

Computer Systems Architecture

Laboratory - Part 0x02

Ruxandra Bălucea
Bogdan Macovei
Cristian Rusu

Noiembrie 2024

Cuprins

1	Introduction	3
1.1	Context	3
1.2	Instruction Formats and Encodings	4
1.3	RISC-V extensions	5
1.4	x86 vs RISC-V comparison	6
2	Working Environment	7
3	RISC-V Assembly Language	9
3.1	Registers	9
3.2	Data Types	10
3.3	Main Instructions	10
3.4	Load-Store Architecture	11
3.5	Relocations	12
3.6	System Calls	13
3.7	Jumps	14
3.8	One-Dimensional Arrays	16
3.9	RV32C - Compressed Instructions	17
4	Calling Convention	19
4.1	Calling a Procedure	19
4.2	Procedure Arguments and Return Value	20
4.3	Creating the Call Frame	21
5	Pipelining	23
5.1	Pipeline Hazards	24
5.2	Performance Measurement	26
6	Cache	27
6.1	Cache Memory Performance	30

7 Exercises	31
7.1 Laboratory 9	31
7.2 Laboratory 10	31
7.3 Laboratory 11	31
8 Additional Resources	32

1 Introduction

1.1 Context

RISC-V is a RISC instruction set architecture (**ISA** - *Instruction Set Architecture*) developed in 2010 at the University of California, Berkeley, with the following main characteristics:

1. It is an **open-source architecture**. Unlike x86 and ARM, anyone can use and implement RISC-V without licensing fees or restrictions, encouraging innovation and reducing dependence on commercial vendors.
2. **Modularity**. RISC-V defines a minimal base instruction set that can be extended with optional features (for floating-point operations, vectors, cryptography, cache management, etc.).
3. **Portability**. The architecture is not tied to a specific hardware implementation, making it easy to use across different types of devices. RISC-V can be used in both simple IoT processors and complex processors for servers and supercomputers.

The global developer community actively contributes to the standardization and implementation of RISC-V, while organizations such as **RISC-V International**¹ coordinate these efforts. Founding members include Andes, Antmicro, ETH Zurich, Google, IBM, lowRISC, Microchip, Qualcomm, Rambus, Rumble, SiFive, etc.

Rapid innovation, technological independence, and lower costs have made RISC-V the choice of many companies for embedded and mobile devices, data centers and supercomputers, artificial intelligence, automotive systems, and even military and government applications.

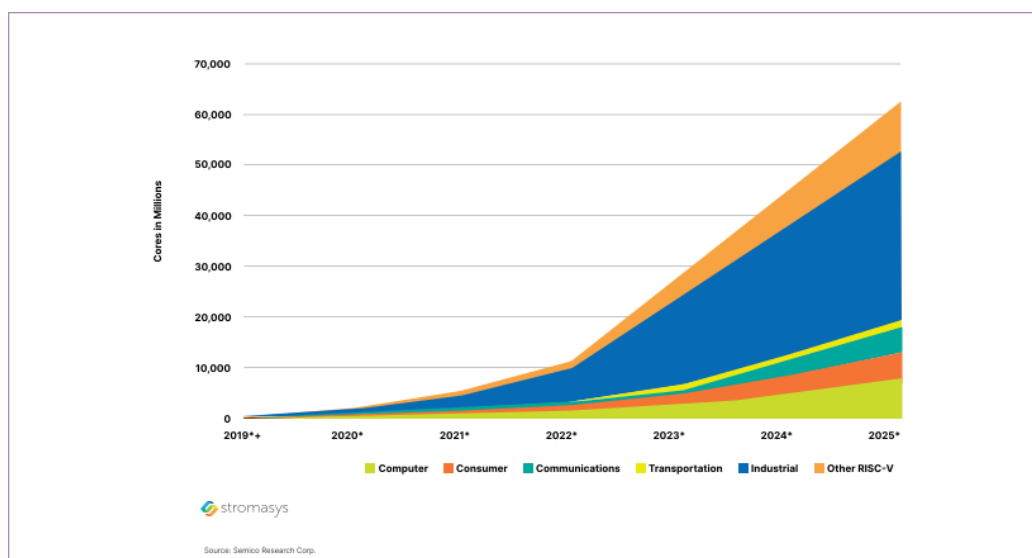


Figura 1: RISC-V evolution ²

¹<https://riscv.org/>

²<https://www.stromasys.com/resources/all-about-the-risc-v-processors/>

1.2 Instruction Formats and Encodings

RISC-V and x86 take fundamentally different approaches to instruction construction and encoding. RISC-V is simple and modular, with a small set of fixed instruction formats, whereas x86 is an older and more complex architecture with variable-length encodings.

An x86 instruction can occupy between 1 and 15 bytes and may contain several components that make decoding slower and more complex, including:

- **prefixes** (for example, to distinguish register sizes - `mov %eax, %ebx` has the encoding `89 c3`, while `mov %ax, %bx` has the encoding `66 89 c3`);
- **opcodes** (the mnemonic) - specify the main instruction;
- **addressing modes** - register/memory;
- additional **operands**.

In contrast, RISC-V uses fixed-length 32-bit (4-byte) instructions organized into several standard formats, such as R, I, S, B, U, and J. This regular structure simplifies instruction decoding in hardware.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]				rs2		rs1		funct3		imm[4:1:11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20:10:1 11:19:12]										rd		opcode		J-type

Figura 2: RISC-V Instruction Formats ³

Example: Consider the instruction `add a0, a1, a2`, which adds the values stored in registers `a1` and `a2` and stores the result in `a0`. This is an R-type instruction. According to the architecture specification manual⁴, the instruction uses `funct7` = 0000000, `funct3` = 000, and `opcode` = 0110011. The register numbers, in decimal, are `a0` = 10, `a1` = 11, and `a2` = 12. Using these values, we can construct the instruction encoding as follows:

func7	rs2	rs1	funct3	rd	opcode
0 0 0 0 0 0 0	0 1 1 0 0	0 1 0 1 1	0 0 0	0 1 0 1 0	0 1 1 0 0 1 1

Therefore, by grouping the bits in sets of four and converting them to hexadecimal, we obtain the encoding `0x00c58533`.

Exercise: What is the encoding of the `addi a0, a1, 15` instruction, which is of type I and adds the immediate value 15 to the value in source register `a1`, storing the result in destination register `a0` (see the register codes above)? We also know that `funct7` = 0000000, `funct3` = 000, and `opcode` = 0010011.

³<https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/instr-table.html>

⁴<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

1.3 RISC-V extensions

As presented earlier, one of the main advantages of the architecture is its modularity and the organization of instructions into specific functional groups that can be included or omitted depending on the processor.

At the core of RISC-V is the base instruction set, known as **RV32I** for 32-bit architectures and **RV64I** for 64-bit architectures. These include the minimum set of instructions required to implement a functional processor architecture. The main features of the base set include:

- Basic arithmetic operations (**add**, **sub**, etc.);
- Memory access operations (**lw**, **sw**, etc.);
- Control-flow instructions (**jal**, **beq**, etc.)

Some examples of standard RISC-V extensions also include:

- **RV32M** - support for more advanced arithmetic operations, such as multiplication (**mul**) and division (**div**). It is essential for applications that require intensive numerical computation.
- **RV32C** - a subset of compressed instructions that reduces code size and saves memory. It is useful in embedded systems and other resource-constrained applications.
- **RV32A** - support for atomic operations, which are crucial in multithreaded programs. Instructions such as **lr** (load-reserved) and **sc** (store-conditional) allow efficient synchronization between multiple threads.
- **RV32F** and **RV32D** - support for floating-point operations. These are essential in applications such as digital signal processing and machine learning.
- **RV32G** - a general-purpose extension equivalent to RV32IMAFD.
- **RV32V** - support for vector computation, essential in applications such as image processing, scientific simulations, and deep learning.
- **RV32B** - instructions optimized for bit manipulation, used in cryptography and signal processing.

Like the base set, all of these extensions also support 64-bit variants.

In addition, organizations and companies can define their own extensions for specific needs, contributing to the diversity of RISC-V implementations.

All ratified extensions, as well as the current status of extensions undergoing ratification, can be found here⁵.

⁵<https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154732/Ratified+Extensions>

1.4 x86 vs RISC-V comparison

We conclude this introductory section with a summary of the key strengths and limitations of the two architectures. More broadly, the comparison highlights the fundamental trade-offs between the CISC and RISC design philosophies.

	x86 (CISC)	RISC-V (RISC)
Advantages	• Compact encoding of simple instructions, resulting in better code density and lower memory requirements. • Support for complex addressing modes and instructions, while maintaining compatibility with legacy software.	• Simplicity and uniformity → easy to implement in hardware and fast to decode. • Scalability → extensions can be added without affecting existing encodings. • Efficient pipelining → fixed formats allow instructions to be decoded in parallel.
Disadvantages	• Slow and complex decoding → difficulties in designing modern processors. • Historical legacy → difficulties in removing redundant or inefficient instructions.	• Sometimes requires more instructions for complex operations compared with x86.

Tabela 1: Comparison of the advantages and disadvantages of x86 and RISC-V

2 Working Environment

The RISC-V ecosystem includes a wide range of tools and simulators for education, research, and industry. In this section, we briefly introduce some of the most widely used ones and discuss their main purposes.

Spike⁶ is the official RISC-V **ISA simulator** and is primarily used as a reference implementation for testing and validation. It supports the standard RISC-V ISA and extensions, with an emphasis on correctness rather than performance. As a result, execution can be relatively slow. **QEMU**⁷, on the other hand, is an **emulator** that supports a wide range of architectures, including RISC-V, ARM, and x86, as well as different system configurations. This makes it particularly useful for operating system and embedded software development. QEMU also integrates with debugging tools such as **GDB** and provides much better execution performance by dynamically translating instructions from the guest ISA to code that can be executed by the host machine.

Hardware design is another major part of the RISC-V ecosystem. One of the most widely used hardware construction languages is **Chisel**⁸, which can be used to describe and implement processor designs. **Chipyard**⁹ provides a framework that brings together several RISC-V processor implementations, including Rocket Chip and BOOM, as well as different accelerators. It also supports FPGA-based deployment, allowing these designs to be tested on **FPGAs**, i.e., physical hardware that can be reconfigured to implement a given digital design.

The RISC-V ecosystem also provides the standard software development tools expected for a modern architecture:

- **RISC-V GNU Toolchain**¹⁰ – provides GCC, binutils, and GDB support for RISC-V.
- **LLVM/Clang**¹¹ – provides compiler and optimization support for RISC-V, including support for a wide range of ISA extensions.

Setting up these tools often requires building them from source and configuring them for the local environment, which can be a time-consuming process. To avoid spending laboratory time on toolchain setup, we will instead use **Ripes** (<https://ripes.me/>), a browser-based RISC-V simulator designed specifically for educational use. Ripes is open source and can also be installed locally from its source repository¹². More detailed instructions and examples are available in the Ripes documentation.

One of the main advantages of Ripes is that it provides a visual representation of how instructions are executed by the processor. Different processor models can be selected, including single-cycle and pipelined implementations, making it possible to follow an instruction as it moves through the different stages of execution. The interface also provides direct access to registers and memory and allows RISC-V assembly programs to be written and executed interactively.

⁶<https://github.com/riscv-software-src/riscv-isa-sim>

⁷<https://www.qemu.org/>

⁸<https://www.chisel-lang.org/docs>

⁹<https://chipyard.readthedocs.io/en/stable/Chipyard-Basics/Chipyard-Components.html>

¹⁰<https://github.com/riscv-collab/riscv-gnu-toolchain>

¹¹<https://github.com/llvm/llvm-project>

¹²<https://github.com/mortbopet/Ripes>

For the exercises in this laboratory, we will use the RV32I base instruction set, with the possibility of enabling additional extensions such as M and C.

We will explore Ripes in more detail later. For now, the following features are sufficient to investigate the concepts introduced in the previous sections:

- **Program sections** (`.text`, `.data`, `.bss`) can be inspected using the `Go to section:` option.
- **The stack** can be located using `Go to register:` and selecting the `sp` register, which holds the stack pointer, similarly to `esp` in x86. Notice that the stack is placed near the upper end of the address space, at approximately `0x7fffffff0`. This follows the process memory layout discussed previously in the laboratory on x86 procedures.
- **Registers** can be inspected directly in the register view on the right-hand side of the interface.
- **Assembly code** can be written directly in the editor. For each instruction, Ripes immediately displays its binary encoding as well as its disassembled representation, allowing us to see the relationship between an assembly instruction and its machine-code encoding. Programs written locally can also be loaded into the simulator.

Note:

1. To follow instruction execution step by step, select a **Single-cycle processor** using the processor selector located below the *File* menu.
2. If the online version of Ripes fails to load, clearing the browser cache or opening the simulator in a private/incognito window usually resolves the issue.

3 RISC-V Assembly Language

3.1 Registers

RISC-V provides 32 general-purpose registers (x0 through x31), each 32 bits wide in the 32-bit architecture. These registers are used to store and manipulate data and, much like registers in x86 assembly, can be thought of as the variables available to assembly programs.

Encoding	RVC Encoding	Register	ABI Name	Description	Callee- saved
0	-	x0	zero	Hardwired zero	-
1	-	x1	ra	Return address	No
2	-	x2	sp	Stack pointer	Yes
3	-	x3	gp	Global pointer	-
4	-	x4	tp	Thread pointer	-
5	-	x5	t0	Temporary register 0	No
6	-	x6	t1	Temporary register 1	No
7	-	x7	t2	Temporary register 2	No
8	0	x8	s0/fp	Saved register 0 / frame pointer	Yes
9	1	x9	s1	Saved register 1	Yes
10	2	x10	a0	Function argument 0 / return value 0	No
11	3	x11	a1	Function argument 1 / return value 1	No
12	4	x12	a2	Function argument 2	No
13	5	x13	a3	Function argument 3	No
14	6	x14	a4	Function argument 4	No
15	7	x15	a5	Function argument 5	No
16	-	x16	a6	Function argument 6	No
17	-	x17	a7	Function argument 7	No
18	-	x18	s2	Saved register 2	Yes
19	-	x19	s3	Saved register 3	Yes
20	-	x20	s4	Saved register 4	Yes
21	-	x21	s5	Saved register 5	Yes
22	-	x22	s6	Saved register 6	Yes
23	-	x23	s7	Saved register	Yes
24	-	x24	s8	Saved register 8	Yes
25	-	x25	s9	Saved register 9	Yes
26	-	x26	s10	Saved register 10	Yes
27	-	x27	s11	Saved register 11	Yes
28	-	x28	t3	Temporary register 3	No
29	-	x29	t4	Temporary register 4	No
30	-	x30	t5	Temporary register 5	No
31	-	x31	t6	Temporary register 6	No

Tabela 2: RISC-V general-purpose registers

Another register worth mentioning is the **pc (program counter)**, which holds the address of

the next instruction to be executed.

3.2 Data Types

The main data types used throughout this laboratory are:

1. **byte** - occupies 1 byte (8 bits) of memory;
2. **halfword** - occupies 2 bytes (16 bits) and can be used to store 16-bit values;
3. **word** - occupies 4 bytes (32 bits) and is typically used to store integers. A **long** is also 32 bits wide.
4. **doubleword** - occupies 8 bytes (64 bits);
5. **ascii**, **asciz**, and **space** have the same meaning as in x86 assembly (**space** is not implemented in Ripes).

3.3 Main Instructions

Instruction	Operands	Description
add	rd, rs1, rs2	Adds rs1 + rs2 and stores the result in rd .
sub	rd, rs1, rs2	Subtracts rs2 from rs1 and stores the result in rd .
mul	rd, rs1, rs2	Multiplies rs1 by rs2 (M extension).
div	rd, rs1, rs2	Divides rs1 by rs2 (M extension).
rem	rd, rs1, rs2	Computes the remainder of rs1 divided by rs2 (M extension).
and	rd, rs1, rs2	Performs a bitwise AND between rs1 and rs2 .
or	rd, rs1, rs2	Performs a bitwise OR between rs1 and rs2 .
xor	rd, rs1, rs2	Performs a bitwise XOR between rs1 and rs2 .
sll	rd, rs1, rs2	Shifts rs1 left by the shift amount specified by rs2 .
srl	rd, rs1, rs2	Logically shifts rs1 right by the shift amount specified by rs2 .
addi	rd, rs1, imm	Adds imm to rs1 and stores the result in rd .
slti	rd, rs1, imm	Compares rs1 < imm and sets rd to 1 or 0.
andi	rd, rs1, imm	Performs a bitwise AND between rs1 and imm .
lui	rd, imm	Places imm in the upper 20 bits of rd ($rd = imm \ll 12$).
auipc	rd, imm	Adds the upper immediate to the current pc ($rd = pc + (imm \ll 12)$).

Tabela 3: Main RISC-V instructions and their operands

RISC-V also defines several commonly used pseudoinstructions. These can be used in assembly code just like regular instructions, but they do not have their own encoding in the ISA. Instead, the assembler translates them into one or more actual RISC-V instructions.

Pseudoinstruction	Possible expansion	Description
<code>li rd, imm</code>	<code>lui rd, imm[31:12]</code> <code>addi rd, rd, imm[11:0]</code>	Loads the immediate value <code>imm</code> into register <code>rd</code> .
<code>la rd, label</code>	<code>lui rd, label[31:12]</code> <code>addi rd, rd, label[11:0]</code>	Loads the address of <code>label</code> into register <code>rd</code> .
<code>mv rd, rs</code>	<code>addi rd, rs, 0</code>	Copies the value from register <code>rs</code> into register <code>rd</code> .
<code>nop</code>	<code>addi x0, x0, 0</code>	Performs no operation.

Tabela 4: Common RISC-V pseudoinstructions

3.4 Load-Store Architecture

In RISC-V, arithmetic and logical operations are performed on values stored in registers rather than directly in memory. Accessing memory requires dedicated *load* and *store* instructions: a value must first be loaded into a register before it can be used in a computation, and a result must be explicitly stored if it needs to be written back to memory. This separation between computation and memory access is one of the defining characteristics of RISC architectures (Reduced Instruction Set Computing) and helps simplify both the instruction set and the processor design.

To load a value from memory, we use the following instruction:

```
lw reg1, offset(reg2)
```

Here, `reg2` contains a base memory address and `offset` specifies a displacement, in bytes, relative to that address. The effective address is therefore computed as `reg2 + offset`, and the value found at this address is loaded into `reg1`.

To store a value in memory, we use:

```
sw reg1, offset(reg2)
```

The destination address is computed in the same way, as `reg2 + offset`, while `reg1` contains the value to be written to memory.

The `lw` and `sw` instructions operate on 32-bit values. Similar instructions are available for 16-bit values (`lh` and `sh`) and 8-bit values (`lb` and `sb`).

One register that can be used when accessing global data is `gp` (*global pointer*). According to the RISC-V ABI, `gp` is reserved as a reference register for accessing global data. It holds an address within the global data area, allowing nearby data to be accessed using offsets relative to this address. In general, `gp` does not necessarily point to the beginning of the `.data` section. In the environment used for this laboratory, we will use `gp` as a reference point for the values declared in the `.data` section.

Exercise: Consider the following code sequence. What will the memory contain after the two instructions have been executed? What do you notice about the values declared in the `.data` section?

```

.data
.long 5
.long 6

.text
.global main
main:
lw a0, 0(gp)
sw a0, 4(gp)

```

3.5 Relocations

When referring to symbols defined in memory, we generally do not want to manually compute the offsets required to access them. Instead, assemblers, linkers, and loaders rely on *relocations* to determine the appropriate addresses or offsets.

A relocation identifies a reference whose final value depends on the location of a symbol or section in memory and may therefore need to be resolved at a later stage. Depending on the information available, this may happen during assembly, linking or when the program is loaded. If the addresses of the relevant sections and symbols are already known, the corresponding value can be determined directly. If a symbol is defined in another object file or library, its final address may only become known during linking. This is common in embedded systems, where the final memory layout is often specified through a linker script. For dynamically positioned code or data, some relocations may instead need to be resolved when the program is loaded or executed.

To reference symbols from data sections, RISC-V provides several relocation types:

- **Upper-part relocations (%hi and %pcrel_hi).** These are used with instructions such as `lui` and `auipc`, respectively, to construct the upper part of an address or address-relative offset.
- **Lower-part relocations (%lo and %pcrel_lo).** These provide the remaining lower part and can be used with instructions such as `lw`, `sw`, or `addi`. A lower-part relocation is used together with a corresponding upper-part relocation, allowing the complete address or offset to be constructed.
- **Jump relocations** – these will be introduced in the section on control-flow instructions.
- Other relocation types.

An important distinction is that relocations can refer either to absolute addresses or to addresses relative to the program counter:

- **%hi(sym)** and **%lo(sym)** are used to construct the absolute address of the symbol `sym`;
- **%pcrel_hi(sym)** is used to obtain the upper part of the offset between the address of `sym` and the current program counter;
- **%pcrel_lo(label)** provides the corresponding lower part. Here, `label` identifies the instruction carrying the associated `%pcrel_hi(sym)` relocation.

Thus, a variable can be accessed using one of the following approaches:

```
.data
a: .long 5

.text
.global main
main:
# Variant 1 (absolute)
lui a0, %hi(a)
lw a1, %lo(a)(a0)

# Variant 2 (PC-relative, using relocations)
10:
auipc a2, %pcrel_hi(a)
lw a3, %pcrel_lo(10)(a2)

# Variant 3 (assembler shorthand expanded into a PC-relative sequence)
lw a4, a

# Variant 4 (relative to the global pointer)
lw a5, 0(gp)
```

Note! In this simulator, the program can run without an explicit `exit(0)` call, since there is no operating system to return control to.

In Ripes, the locations of the `.data` and `.text` sections are known in advance, and the symbol `a` is defined in the same source file. We can therefore experiment with all the variants presented above. For the programs implemented throughout this laboratory, we will use Variant 3. **PC-relative relocations currently have limited support in Ripes and may not behave correctly.**

Exercise: Analyze the four variants above and determine the addresses and values obtained at each step. What is the difference between `%hi` and `%pcrel_hi`, and between `%lo` and `%pcrel_lo`? Why can PC-relative addressing also be used when the absolute addresses of the sections are not known in advance? Observe the incorrect behavior of Variant 2 in Ripes and compare it with Variant 3.

Exercise: Modify Variants 1, 2, and 4 so that registers `a1`, `a3`, and `a5` contain the address of `a`, rather than its value. Use `addi` instead of the corresponding load instruction.

Exercise: Modify the variants above to store a value instead of loading one, and determine which variants are correctly supported by Ripes. **Note!** The sequences presented above are valid RISC-V constructs; any incorrect behavior observed here is due to limitations of the simulator rather than the architecture itself.

3.6 System Calls

The mechanism for using system calls is similar to the one previously introduced for x86, with the main difference being the registers involved. To perform a system call, we use the following

instruction:

```
ecall
```

which uses the following registers:

- `a7` - stores the system call number;
- `a0` - `a6` - store the arguments.

For example, the system call number for `exit` is 93. Therefore, to perform the equivalent of `exit(0)`, we use the following sequence:

```
li a7, 93
li a0, 0
ecall
```

The system call numbers supported by **Ripes** can be found under **Help > System calls**.

Exercise: Declare a string and an integer in memory and print both values to the console.

3.7 Jumps

An important instruction for unconditional jumps is:

```
jal rd, imm
```

It stores the value `pc + 4` in register `rd` and jumps to the address `pc + imm`.

In general, for unconditional jumps we will use:

```
j label
```

which is a pseudoinstruction for:

```
jal x0, label
```

Since register `x0` cannot be modified, the value `pc + 4` is effectively discarded, while execution continues at the address corresponding to `label`.

Since symbols are involved once again, a relocation specific to this instruction can be used – **R_-RISCV_JAL**, which allows the symbol to be resolved to an address relative to the `pc`.

For conditional branches, unlike x86, RISC-V combines the comparison and the branch into a single instruction.

Instruction	Operands	Description
blt	rs1, rs2, imm	If $rs1 < rs2$ (signed), branch to $pc + imm$
bltu	rs1, rs2, imm	If $rs1 < rs2$ (unsigned), branch to $pc + imm$
bge	rs1, rs2, imm	If $rs1 \geq rs2$ (signed), branch to $pc + imm$
bgeu	rs1, rs2, imm	If $rs1 \geq rs2$ (unsigned), branch to $pc + imm$
beq	rs1, rs2, imm	If $rs1 == rs2$, branch to $pc + imm$
bne	rs1, rs2, imm	If $rs1 \neq rs2$, branch to $pc + imm$

Tabela 5: Main RISC-V conditional branch instructions.

Pseudoinstruction	Operands	Description
ble	rs1, rs2, imm	If $rs1 \leq rs2$ (signed), branch to $pc + imm$
bleu	rs1, rs2, imm	If $rs1 \leq rs2$ (unsigned), branch to $pc + imm$
bgt	rs1, rs2, imm	If $rs1 > rs2$ (signed), branch to $pc + imm$
bgtu	rs1, rs2, imm	If $rs1 > rs2$ (unsigned), branch to $pc + imm$
beqz	rs1, imm	If $rs1 == 0$, branch to $pc + imm$
bnez	rs1, imm	If $rs1 \neq 0$, branch to $pc + imm$

Tabela 6: Main RISC-V conditional branch pseudoinstructions.

Similarly, instead of an immediate value, we can use a symbol whose relative address is determined by resolving an **R_RISCV_BRANCH** relocation.

Consider the following example:

```
.data
.long 6
.long 7
equal_msg: .asciz "The numbers are equal\n"
different_msg: .asciz "The numbers are different\n"

.text
.global main
main:

lw a0, 0(gp)
lw a1, 4(gp)

beq a0, a1, equal
li a7, 4
la a0, different_msg
ecall
```

```

j end

equal:
li a7, 4
la a0, equal_msg
ecall

end:
li a7, 93
li a0, 0
ecall

```

Exercise: Analyze the program and how the symbols are converted into immediate values.

3.8 One-Dimensional Arrays

The general mechanism is similar to x86, but RISC-V does not provide a dedicated addressing mode for accessing array elements.

To illustrate this, consider the following example, which computes the sum of the elements of an integer array and stores the result in memory:

```

.data
n: .long 5
s: .long 0
v: .long 1, 2, 3, 4, 5

.text
.global main
main:
lw a0, 0(gp)      # load n into a0
addi a1, gp, 8    # store the address of v in a1
li a2, 0          # a2 will hold the running sum
li a3, 0          # a3 will hold the index

begin:
bge a3, a0, final

slli a4, a3, 2    # a4 = 4 * index
add a4, a4, a1    # a4 = address of v + 4 * index
lw a4, 0(a4)      # a4 = *(address of v + 4 * index)
add a2, a2, a4    # add the current element to the total sum
addi a3, a3, 1    # increment the index
j begin

final:
sw a2, 4(gp)

li a7, 93

```

```
li a0, 0
ecall
```

Exercise: Rewrite the previous example by first replacing the value of `n` with `n * 4`, and then incrementing the index by 4 at each iteration.

3.9 RV32C - Compressed Instructions

To improve code density and memory efficiency, RISC-V also provides a compressed instruction subset, which allows certain instructions to be represented using 16 bits instead of 32. This reduces program size and can also reduce instruction-fetch bandwidth.

RVC uses a simple compression scheme that allows instructions from the base instruction set to be represented in a shorter form when:

1. the immediate value or offset is sufficiently small;
2. one of the registers is `x0` (zero register), `x1` (return address), or `x2` (stack pointer);
3. the same register is used as both destination and source;
4. registers with compressed encodings are used (see Table 2).

Examples

```
lw t0, 4(sp)    --> c.lwsp t0, 4(sp)    // the sp register is used
addi t0, t0, 1  --> c.addi t0, 1        // source == destination == t0
lw a0, 4(s0)    --> c.lw a0, 4(s0)      // a0 and s0 have compressed encodings
lw t0, 516(sp)  --> not compressed     // the immediate is too large
```

The following instruction formats are defined for encoding compressed instructions:

Format	Meaning	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
CR	Register	funct4				rd/rs1				rs2				op					
CI	Immediate	funct3		imm		rd/rs1				imm				op					
CSS	Stack-relative Store	funct3		imm						rs2				op					
CIW		funct3		imm								rd'		op					
CL	Load	funct3		imm			rs1'			imm		rd'			op				
CS	Store	funct3		imm			rs1'			imm		rs2'			op				
CB	Branch	funct3		offset			rs1'			offset					op				
CJ	Jump	funct3		jump target												op			

Figure 3: RVC instruction formats ¹³

Compressed instructions can be distinguished from 32-bit instructions directly from their encoding. All standard 32-bit instructions have their two least significant bits set to 11, while compressed instructions use 00, 01, or 10. This allows the processor to determine the instruction length from its first bits and decode it accordingly. After identification and decoding, the instruction continues through the remaining processor stages.

¹³<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-209.pdf>

Instruction	Operands	Description
<code>c.addi</code>	<code>rd, imm</code>	Adds an immediate value to register <code>rd</code> .
<code>c.addisp</code>	<code>rd, imm</code>	Adds an immediate value to the <code>sp</code> register.
<code>c.add</code>	<code>rd, rs</code>	Adds the value in register <code>rs</code> to register <code>rd</code> .
<code>c.lwsp</code>	<code>rd, offset(sp)</code>	Loads a word from address <code>sp + offset</code> into register <code>rd</code> .
<code>c.swsp</code>	<code>rd, offset(sp)</code>	Stores a word from register <code>rd</code> at address <code>sp + offset</code> .
<code>c.li</code>	<code>rd, imm</code>	Loads an immediate value into register <code>rd</code> .
<code>c.j</code>	<code>offset</code>	Performs an unconditional jump using the specified offset.
<code>c.beqz</code>	<code>rs, offset</code>	Branches if register <code>rs</code> is equal to zero.
<code>c.bnez</code>	<code>rs, offset</code>	Branches if register <code>rs</code> is not equal to zero.
<code>c.not</code>	<code>rd, rs</code>	Computes the bitwise complement of the value in register <code>rs</code> and stores the result in register <code>rd</code> .
<code>c.lw</code>	<code>rd', offset(rs')</code>	Loads a word from address <code>rs' + offset</code> into register <code>rd'</code> .
<code>c.sw</code>	<code>rd', offset(rs')</code>	Stores a word from register <code>rd'</code> at address <code>rs' + offset</code> .

Tabela 7: Main RVC instructions and their operands

Some of the main compressed instructions are listed above. A complete table of compressed instructions can also be found here¹⁴.

To enable compressed instructions in Ripes, select the C extension from the same menu used to configure the processor.

Exercise: Rewrite the example from Subsection 3.8 so that the array is traversed from the last element to the first, using `4 * (n - 1)` as the initial index. What is the size of the `.text` section?

Exercise: Rewrite the previous code using compressed instructions. What is the new size of the code section?

¹⁴<https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/rvc-instr-table.html>

4 Calling Convention

To allocate space on the stack, we need to use arithmetic and memory access instructions. Although there are no instructions such as `push` and `pop`, the behavior is the same.

For example, to store the value from register `s0` on the stack, we will use the following sequence:

```
addi sp, sp, -4
sw s0, 0(sp)
```

Conversely, to remove the element from the top of the stack and store it in register `s0`, we will use:

```
lw s0, 0(sp)
addi sp, sp, 4
```

Note! This example only illustrates the basic mechanism of the stack. According to the standard RISC-V calling convention, the `sp` register must remain aligned to 16 bytes.

4.1 Calling a Procedure

To call a procedure, we can use two instructions:

```
call procedure
jal procedure
```

The latter is actually a pseudoinstruction for:

```
jal x1, procedure
```

which, as we recall, saves the address of the following instruction in register `x1`, i.e., register `ra` (return address). As its name suggests, this register is used to store the return address. Therefore, unlike x86, returning from a function is based on a register, with the return address no longer being stored on the stack.

The `call` pseudoinstruction allows a procedure located at a greater distance to be called and can be expanded into a sequence consisting of the `auipc` and `jalr` instructions.

Returning from a procedure is performed using the instruction:

```
ret
```

RISC-V also has a jump pseudoinstruction that we have not presented yet, namely:

```
jr reg
```

which jumps to the address stored in register `reg` and is itself a pseudoinstruction for:

```
jalr x0, reg, 0
```

the first operand specifying, as in the case of the `jal` instruction, the location where the address of the following instruction will be stored, while the last operand represents the offset.

As can be easily observed, the return instruction is therefore simply an alias for:

```
jr ra
```

Thus, as a basic skeleton for calling a procedure, we have the following sequence:

```
.data
.text
.global main
main:
    call proc    // jal proc
                // jal ra, proc

    li a7, 93
    li a0, 0
    ecall

proc:

    ret          // jr ra
                // jalr x0, ra, 0
```

Note! It appears that Ripes always starts execution at the first procedure it encounters, which is why we will always place the `main` function first.

4.2 Procedure Arguments and Return Value

In RISC-V, arguments are also passed through registers, according to the convention we have already seen for system calls. Registers `a0` - `a7` are used to pass the first 8 arguments. If there are more than 8 arguments, the remaining ones are stored on the stack, as in x86.

Return values are passed through registers `a0` and `a1`. For return values that do not fit into these two registers, a memory area reserved by the caller is used.

Example: Consider a function with the declaration `proc(x,y,&sum)`, which adds two integers `x` and `y`, stores the result in `sum`, and also returns the difference between the two numbers. At this point, a minimal program that calls this function with the arguments 3, 2, and the address of a variable in memory, respectively, is shown below.

```
.data
    x: .long 3
    y: .long 2
    sum: .long 0
    dif: .long 0

.text
.global main
main:
    lw a0, 0(gp)
    lw a1, 4(gp)
    addi a2, gp, 8
    call proc
```

```

sw a0, 12(gp)

li a7, 93
li a0, 0
ecall

proc:
    add a3, a0, a1
    sw a3, 0(a2)
    sub a0, a0, a1

    ret

```

4.3 Creating the Call Frame

As in x86, a frame pointer can be used to keep track of the current stack frame. The `fp` register (frame pointer) provides a fixed reference point within the frame. In the case of nested function calls, the `ra` register must also be saved to prevent the current return address from being overwritten. The `fp` register is callee-saved, while `ra` holds the return address of the current call and must be preserved if the function makes additional calls. The frame pointer is set to the value of `sp` at procedure entry, corresponding to the CFA (Canonical Frame Address), or the top of the frame. Ripes uses the register name `s0` for `fp`.

Thus, a procedure will have the following basic structure:

```

proc:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi s0, sp, 8

    // procedure body

    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret

```

In addition, the concept of callee-saved/caller-saved registers works in the same way as for x86. The type of each register is specified in Table 2.

Exercise: Observe the values stored in `s3` and `t5`, respectively, at the end of the program execution. Add the necessary instructions to also preserve the value of the caller-saved register `t5`.

```

.data
x: .long 3
y: .long 2

```

```

    sum: .long 0
    dif: .long 0

.text
.global main
main:
    li s3, 10
    li t5, 11

    lw a0, 0(gp)
    lw a1, 4(gp)
    addi a2, gp, 8
    addi a3, gp, 12
    call proc

final:
    li a7, 93
    li a0, 0
    ecall

proc:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    addi s0, sp, 8

    addi sp, sp, -4
    sw s3, -12(s0)

    add s3, a0, a1
    sw s3, 0(a2)

    sub t5, a0, a1
    sw t5, 0(a3)

    lw s3, -12(s0)
    addi sp, sp, 4

    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
    ret

```

Exercise: Modify the program to allocate and deallocate all 12 bytes at once, adjusting the offsets for `ra` and `fp` accordingly.

More information about calling conventions can be found here ¹⁵.

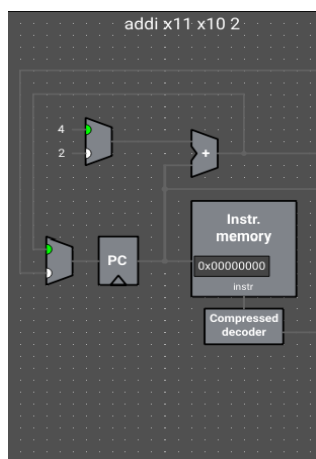
¹⁵<https://d3s.mff.cuni.cz/files/teaching/nswi200/202324/doc/riscv-abi.pdf>

5 Pipelining

Pipelining is a fundamental concept in modern processor design and is used in both RISC (Reduced Instruction Set Computing) and CISC (Complex Instruction Set Computing) processors. So far, we have used Ripes to simulate a single-cycle processor, in which one instruction is executed at a time and completes all processing within a single clock cycle. We will now analyze a 5-stage processor, starting with the **5-stage processor w/o forwarding or hazard detection** available in Ripes. Its pipeline consists of the following stages:

- **Instruction Fetch (IF)**: The instruction is fetched from memory.
- **Instruction Decode (ID)**: The instruction is decoded and the operands are read from the register file.
- **Execute (EX)**: The required operation is performed, typically using the arithmetic logic unit.
- **Memory Access (MEM)**: Data memory is accessed when required by the instruction, for example by load and store instructions.
- **Write Back (WB)**: The result is written back to the destination register (when required).

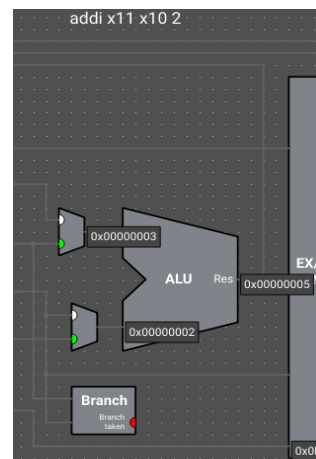
Let us now consider a program containing the instruction `addi a1, a0, 2`, assuming that `a0 = 3`.
 3. Let us examine how this instruction progresses through each stage of the pipeline.



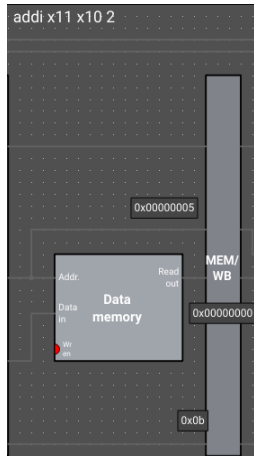
1. During the **fetch** stage, the instruction is fetched from the address indicated by the PC (0 in this case). Compressed instructions are decompressed, and the next PC value is determined based on the instruction length, which can be 2 or 4 bytes.



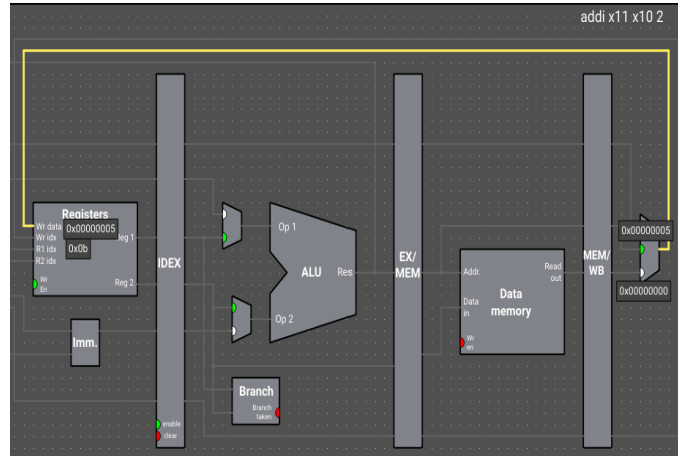
2. During the **decode** stage, the instruction is decoded and its operands and destination register are identified. The required values are read from the register file and the immediate is extracted. In this case, we obtain the value 3 from `a0` (register 10), a garbage value from register 2 (which will not be used), and the immediate 2.



3. During the **execute** stage, the required operands are sent to the ALU, which performs the operation specified by the instruction (addition with an immediate value). This is a simplified view; to also observe the control signals that determine the operation, you can select the extended view.



4. During the **memory** stage, data memory is accessed if the instruction requires a load or store operation. The **addi** instruction does not access memory, so the result computed by the ALU, 5 in this case, is simply passed to the next stage.



5. During the **write back** stage, the result is written to the destination register if required by the instruction. In this case, the result computed by the ALU, 5, is written to register **a1**.

Despite its advantages, pipeline performance depends on several factors that can introduce performance penalties.

5.1 Pipeline Hazards

A hazard is a problem that prevents the processor from executing instructions efficiently in the pipeline. There are three main types:

1. **Data hazards** - These occur when an instruction needs the result of another instruction and that result is not yet available. Some potential solutions are:
 - (a) **Data forwarding:** Results from intermediate stages are forwarded directly to the stages that need them, without waiting for them to be written to the register file.
 - (b) **Stalling:** The execution of an instruction is temporarily delayed until the required data becomes available.
 - (c) **Out-of-order execution:** An OoO processor decouples the order in which instructions are fetched and decoded (which remains in-order) from the order in which they are executed (which is out-of-order). By reordering instruction execution, the processor can reduce the latency introduced by hazards. Finally, instructions are kept in a buffer, and their results are committed in the original program order (*in-order commit*).
2. **Control hazards** - These occur in the case of branch instructions, when the processor does not know which instruction to fetch until the branch condition is evaluated. To reduce the

resulting penalties, **branch prediction** can be used to predict the direction of the branch and begin executing instructions along the predicted path. If the prediction is incorrect, the incorrectly fetched instructions are removed from the pipeline.

3. **Structural hazards** - These occur when two or more instructions attempt to use the same hardware resource at the same time (e.g., a single memory access unit). To avoid such conflicts, the processor can be designed with additional hardware resources.

Exercise: Consider the following memory declaration: `.long 200, 300, 400`. Analyze the pipeline stages for each of these instructions: `lw a0, 4(gp)`, `sw a0, 8(gp)`, and `beq a0, a1, 12`.

Let us now observe the pipeline when executing a sequence of instructions:

```
.data
.long 100, 200, 300
.text
.global main
main:
    lw a0, 4(gp)
    lw a1, 0(gp)
    add a0, a0, a1
    sw a0, 8(gp)
```

Exercise: What value is stored in memory? What happened?

Let us also consider the following code:

```
.data
.long 100, 200, 300
.text
.global main
main:
    lw a0, 0(gp)
    lw a1, 4(gp)
    blt a0, a1, label

    sw a0, 8(gp)
    j final

label:
    sw a1, 8(gp)

final:
    li a7, 93
    li a0, 0
    ecall
```

Exercise: What value is stored in memory? What happened?

To address these issues, two improved processor variants are provided, each implementing one of the following mechanisms:

1. **Pipeline stall** (5-Stage processor w/o forwarding unit) - **nops** (stalls) are introduced into the pipeline until the required source values are updated.
2. **Forwarding/Bypassing** (5-stage processor w/o hazard detection) - results are forwarded directly from the pipeline stages to the instructions that need them, before being written to the register file.

In this case, however, the second variant is still not sufficient, as additional cycles are required between dependent accesses. For both efficiency and correctness, we will therefore use the processor that implements both mechanisms (5-stage processor).

Exercise: Observe the advantage of the forwarding mechanism (5-Stage processor w/o forwarding unit vs 5-stage processor).

In addition, Ripes provides an extended logic diagram for a superscalar dual-issue processor, which allows two instructions to enter the pipeline simultaneously. Compared to the previous processor, it has 6 stages, with an additional **Issue** stage in which instructions are dispatched to the available functional units (ALU, memory access units).

5.2 Performance Measurement

1. **Throughput** - Measures how many instructions can be completed per unit of time. (5-stage processor - one instruction, 6-stage dual-issue processor - 2 instructions).
2. **Latency** - The total time required to complete a single instruction (from Fetch to Write Back - 5 vs. 6).
3. **CPI (Cycles Per Instruction)** - The average number of cycles required to complete an instruction. In a pipelined processor, the ideal CPI is 1, but in practice it can be higher due to hazards.
4. **IPC (Instructions Per Cycle)** - The actual number of instructions executed per clock cycle. A superscalar processor (dual-issue or quad-issue) can have an IPC greater than 1 ($IPC = 1/CPI$).
5. **Pipeline utilization** - The percentage of time during which the pipeline is fully occupied with useful work. Idle time means that resources are not being used efficiently.

Example: A program executes an equal number of arithmetic, memory access, and floating-point instructions. Assume that an arithmetic instruction takes 4 cycles, a memory access instruction takes 5 cycles, and an instruction using the floating-point unit takes 6 cycles. What is the **CPI**? What about the **IPC**? **Note:** This type of calculation ignores many other factors (for example, it assumes that the execution of these instructions does not overlap).

Answer:

$$CPI = \frac{1}{3} * 4 + \frac{1}{3} * 5 + \frac{1}{3} * 6 = 5$$

$$IPC = \frac{1}{5} = 0.2$$

6 Cache

Cache memory is a type of fast memory used to improve processor performance by reducing the time required to access frequently used data from main memory (RAM). The cache is usually located close to the processor and plays a crucial role in modern computing systems.

The cache acts as an intermediate memory between the processor and main memory, storing recently used data or data predicted to be needed in the future. In this way, it reduces memory access latency and improves the efficiency of instruction execution.

The cache exploits the principles of **spatial locality** (data located close to a recently accessed location is likely to be needed in the future) and **temporal locality** (recently accessed data is likely to be used again in the near future).

Exercise: Consider a 32-bit computing system with a 16-Kbyte cache. Assume that the memories are divided into 32-byte blocks. How many blocks are possible in main memory? How many blocks are there in the cache?

Cache memory is organized into multiple hierarchical levels, each with distinct characteristics (L1, L2, L3, etc.). Devices with limited capabilities may have no cache at all or may have fewer levels of cache memory. The Ripes simulator, for example, only provides L1 cache memory, which we will use to explore the main characteristics of this type of memory.

We will start by selecting a direct-mapped cache with 32 lines or blocks and a line size of 4 words (16 bytes), using the first preset. Let us first follow the algorithm used to determine the mapping between an address in main memory and its corresponding location in the cache.

Consider the code below (It is recommended to use the **5-stage processor**, which appears to be the most stable processor variant for presenting cache concepts):

```
.data
.long 0x10, 0x11, 0x12, 0x13
.long 0x14, 0x15, 0x16, 0x17
.long 0x18, 0x19, 0x20, 0x21
.text
.global main
main:
    lw a0, 20(gp)      #0x10000014 --> 0x15
    lw a1, 24(gp)      #0x10000018 --> 0x16
    add a0, a0, a1
    sw a0, 28(gp)      #0x1000001c --> 0x17
    lw a1, 0(gp)       #0x10000000 --> 0x10
    add a0, a0, a1
    sw a0, 32(gp)      #0x10000020 --> 0x18

final:
    li a7, 93
    li a0, 0
    ecall
```



```

.data
.long 0x10, 0x11, 0x12, 0x13
.long 0x14, 0x15, 0x16, 0x17
.long 0x18, 0x19, 0x20, 0x21
.text
.global main
main:
    lw a0, 8(gp)
    lw a1, 12(gp)
    add a2, a0, a1
    lw a0, 24(gp)
    lw a1, 28(gp)
    add a3, a0, a1
    lw a0, 40(gp)
    lw a1, 44(gp)
    add a4, a0, a1
final:
    li a7, 93
    li a0, 0
    ecall

```

Exercise: Let us now consider a 2-way set-associative cache with a line size of 2 and only 4 cache lines (2 sets). What do you observe?

When the cache is full, the processor must also decide which block to replace. Common replacement policies include:

- **LRU (Least Recently Used)** - The block that has not been used for the longest time is replaced.
- **FIFO (First In First Out)** - The block that was added first is replaced.
- **Random** - The block to be replaced is selected randomly.

Another cache characteristic (unfortunately not functional in Ripes) is the write policy:

1. **Write-through** - The processor modifies the data in the corresponding cache block, and the modification is also written to main memory.
2. **Write-back** - The processor writes the data only to the cache block and sets an indicator called the dirty bit. If the block needs to be replaced, the data is written to main memory if this bit is set.

Another policy related to writes is **(No) Write Allocate** - the decision of whether or not to bring the block into the cache on a write.

6.1 Cache Memory Performance

Cache performance is determined by:

1. **Hit rate** - The percentage of accesses that find the required data in the cache. A high hit rate significantly reduces memory access latency.
2. **Memory latency** - The time required to access data. Latency is lowest for L1 and increases for L2, L3, and main memory.
3. **Miss penalty** - The additional time required to obtain the data from the next level of the memory hierarchy when it is not present at the current level.
4. **Prefetching policies** - The processor can proactively fetch data that may be needed in the future based on access patterns.

7 Exercises

7.1 Laboratory 9

1. Given two natural numbers, x and y , write a program that swaps their values.
2. Given four numbers, a , b , c , and d , compute the result of the operation $((a+b+5)*2) \ll c | d$ and display it on the screen.
3. Determine the encoding of the instruction `and t0, s1, a5`.

7.2 Laboratory 10

1. Display the number of divisors of an integer stored in memory, followed by a list of its divisors. (“The number 6 has 4 divisors: 1, 2, 3, 6”).
2. Implement a procedure `divisors(int x)` that performs the operation described above and returns the number of divisors.
3. Implement a procedure `elements_with_divisors(int *v, int n, int k)` that iterates through the vector v , containing n elements, and returns the number of elements that have exactly k divisors. In `main`, display a message of the form “There are nr elements with exactly k divisors”. Use the procedure implemented in the previous exercise.

7.3 Laboratory 11

1. A processor has a 5-stage pipeline: Fetch, Decode, Execute, Memory, and Write-back. Determine how many cycles are required to execute 4 instructions if no hazards occur. What impact would a data hazard between instructions 1 and 2 have on the total number of execution cycles?
2. Consider a processor with the following instruction frequencies and costs:
 - Integer ALU – 50%, 1 cycle
 - Load – 20%, 5 cycles
 - Store – 10%, 1 cycle
 - Branch – 20%, 2 cycles

Calculate the CPI and determine which of the following changes would provide better performance:

- implementing a branch prediction mechanism to reduce the cost of branch instructions to 1 cycle;
 - using a data cache that reduces the cost of a load instruction to 3 cycles.
3. If a superscalar processor is introduced to handle data hazards, how many cycles can be saved for the following sequence of instructions:

```
ADD R1, R2, R3
SUB R4, R1, R5
MUL R6, R7, R8
```

4. The main memory has a size of 2^{16} bytes, while the cache has a capacity of 1 KB and a block size of 32 bytes. Calculate the total number of blocks in main memory. Calculate the number of cache lines. Determine where the address 0x3A7F will be mapped in the cache, assuming a direct-mapped cache.
5. A 2-way set-associative cache with 4 sets and a line size of 1 byte uses the LRU (Least Recently Used) replacement policy. Consider the following sequence of memory addresses: 0x0001, 0x0005, 0x0021, 0x0005, 0x0045. Explain how the data is organized in the cache after each access. Which block is replaced when address 0x0045 is accessed?

8 Additional Resources

- <https://riscv.org/educational-resources/>
- <https://www.allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture/>
- <https://www.cse.iitd.ac.in/~srsarangi/archbook/chapters/riscv.pdf>
- https://passlab.github.io/CSE564/notes/lecture09_RISCV_Impl_pipeline.pdf
- <https://nsec.sjtu.edu.cn/data/MK.Computer.Organization.and.Design.4th.Edition.Oct.2011.pdf>