0-1	Saved registers
f10-11	fa0-1	Arguments/return values
f12-17	fa2-7	Function arguments
f18-27	fs2-11	Saved registers
f28-31	ft8-11	Temporary registers

It is not possible to directly load an immediate into a floating point register. Like x86, floating point registers can only be initialized by loading values from memory.

Floating Point Control and Status Register

There is an additional special register called the *floating point control and status register* ($fcsr$), whose structure is shown in Figure 6.1. Its lower 8 bits encode important information. The first 5 bits starting from the LSB store exceptional conditions encountered since these bits were last reset. This is known as the $fflags$ field. The rest of the 3 bits store the rounding mode.

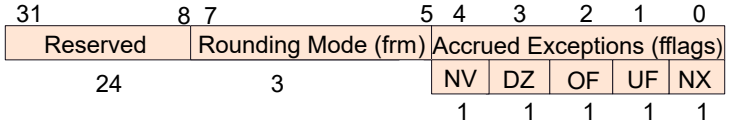

Figure 6.1: RISC-V $fcsr$ register

Accrued Exception Flags ($fflags$)

Mnemonic	Explanation
NV	Invalid operation
DZ	Divide by zero
OF	Overflow
UF	Underflow
NX	Inexact

Table 6.12: Accrued exception flags

The $fflags$ field stores five flags, which are also known as the accrued exception flags. The first four flags – invalid operation, divide by zero, overflow and underflow – have their standard meanings. Let us discuss the fifth flag (inexact) that we have not encountered before. This is set when the result cannot exactly be stored in a floating point register and some rounding was required. Next, let us discuss the different rounding modes. They are stored in bits 6-8 of the $fcsr$.

Rounding Modes

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to nearest, prefer even LSBs
001	RTZ	Round towards zero.
010	RDN	Round down (towards -∞).
011	RUP	Round up (towards +∞).
100	RMM	Round to nearest, prefer the number with the maximum magnitude
101		Invalid. Reserved for future use.
110		Invalid. Reserved for future use.
111	DYN	Selects a rounding mode dynamically (stored in the frm field of the $fcsr$)

Table 6.13: Encoding the rounding mode

RISC-V instructions can use a static rounding mode (encoded in the instruction) or a dynamic rounding mode (encoded in the $fcsr$'s frm field). The default rounding mode is RNE. We round the result to the nearest value that can be represented in the IEEE 754 format. If the

real value is between two representable values, then the result is rounded to the value that has an *even* LSB. The next rounding mode is RTZ, which is round towards zero. It is equivalent to truncation where the bits that cannot be fit in the format are simply removed. The next two rounding modes are self-evident: RDN (round towards $-\infty$ or the floor function) and RUP (round towards $+\infty$ or the ceiling function).

The RMM rounding mode is similar to RNE. However, if the result is between two representable values, then we round towards the number that has the higher magnitude (away from zero). The next two values are not used at the moment. Finally, the DYN mode selects a rounding mode dynamically (stored in the frm field of the $fcsr$).

6.4.2 Load and Store Instructions

Semantics	Example	Explanation
flw rd, imm(rs1)	flw f1, 48(sp)	f1 ← mem[48 + sp]
fsw rs2, imm(rs1)	fsw f1, 48(sp)	mem[48 + sp] ← f1

Table 6.14: Single precision load and store instructions

Let us now look at the basic floating point load and store instructions in Table 6.14. They load and store values from memory, respectively. They do not perform type conversion. The $fcvt$ instruction and its variants can be used to perform type conversion, as we shall see later.

The key idea in RISC-V is the same as in x86, which is that floating point immediates cannot be directly loaded into registers. Their contents need to be stored in memory first and then the 32-bit floating point value can be loaded into a floating point register. In this sense, this part of the ISA is less powerful than its integer counterpart. However, this does not cause much of a performance loss in practice because most of the time we do not face the need for loading floating point immediates, other than while loading built-in constants such as π and e. In this case, we can use the assembler pseudoinstruction *la* to store the contents of the constant to memory and then load the address of the starting memory address to a register (refer to Example 115). The floating point load and store instructions otherwise are quite similar to their integer counterparts in RISC-V.

Example 115
Load the value of a constant val into a floating point register $fs1$.

Answer:

```
val: .float 3.14

.main:
la a1, val
flw fs1, 0(a1)
```

Semantics	Example	Explanation
fadd.s rd, rs1, rs2	fadd.s f1, f2, f3	f1 ← f2 + f3
fsub.s rd, rs1, rs2	fsub.s f1, f2, f3	f1 ← f2 - f3
fmul.s rd, rs1, rs2	fmul.s f1, f2, f3	f1 ← f2 × f3
fdiv.s rd, rs1, rs2	fdiv.s f1, f2, f3	f1 ← f2 ÷ f3
fmin.s rd, rs1, rs2	fmin.s f1, f2, f3	f1 ← min(f2 , f3)
fmax.s rd, rs1, rs2	fmax.s f1, f2, f3	f1 ← max(f2 , f3)
fsqrt.s rd, rs1	fsqrt.s f1, f2	f1 ← $\sqrt{f2}$

Table 6.15: Floating point arithmetic instructions

6.4.3 Floating Point Arithmetic Instructions

Table 6.15 shows the floating point arithmetic instructions. They are of the form $\langle inst \rangle.s$. The ".s" suffix corresponds to single precision floating point instructions. The ".d" suffix corresponds to double precision floating point instructions. The instructions $fadd.s$, $fsub.s$, $fmul.s$, $fdiv.s$, $fmin.s$, $fmax.s$ and $fsqrt.s$ have their usual meanings. Note that we do not have variants that accept immediates directly as source operands. In the case of floating point instructions, immediates can only be loaded using flw instructions or converted from integers. Refer to Example 116.

Example 116
Compute $\sqrt{\pi + e + \pi \times e}$*, and store the result in* $fa0$.

Answer:

```
# declare the constants
pi: .float 3.14
e:  .float 2.72

.main:
# load them into floating point registers
   la a1, pi
   flw fs1, 0(a1)

   la a2, e
   flw fs2, 0(a2)

   fadd.s ft1, fs1, fs2    # pi + e
   fmul.s ft2, fs1, fs2    # pi * e
   fadd.s ft3, ft1, ft2    # pi + e + pi * e

   fsqrt.s fa0, ft3        # sqrt (pi + e + pi*e)
```

To support operations such as dot products, matrix multiplication, and similar operations,

Semantics	Example	Explanation
fmadd.s rd, rs1, rs2, rs3	fmadd.s f1, f2, f3, f4	f1 ← f2 * f3 + f4
fmsub.s rd, rs1, rs2, rs3	fmsub.s f1, f2, f3, f4	f1 ← f2 * f3 - f4

Table 6.16: Fused addition and subtraction instructions

RISC-V supports a few fused arithmetic instructions such as the fused addition and subtraction operations (refer to Table 6.16). The fused add instruction ($fmadd.s$) takes three register source operands as arguments. It multiplies the first two and adds the product to the third. On similar lines, the fused subtract instruction subtracts the third source operand from the product of the first two register-based source operands.

6.4.4 Floating Point Conversion Instructions

Semantics	Example	Explanation
fcvt.s.w rd, rs1	fcvt.s.w f1, x5	f1 ← (float) x5
fcvt.w.s rd, rs1	fcvt.w.s x5, f1	x5 ← (int) f1

Table 6.17: Floating point ↔ integer conversion instructions

Example 117
Compute $\pi \times e + 4$, and store the result in $fa0$. Convert the result to an integer and store the result in $a0$.

Answer:

```
pi: .float 3.14
e:  .float 2.72

.main:
    la a1, pi              # load pi
    flw fs1, 0(a1)

    la a2, e               # load e
    flw fs2, 0(a2)

    addi   t1, zero, 4     # load 4.0 in a register
    fcvt.s.w ft1, t1

    fmadd.s ft0, fs1, fs2, ft1    # pi * e + 4
    fcvt.w.s a0, ft0              # convert to int
```

Table 6.17 shows the floating point to integer conversion (and vice versa) instructions. The $fcvt.s.w$ instruction proves to be very helpful. It can be used to convert integer immediates to

Semantics	Example	Explanation
flt.s rd, rs1, rs2	flt.s s1, f2, f3	if (f2 $<$ f3) set s1 to 1
fle.s rd, rs1, rs2	fle.s s1, f2, f3	if (f2 $\leq$ f3) set s1 to 1
feq.s rd, rs1, rs2	feq.s s1, f2, f3	if (f2 == f3) set s1 to 1

Table 6.18: Floating point comparison instructions

floating point numbers, whenever we wish to multiply a floating point number with a multiplier of the form 2.0 or 3.0.

6.4.5 Floating Point Comparison Instructions

Comparing floating point numbers is not the same as comparing integers. They cannot be directly given as arguments to conditional branch instructions. In this case, the status of the comparison needs to be stored in an integer register. This register can then be compared with the *zero* register using a regular conditional branch instruction.

Table 6.18 shows the three floating point comparison instructions that store the result in an integer register. Let us explore their usage using an example (Example 118).

Example 118
First, initialize $a0 = 0$*, then set* $a0 = 17$ *if* $e < \pi$.

Answer:

```
pi: .float 3.14
e:  .float 2.72

.main:
    la a1, pi                   # load pi
    flw fs1, 0(a1)

    la a2, e                    # load e
    flw fs2, 0(a2)

    add a0, zero, zero          # a0 = 0

    flt.s t0, fs2, fs1          # compare pi and e

    beq t0, zero, .end          # if (t0 == 0) jump to .end
    addi a0, zero, 17           # a0 = 17 because t0 == 1

.end:
```

6.5 Instruction Encoding

6.5.1 Arithmetic and Data Transfer Instructions

RISC-V has four core non-branch instruction formats: R, I, S and U. These are for 32-bit instructions. RISC-V assemblers and compilers further align these instructions to 4-byte (32-bit) boundaries. Figure 6.2 shows a visual representation of these four formats and Table 6.19 shows examples of instructions in each format.

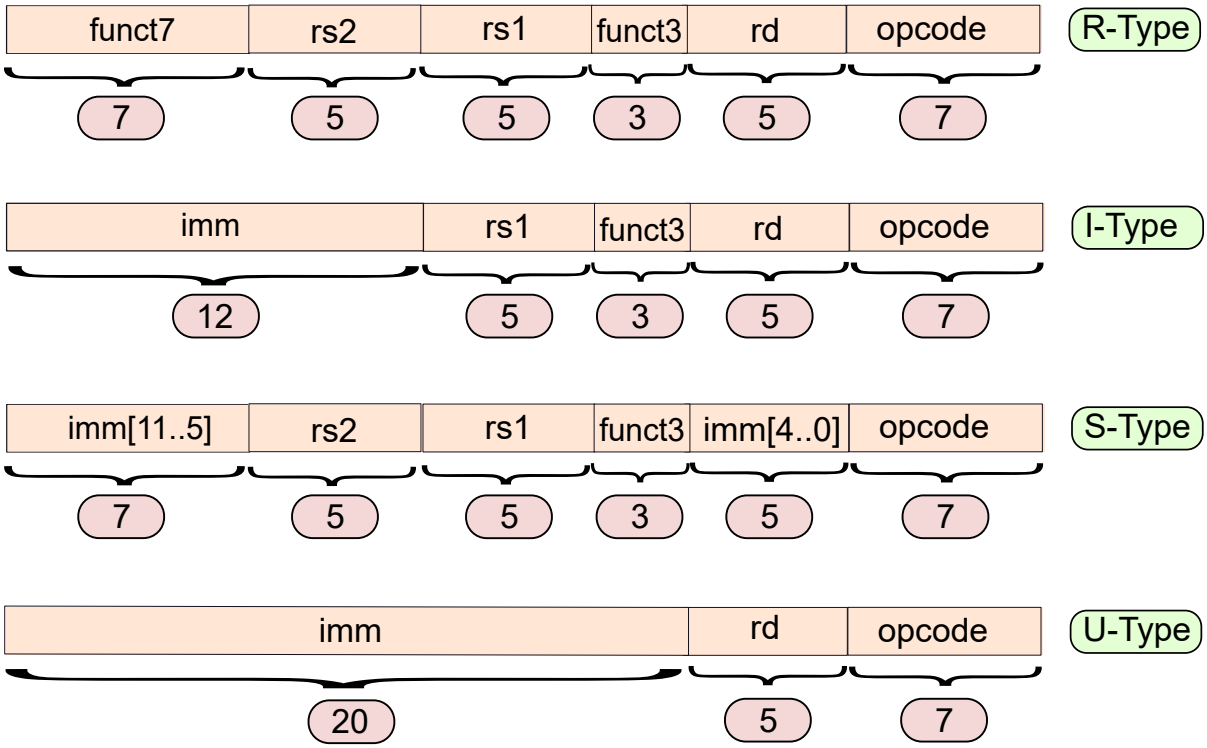

Figure 6.2: RISC-V instruction formats: R, I, S and U formats

Format	Structure	Instructions
R	rd, rs1, rs2	add, sub, mul, div, rem, and, or, xor, sll, srl, sra, slt, sltu
I	rd, rs1, imm rd, imm(rs1)	addi, andi, ori, xori, slli, srli, srai, slti, sltiu lw, jalr
S	rs2, imm(rs1)	sw
U	rd, imm	lui

Table 6.19: Instructions belonging to each of the four arithmetic RISC-V formats

Let us now take a deep look at each of these four instruction formats. The first 7 bits are reserved for the opcodes in all the formats. This means that we can support a maximum of

128 instructions in the ISA; moreover, finding the opcode is also quite easy (first 7 bits). The reason that this is important is because minimizing the decoder's complexity is a key goal of ISA design. Given that all the opcode bits are in the same positions across the formats, finding the opcode and consequently the type of the instruction is easy.

The next 5 bits are used to store the id of the destination register *rd* in the R, I and U formats. S-type instructions do not have a destination register. For example, the *sw* instruction, which is an S-type instruction, does not have a destination register. It however has two source registers and an immediate. Instead of changing the positions of the source registers in the S-type instruction format, the ISA designers made the right decision to use the bits for the destination register to store a part of the immediate. Hence, the first 5 bits of the immediate are stored at the corresponding positions.

Next, let us consider U-type instructions (such as *lui*). We can use the remaining 20 bits to store the 20-bit immediate. We don't need to store any more information. Such instructions have only two arguments. The destination register *rd* and the 20-bit immediate.

The rest of the three formats (R, I and S) store a 3-bit field $funct3$. It is used to hierarchically organize the instructions and to also support more instructions. In RISC-V, a single opcode can correspond to multiple instructions. For example, the opcodes of the *add*, *slt* and *xor* instructions are the same: 0110011. They are differentiated by the values of the $funct3$ field. We can thus support more instructions than 128. However, that is not the main aim here. We can group similar instructions such that they are processed by the hardware in the same manner. They will still have differences between them such as *add* and *xor*. However, most of the processing can remain the same given that both are R-type instructions.

The next 5 bits store the $rs1$ field (first source register) in all three formats (R, I and S). Subsequently, differences arise. I-type instructions need to store a 12-bit immediate. They don't have a second source register ($rs2$). They thus use the remaining 12 bits to store the immediate. All immediates are sign-extended before being used in hardware unless the instruction has a 'u' suffix (for unsigned). The R and S-type instructions store the second source register $rs2$ in the next 5 bits.

Let us now consider the last 7 bits in the R and S formats. R-type instructions have another opcode extender called $funct7$ (in addition to $funct3$). It serves the same purpose as $funct3$. The aim is to increase the number of instructions and also create a grouping of instructions based on their similarity. It is possible for two instructions to have the same opcode and $funct3$ fields, yet have a different $funct7$ field. Consider *add* and *sub*. They have the same opcode (0110011) and same $funct3$ (000); however, their $funct7$ fields are different – 0000000 and 0100000, respectively.

In contrast, S-type instructions store the remaining 7 immediate bits in the uppermost (most significant) 7 positions. Recall that we had already stored 5 immediate bits in the positions at which other formats store the id of the destination register (rd). This is a very reasonable decision because the explicit aim is to ensure that the same field is stored at the same set of positions across the formats – it is very easy for the decoder to extract it.

6.5.2 Control Flow Instructions

Let us now look at the instruction formats for encoding control flow instructions. There are two formats in this space: the B and J formats. Refer to Figure 6.3 and Table 6.20.

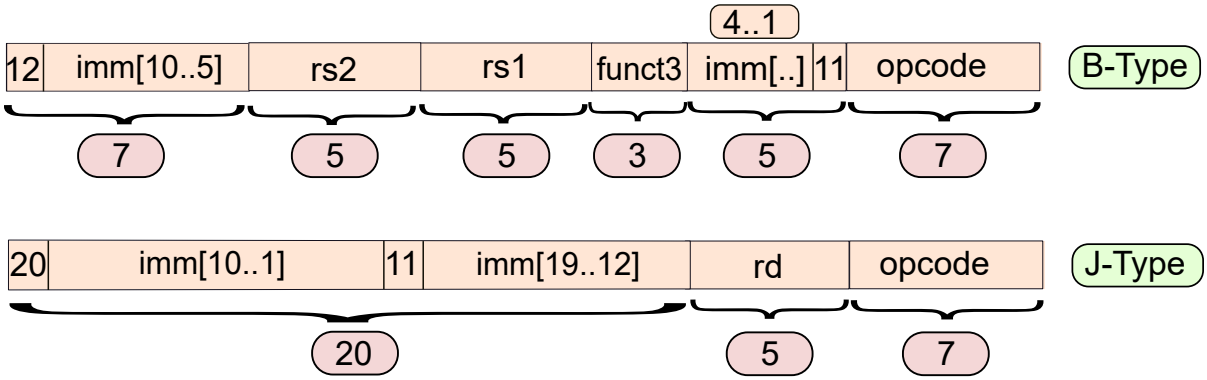

Figure 6.3: The B and J formats

Format	Structure	Instructions
B	rs1, rs2, imm	beq, bne, blt, bge, bltu, bgeu
J	rd, rs1, imm	jal

Table 6.20: Instructions that are encoded in the B and J formats

All the conditional instructions like *beq*, *bne* and *blt* are implemented using the B format. Recall that such instructions take two source registers. They are compared and then based on the results of the comparison, a taken/not-taken decision is made. The instruction uses PC offset-based addressing, where the offset is encoded in the immediate field.

The immediate is encoded in a special manner in the B format. The offset needs to be a multiple of 2 (limitation of the ISA). This means that its LSB is 0. Given that this bit is fixed, there is no need to represent it. In other words the 0^{th} bit is set to 0 and thus need not be represented. The format thus stores the rest of the 4 bits in the first 5-bit field. Note that as compared to the S format, the immediate bits 4..1 are stored in exactly the same positions. This makes extracting these bits very easy and there is consequently no need to design additional decoder hardware to handle these bits differently in the B format. Given that the 0^{th} bit is not stored, its corresponding position can be used to store the 11^{th} bit.

The rest of the immediate bits are stored in the most significant bit positions. The most significant 7 bits store the 12^{th} bit and the bits 10..5. We thus store 13 immediate bits: 12 of them are explicitly stored and the least significant immediate bit is assumed to be 0. This format can thus encode an offset between -4096 and 4095 ($\approx \pm$ 4 KB). This offset is sign-extended and added to the PC.

Next, consider the J format. It takes a single destination register and a 20-bit immediate as its arguments. The immediate here encodes an offset that is a multiple of 2 (akin to the B format). There is no need to store the LSB, which is set to 0. Like the B format, there is a need to store 20 bits. The order of storing the bits from the most significant position to the least significant position is as follows: bit 20, bits 10..1, bit 11, bits 19..12. Given that we encode 21 bits in this format (20 explicitly and 1 implicitly), we can represent an offset range that is within $\pm 1\,MB$ of the current PC.

6.5.3 Floating Point Instructions

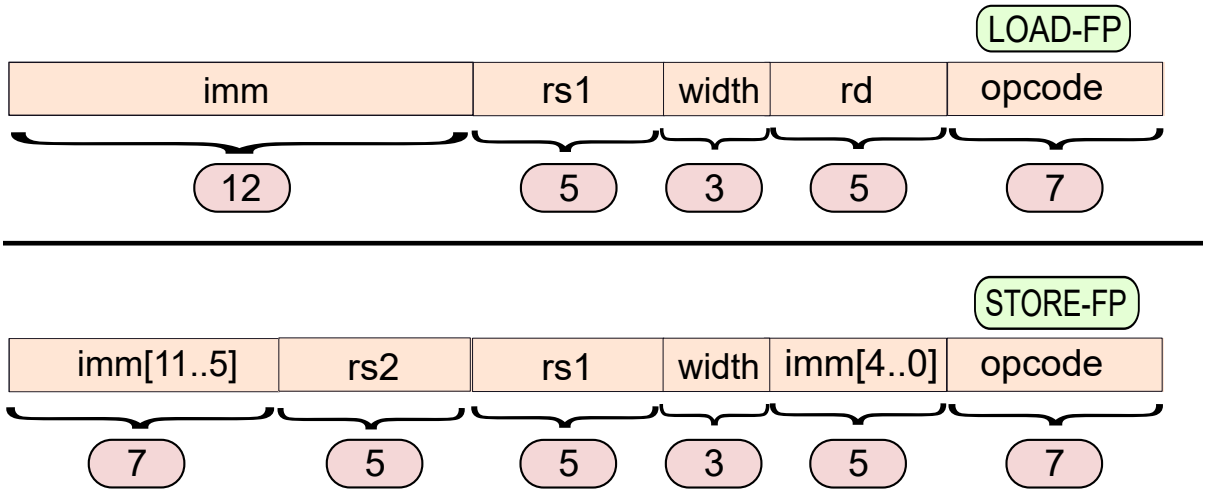

Figure 6.4: Encoding the flw and fsw instructions

Figure 6.4 shows the encoding of the flw and fsw instructions. flw instructions are encoded in the I format. The $funct3$ instruction is replaced with the *width* field (amount of data that is loaded). Similarly, the fsw instruction is encoded using the S format. The only change is that the $funct3$ field is replaced with the *width*.

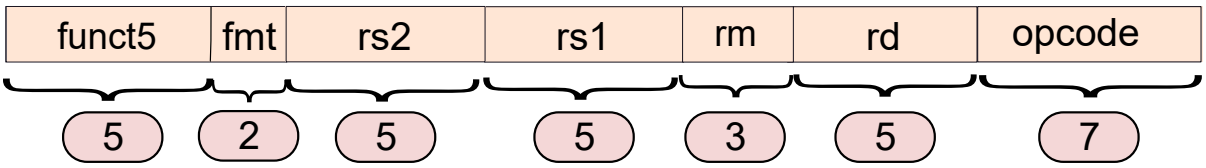

Figure 6.5: Encoding of floating point arithmetic instructions

Figure 6.5 shows the encoding format of floating point arithmetic instructions (variation of the R format). All such instructions take one floating point destination register and one or two source registers as inputs. The format is the same for all variants. The rm field encodes the rounding mode and the fmt field represents the precision (32-bit, 64-bit, 16-bit, 128-bit).

The opcode field is typically the same for all common floating point arithmetic instructions. The $funct5$ field stores the code for the specific type of instruction. For instructions like $fqrt$ that do not have the second source register, the $rs2$ field is set to 0.

The same format is also used by the floating point conversion instructions ($fcvt.w.s$ and $fcvt.s.w$).

This format is also used by floating point comparison instructions. The rm field in this case stores the following comparison conditions: EQ, LT and LE. The $funct5$ field stores a code for floating point comparison ($FCMP$).

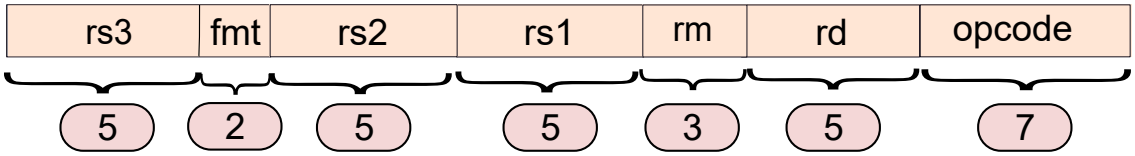

Figure 6.6: Encoding of fused multiply and add instructions

Figure 6.6 shows the encoding format of the $fmadd$ and $fmsub$ instructions. Instead of the $funct5$ field, the third source register $rs3$ is stored in its place. The rest remains the same.

6.6 Summary and Further Reading

6.6.1 Summary

Summary 6

1. *The RISC-V ISA refers to a family of instruction sets. The basic ISA is RV32 (32-bit). There are 64-bit and 128-bit variants as well that are currently under different stages of development. They are named RV64 and RV128, respectively.*
2. *The ISA has a modular structure. Different sets of instructions can be added to it depending upon the use case. Each such module is known as an "extension".*
3. *Some popular extensions are as follows: integer (default), embedded, atomic instructions, single and double-precision floating point arithmetic, and vector arithmetic.*
4. *There is a compressed instruction set (suffix 'C') that is similar in principle to ARM Thumb.*
5. *There are 32 integer registers. The zeroth integer register ($x0$ or $zero$) is hardwired to 0. There is an elaborate usage convention that most assembly programmers are expected to follow.*
6. *The usage convention distinguishes between temporary registers (caller saved), callee saved registers and function arguments/return values.*
7. *The integer registers are named $x0 \ldots x31$. They additionally can be addressed using their mnemonics $t0-6$ (temporary), $s0-12$ (callee saved), $a0-7$ (arguments and return values), ra (return address), sp (stack pointer), gp (global pointer) and $zero$. Addressing registers by their mnemonics is preferred.*
8. *For example, the integer register $t3$ is the third temporary register that is the same as $x28$.*

9. *The RISC-V ISA is a RISC ISA that accepts 12-bit immediates in arithmetic instructions and 20-bit immediates in branch instructions and the load-upper-immediate instruction.*

10. *There is no dedicated move-immediate instruction in the ISA. Instead, the way to load an immediate is to use the addi instruction to add the 12-bit immediate to the register zero. The upper 20 bits can then be set by the lui (load upper immediate) instruction.*

11. *Akin to other RISC ISAs, RISC-V supports all the standard arithmetic and logical instructions including some unsigned variants.*

12. *There is no dedicated flags register that stores the result of the last comparison. Instead, branch instructions take two register arguments. They directly compare them and depending upon the branch condition, jump to the label specified in the instruction.*

13. *The jal and jalr instructions are used to jump to a different location and store the return address in the first source register. If the register is zero, then the return address is not saved. The jal instruction can be used to implement the classical call instruction while the jalr instruction can be used to implement the return instruction.*

14. *There are two important assembler directives that translate to multiple RISC-V instructions at runtime. They are li (load 32-bit immediate) and la (load the address of a constant defined in the assembly file into a register).*

15. *RISC-V has 32 floating point registers numbered* $f0 \ldots f31$. *No floating point register is hardwired to 0. They also have a usage convention and are also known by their mnemonics. These mnemonics have a similar pattern:* $ft0-11$, $fa0-7$ *and* $fs0-12$.

16. *The floating point control status register (*$fcsr$*) is used to control the behavior of floating point instructions. It stores the rounding mode and floating point exceptions seen after the last time this register was reset (divide-by-zero, overflow, etc.).*

17. *There is no direct way of loading a floating point immediate into a register in RISC-V. In RISC-V, an immediate is associated with a label, and it is assumed to be stored in memory before the execution of the code starts. The address of the label (or the immediate) can be loaded to a register using the assembler directive la. Subsequently, the* flw *instruction can be used to load the corresponding floating point value. The* fsw *instruction can be used to store floating point values.*

18. *All single-precision floating point arithmetic instructions operate in a manner that is more or less similar to their integer counterparts. They have the ".s" suffix. For example, the floating point add instruction is named* $fadd.s$.

19. *Another way of loading or storing immediate values is using the floating point conversion instructions:* $fcvt.w.s$ *and* $fcvt.s.w$.

20. *Floating point comparison instructions have an integer destination register and two floating point source registers. The hardware compares the source registers based on the type of the comparison that needs to be performed, and then the Boolean result is stored in the destination register.*

21. *RISC-V has six different instruction formats: 4 integer formats (R, I, S and U) and 2 branch formats (B and J).*

22. *Most arithmetic instructions that do not have immediates use the R format. The I format is used for instructions that use an immediate such as the addi instruction or the lw (load) instruction. Store instructions are encoded using the S format and the lui instruction uses the U format.*

23. *All the conditional branch instructions use the B format. The B format admits a 12-bit immediate with an additional and implicit LSB bit that is hardwired to 0. jal is a J-type instruction that has a single destination register and a 20-bit immediate (the LSB is not specified because it is 0). Effectively, the B format has a 13-bit immediate and the J format has a 21-bit immediate.*

24. *Floating point instructions use the I format for flw and S format for fsw instructions, respectively. The rest of the instructions primarily rely on minor variations of the R format.*

6.6.2 Further Reading

The most definitive resource for understanding the RISC-V ISA is its official manual[ris,] that can be downloaded from `https://riscv.org/`. The site hosts two kinds of specifications: unprivileged specification and privileged specification. The privileged specification is for writing system software and operating systems. All the RISC-V specifications undergo active development and periodically new versions are released. A GitHub repository tracks the development of these specifications. It is accessible at `https://github.com/riscv/`. Readers can additionally refer to two classic books[Waterman, 2016, Patterson and Waterman, 2017] to learn more about the RISC-V ISA.

Readers should also read a few classical papers [Chen and Patterson, 2016, Greengard, 2020, Asanović and Patterson, 2014, Mezger et al., 2022] to understand the history of RISC-V. This will give them a perspective of the developmental history of RISC-V and how this ISA came about in an era when instruction set development was considered to be an already solved problem that did not warrant further attention.

The next port of call can be papers that critically investigate the RISC-V ISA. The following references [Frolov et al., 2021, Kanter, 2016, Singh and Sarangi, 2021] will prove to be useful. They critique some design choices of the RISC-V ISA and compare it with other RISC and CISC ISAs (particularly reference [Singh and Sarangi, 2021]). In this context, readers should consider the formal specifications of RISC-V [Bourgeat et al., 2021] if they are considering implementing the ISA or designing a machine-accurate emulator for it.

Next, let us consider performance and implementation-related aspects. Researchers can

look at architecture simulators that simulate RISC-V instructions and their vector extensions such as the simulator released by Ramirez et al. [Ramírez et al., 2020]. The next logical step is to study RISC-V processors such as BOOMv2 [Celio et al., 2017], RISC-V 2 [Patsidis et al., 2020] and the processor in reference [Stangherlin and Sachdev, 2022]. RISC-V processors are also being designed to operate in high-radiation environments like outer space. Many space research organizations are creating their bespoke RISC-V processors [Wessman et al., 2021].

Exercises

RISC-V Assembly Programming

Ex. 1 — Solve all the exercises listed at the end of the chapter on the ARM assembly language using RISC-V.

RISC-V Assembly Concepts

Ex. 2 — Why does RISC-V not have a *mov* instruction? What is the advantage of making this choice?

Ex. 3 — How does the assembler implement the *li* directive (pseudoinstruction)?

* **Ex. 4** — RISC-V does not have a $flags$ register. However, it stores some information in the $fcsr$ register. Why is this required?

Ex. 5 — Explain the different rounding modes in RISC-V.

Ex. 6 — Why is it not a good idea to have instructions to load floating point immediates directly into registers (similar to *addi* and *lui* for integers)?

* **Ex. 7** — How does the assembler implement the *la* directive?

Ex. 8 — What is the advantage of maintaining the positions of the fields across the different RISC-V instruction encoding formats?

* **Ex. 9** — How do the opcode, $funct3$, $funct5$ and $funct7$ fields help in implementing RISC-V extensions?

Ex. 10 — What is the advantage of making it easy to extract the sign bit of the immediate in the different formats, especially the B and J formats?

Design Problems

Ex. 11 — Extend the RISC-V assembler available on the author's website to support the following extensions: double precision, vector, SIMD and cryptographic operations.

Ex. 12 — Cross-compile a piece of C code using the RISC-V and ARM cross-compilers. Use the -03 *gcc* optimization. Next, run them on the Qemu emulation engine. Compare the performance and find the reasons for the differences.