A beginner's guide to RISC-V

Written by:

Besse Julien

Dugas Mateo

El Azzaoui Ismaïl

Mahalingasivam Sharujan

1. Introduction

2. Description of a RISC-V CPU
   - *a. The Program Counter (PC)*
   - *b. The Instruction Memory (IMem)*
   - *c. The Decode Logic*
   - *d. The Register File (RF)*
   - *e. The Arithmetic Logic Unit (ALU)*
   - *f. The Data Memory (DMem)*

3. So, what is RISC-V?
   - *a. Instructions*
   - *b. Registers*
   - *c. Applications*
   - *d. Instruction list*

4. Conditional statements and loops
   - A. A Python example
   - B. Preliminary steps
     - *a. Register's alias*
     - *b. Shifts*
     - *c. Logical operations*
     - *d. Labels*
     - *e. Pseudo-instructions*
   - C. Conditions
     - *a. Comparisons*
     - *b. If statements*
     - *c. Else statements*
   - D. While loops

5. Procedures and stack
   - *a. Procedure call*
   - *b. Caller/Callee convention and the stack*

6. Final exercise
   - *a. Creation of the collatz procedure*
   - *b. Creation of the sequence procedure*
   - *c. Creation of the convergence_speed procedure*

Annex: How to install and use RIPES

1. Introduction

RISC-V is an Instruction Set Architecture, developed in 2010 in the Par Lab of the university of California in Berkeley by Krste Asanović, Yunsup Lee and Andrew Waterman. As all the other projects coming from the Par Lab, it has been published as open source. However, it is not open source like a program could be, as there is no source code, but it is rather that everyone is free to use it and to propose improvements. This is one of the key elements that led to its success, as it does not actually bring any new technologies to the market, but it allows manufacturers to adapt it to their needs while dodging expensive license tariffs, like ARM licences.

The RISC-V foundation was founded in 2015 in order to promote and support the development of RISC-V, but not to control it, in accordance with its open-source philosophy. The foundation even decided to move in Switzerland following fears of trade protectionism from the USA.

Members of the organization include corporations, academic partners, and individuals, with the board of Directors being made of representative for each of those categories. The Board of Directors, alongside other committees such as, but not limited to, the Technical Steering Committee, the ISA Committees, or the Horizontal Committees, drive the development of RISC-V with the help of Task Forces created for specific purpose.

A lot of manufacturers, such as NVIDIA or Western Digital, have already begun to transition to RISC-V, and it is expected than in 2025, almost 68 billion devices will use the RISC-V architecture. It is, therefore, a must-have for any computer engineer entering the labour market.

⚠DISCLAIMER:

→The goal of this guide is not to make you a professional RISC-V developer, but to give you an idea of the stakes and the philosophy of this architecture.

→The examples shown in this document were chosen for clarity. You will see more efficient ways to write them in the course dedicated to RISC-V.

Motivation and Context:

This guide is meant to be an introduction to the basis of computer architecture using the RISC-V architecture as a reference.

Nowadays, the main components of a computer are the following: the CPU, the (main) memory and the input/output (I/O) interfaces (to interact with the user). Let us quickly describe the roles of each component:

- The CPU is the Central Processing Unit. It is the unit that runs the computer programs, it performs basic arithmetic and operation of control in order to execute all the process needed by the computer.
- The main memory is where all the data needed by the computer is stored. The user's data, the source code of downloaded programs, the operating systems files are all stored in the main memory.
- I/O interfaces are links between the computer and the devices that allow interaction with the user. The mouse and the keyboard, the microphone, and speakers are devices that use I/O interfaces.

As you know, the hardware of a computer can only read binary language. The layer just above binary is the assembly language which is a low-level programming language easier to apprehend than binary. Going from a code written from a high-level language, you will be able to transcribe it into assembly. Therefore, to make the most out of this guide, having some basics in Python would be beneficial.

Let us get into the heart of the subject. The Instruction Set Architecture (ISA) is the conceptual model of a computer. It is the boundary between the software and the hardware. The ISA is a standard that defines all the possible instructions that can be executed on a CPU and their translation in binary. An instruction is a single operation that can be done by the hardware.

The ISA implementation takes place in the Central Processing Unit (CPU) which is used to execute sequences of instructions (defined by the ISA) from a computer program.

On the other hand, a computer program is a set of instruction that can be run by the CPU. For instance, there is program that runs on your computer to track which keyboard keys are hit and translate that action into the correct response.

In this guide, you will find the following information:

- The first part is a brief description of a RISC-V CPU, it is hardware oriented and will give you an overall idea on how a general CPU works as well.
- Next, we have a description about RISC-V ISA, you will understand how instructions and registers works. And we will give you some basic examples.
- After that, the true low-level programming starts. You will learn how to create conditional statements and while loops.
- Then, the procedure calls will allow you to create sub-function in your assembly program, just like function calls in high level programming. And last but not least, an application about all you learned, via the translation of the Python code given in 4.A.
- To conclude, there is an annex about how to download the *Ripes* software, this tool is very useful if you want to practice the RISC-V architecture and its language.

2. Description of a RISC-V CPU

There are 6 main components in a computer that work together.

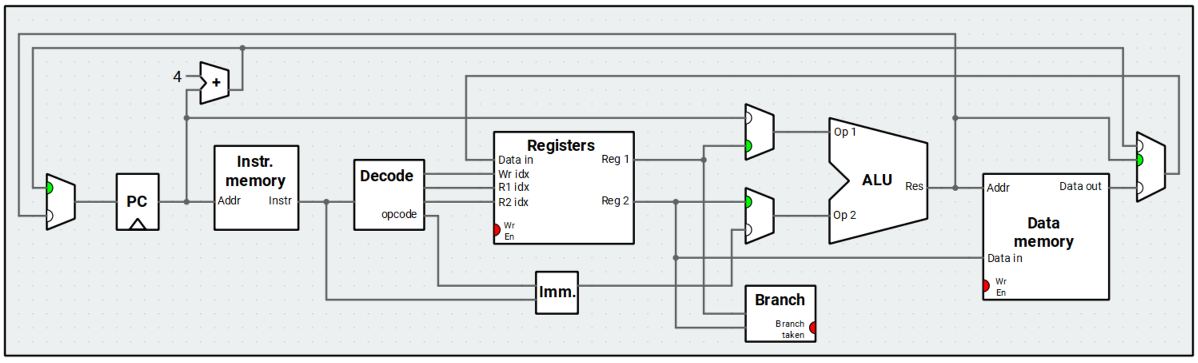
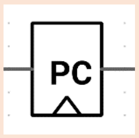


Figure 1: Components in a CPU [1]

a. *The program Counter (PC)*

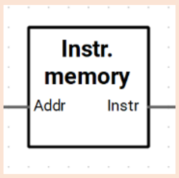


The PC determines which instruction should be executed next by calculating its address in the Instruction Memory.

When there is only sequential fetching, the PC can be a simple counter. But since instructions are 4 bytes long, and as the PC is referring to the first byte of an instruction, its value must be incremented by 4 at each step.

However, when it comes to assembly language, branch instructions exist: those instructions are mainly here to allow loops and the creation of callable sub-functions. With such instructions, the PC needs to be able to jump from one address to another directly (and not just slowly increase its value). This makes the Program Counter more complicated, but it is necessary to produce more advanced code.

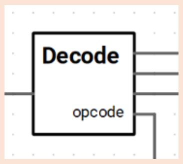
b. *The Instruction Memory (IMem)*



The Instruction Memory is **where the instructions**, of 4 bytes each, **are stored**. The role of this unit is also to select the instruction needed by the CPU. When the value of the PC changes, this information goes to the IMem; this allows the decode logic to have access to the correct instruction.

Instructions have different types; therefore, they will have to be decoded differently by the Decode Logic.

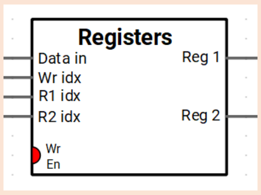
c. *The Decode and control unit*



An instruction is 32 bits long, and is composed of several groups of bits, that either represent what the operation should be or on what the operation operates.

This unit is used to break down and translate instructions from the memory into understandable parts.
The control unit is the unit which communicates with other component of the CPU to process the instruction. It also manages timing; it is a very important part of the CPU. In this image, the control unit and the decode unit are merged.
To put it in simple words, the Decoder Unit needs to break the 32 bits long instruction word, into smaller parts, that have a known meaning.

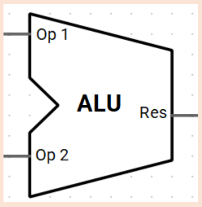
d. *The register File (RF)*



The register file stores 32 registers, each of these registers is 32 bits long, they are used as **a quick data storage**.
The first register (*x0*) is always wired as zero. The common RF is a 2-read/1-write model, which means it has 1 writing port and 2 reading ports. For each of these ports, there is an “enable” port. If we want to read two registers from the RF, we must assert the value read by both “read enable” ports.

The decode logic gives the signals needed to read from the RF. Then the RF outputs the value of the desired register to the ALU, which returns the result. The result will then be written back in the RF, at the address given by the decoder unit.

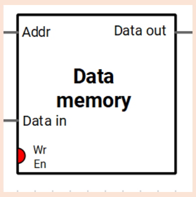
e. *The Arithmetic Logic Unit (ALU)*



The ALU computes the results of every possible instructions using source values from the register file and immediate value coming directly from the Decode Logic. Then, based on the actual instruction, it chooses the correct result.

This is where the useful work is done. From the simplest logical operation to a more advanced operation like addition or multiplication.

f. *The Data Memory (DMem)*



The Data Memory is simply the main memory of the CPU. When the result of an instruction needs to be stored, a store instruction is used. And reciprocally, a load instruction can fetch the value of an address in the DMem and write it in the requested register.

On classic devices, such as a computer, **there are more than one data memory**.

There is always a non-volatile memory, that can store the data even when the device is shut down and caches that are used to store recently used memory. As caches are faster than the non-volatile memory, those are in close contact with the CPU.

3. So, what is RISC-V?

a. Instructions

After this description, you may still be asking yourself: *what is* RISC-V?

The answer is quite simple: it is the **set of instructions** (ISA) that we described in the paragraph 2.c. Decoder logic. There are between 40 and 50 instructions depending on the version, and they allow you to achieve many operations. Those are the main bricks of programming.

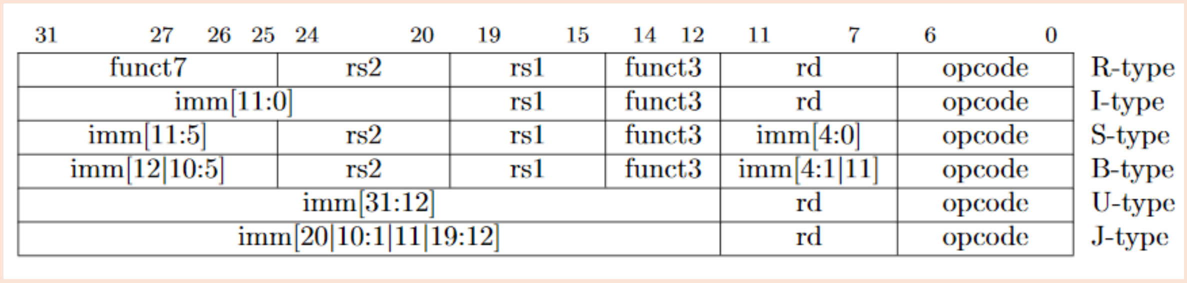
Two or three field represent what the instruction is supposed to do:

- opcode: this part is 7 bits long and describes the **type of instruction**. There are 6 types of instructions; each one is used in different circumstances.
- Function field (func3 and func7): they are used to indicate the **type of operation/function** that the instruction should accomplish.

Then are the field that are used directly by the function:

- rs1 and rs2: it indicates which register contains the **source operands** of the operation.
- rd: it contains the address of the register in which to **write the result** of the operation.
- immediate: this field is not always present, but whenever an instruction needs to contain information itself (such as the value to add to a register) it is stored in those bits.

There are different types of instructions which are:



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

Figure 2: Types of Instruction [2]

In assembly language, an instruction is written as the following: the mnemonic (name) of the instruction (*addi*, *sll*...), the register in which the result will be written, and two inputs that can be, registers or immediate values. **Global syntax: name *rd, rs1, rs2***

Finally: what is the purpose of RISC-V? It is also very simple: you are used to codes written in high-level language (python, java, etc), which are easy to read for humans, but incomprehensible for processors. For your code to be executed, it needs to be translated in assembly instructions: RISC-V is one of the options to do that. And the following step is converting those instructions into binary (only zeros and ones).

b. Registers

Before the first examples, we need to explain what registers are. **Registers are slots of memory in the Registers files of the CPU**, for RISC-V, they are 32 bits –4 bytes– long. Each register has an address, those address are noted *x0* through *x31*. Each of them is used by the CPU as a fast access and short-term memory.

Some of them have **special roles**, other ones are for local variables. Register *x0* is always 0. Registers from *x10* to *x17* are used for function arguments and return values, so you can use them at any time.
Later, we will learn the different roles of each of them.

c. *Applications*

First example:

To make things clearer, here is how an **add instruction** of our ISA is interpreted by the decoder logic.

The command line *add x10, x12, x13* should sum the values of register *x12* and *x13* and put the result in register x10.

Its binary code is the following: 00000000110101100000010100110011. This is not very understandable yet, but we know the 7 least significant bits are for opcode; this will allow us to find the type of function. In this case the opcode is 0110011, which corresponds to an **R-type command**.
With this information we can now split the rest of the binary accordingly:

Func7	Rs2	Rs1	Func3	Rd	Opcode
0000000	01101	01100	000	01010	0110011

As Func3 and Func7 are both equal to 0, with the instruction list in part d., we can confirm we have an add instruction. We also see that rs1 and rs2 represent, indeed, the source register (1100 and 1101 correspond to 12 and 13 in binary) and rd the output register, *x10*, as intended.

This method can be applied to all the instructions of the set.

As an exercise, you can try to translate this binary code 00000000110101010100010100110011, and convert in binary this command: *and x10, x17, x15*
Solutions: xor x10, x10, x13 and 00000000111110001111010100110011

Second example:
Another important category of instructions is the **branch instruction**. Those instructions allow the creation of more complex behaviour, such as conditional code and loops.

These instructions are all **B-type** (opcode 1100011).

Let us take an example: 00000000110101100100011001100011
We can easily split the binary into its components through the FIGURE 2.

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|12]	Opcode
0000000	01101	01100	100	01100	1100011

Func3 is 100, according to the list of instruction this is a *blt* instruction: Branch if Less Than. The 2 source registers are *x12* and *x13* and “immediate” contains the number 12.

This instruction does the following: if *x12* is less than *x13* then the next instruction to be executed will be the one that is stored 3 registers below the blt instruction (this is relative to the PC) — because instruction register are 4 bytes long, their addresses is always a multiple of 4, here 3*4 = 12.
The command line is: *blt x12, x13, label*.

Third example:

The **I-type instruction** are instructions that take as an input not 2 registers but only one, **the other input is encoded directly in the instruction, it is called an immediate value**. As shown in the array from earlier, there is 12 bits of data for the immediate argument: this means that the immediate argument can only go from -2048 to 2047.

The instruction *addi x10, x11, 2047* is written 01111111111101011000010100010011 in binary.

Imm[11:0]	rs2	funct3	rd	Opcode
011111111111	01011	000	01010	0010011

This instruction takes the value in *x11*, add 2047 to it, and places the result in *x10*.

This type of function is especially useful for loading constants in a code. Indeed, you can take *x0* as your first argument. And *x0* always is 0 so you will end up with the imm value in the result register.

Other instruction types:

There are other types of instructions: S-type to store, J-type for jumps and U-type for unconditional jumps.

Important instructions are sw for store word and lw for load word. **These two instructions are the ones that interact with the data memory**. For this you only need one thing, depending on the instruction: the address of the data you want to retrieve for lw and the address in which you want your data to be stored in for sw.

Example:

The syntax of these instructions is odd:
lw rd, imm(rs1)
- rd is the register where to take or put the data
- the value of register rs1 is the address in the main memory to look for
- imm is an immediate value, it's an offset from the rs1 value

lw x1, 4(x2) will take the value stored in the memory address indicated by '*value of x2 + 4*' and load it in *x1*: in this case *x1* will then have the value of memory 0xC (the fourth slot) hence 0x1.

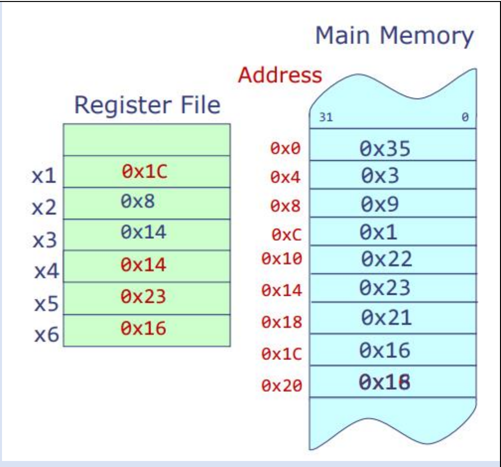


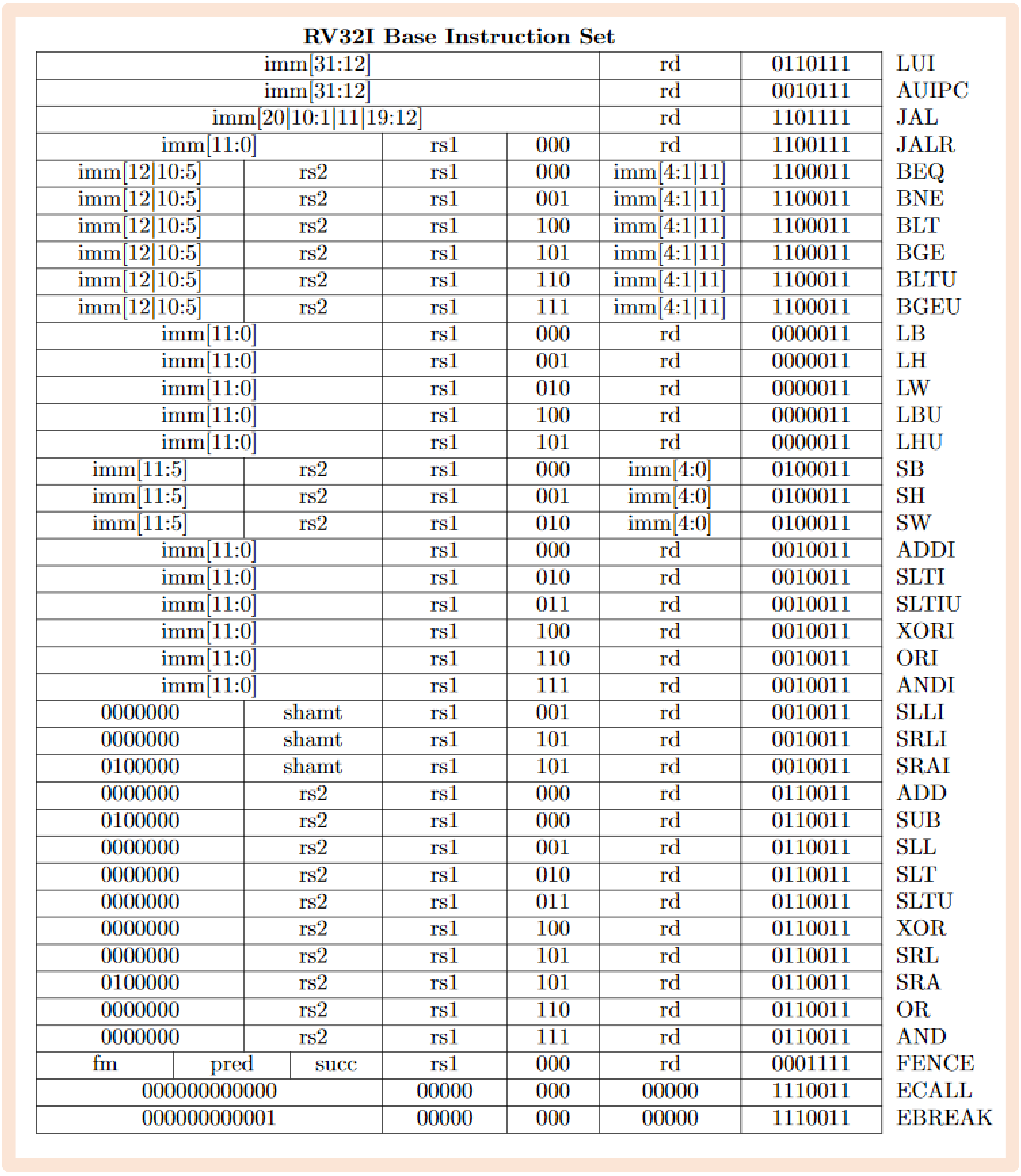
Figure 3: Register and Memory [3]

Be careful, the store instruction has the source operand as a first argument.
sw x1, 0(x2) will store the value of register *x1* in memory address '*value of x2*' hence in 0x8. Therefore, 0x8 will contain the value 0x1C.

sw x6, 8(x4) will store the value of *x6* in the address 0x1C (= *value of x4 + 8*), hence the value in 0x18 is 0x16

d. Instruction list

Here is the base set of instructions, to which extensions can be added:



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 3: Base Instruction Set [4]

4. Conditional statements and while loops

A. A Python example

```
def collatz(n):
    if n%2 == 1:
        return 3*n+1
    else:
        return n/2

def sequence(n):
    C=0
    while n != 1:
        n = collatz(n)
        C+=1
    return C

def convergence_speed(a, b): #a and b integers
    i = a
    idx_max = a
    max = sequence(a)
    while i<b:
        i+=1
        temp = sequence(i)
        if temp>max:
            idx_max = i
            max = temp
    return idx_max, max
```

This python code allows us to study the convergence speed of the **Collatz sequence** (Syracuse in French), defined as:

$$u_{n+1} = \begin{cases} 3u_n + 1, & u_n \text{ odd} \\ u_n/2, & u_n \text{ even} \end{cases}$$

An initial term u_0 from which the sequence does not converge to the sequence of number “4,2,1” has yet to be found. Our goal is to see how fast we get to “4,2,1” for different initial terms.

The first function calculates the term u_{n+1}, the second one computes at which N we get to $u_N = 1$ and the last one returns the maximum N of a range of integers between a and b.

At the end of this document, **you should be able to write these functions in RISC-V**.

B. Preliminary steps

a. Register's alias

Registers are referenced from *x0* to *x31*. However, in the Application Binary Interface, we can see that **each register has a defined purpose**. Hence, we rename them with aliases, to refer to them more easily. You don't know yet what each one of them are for, it's normal.

Here is the matching between registers and aliases:

Registers	Alias	Description
x0	*zero*	Hardwired zero
x1	*ra*	Return address
x2	*sp*	Stack Pointer
x3	*gp*	Global Pointer
x4	*tp*	Thread Pointer
x5 -x7/ x28-x31	*t0-t6*	Temporary registers
x8-x9/x18-x27	*s0-s11*	Saved registers
x10-x17	*a0-a7*	Function arguments (a0 and a1 are also used to return values)

b. Shifts

Logical shifts are operations which consist of shifting a binary word to the right or to the left a defined number of positions. By shifting to the right, **the value is divided** by a power of 2 and by shifting to the left, **the value is multiplied** by a power of 2.

However, we must be careful when doing a right shift, as the LSB (Least Significant Bit) disappears. Indeed, dividing an odd number by a power of 2 does not give an integer, hence, right shifting an odd number will give you the whole part of the result. Likewise, the left shift may cause a binary overflow if the multiplication by the power of 2 exceeds the magnitude that can be represented with the allotted number of bits.

In RISC-V, the instructions *sll*, *srl* (respectively for left and right logical shift) are used with one destination register and two source registers. The first source gets shifted as many times as the value stored in the second source. The result is then stored in the destination register.

The instructions are *slli* and *srli* can be used with an immediate value instead of the second source register.

For example, after executing the instruction sequence:

```
addi a0, x0, 0b110
srli a1, a0, 1
```

The value stored in *a1* will be 0b011. ($6/2 = 3$)

And, after executing this instruction sequence:

```
addi a0, x0, 0b00011010
addi t0, x0, 3
sll a2, a0, t0
```

The value stored in *a2* will be 0b11010000. ($26 \times 2^3 = 208$)

c. *Logical operations*

Logical operations are bitwise operations, they are directly supported by the processor.

Here are three important operations that can be done between two bits a & b:

	a	b	x	
AND	0	0	0	The result of the operation is 1 only when both bits are 1.
	0	1	0	
	1	0	0	
	1	1	1	
	a	b	x	
OR	0	0	0	The result of the operation is 1 if at least one of the bits is 1.
	0	1	1	
	1	0	1	
	1	1	1	
	a	b	x	
XOR	0	0	0	The result of the operation is 1 if only one of the two bits is 1.
	0	1	1	
	1	0	1	
	1	1	0	

Let us take an example: we want to know if a certain binary-encoded number is odd or even.

A binary word is odd if the LSB is 1, otherwise, it is even. Indeed, as each bit of the word is multiplied by a power of 2, the only power of 2 that is not even is 1 (2^0), hence why the LSB is the only bit that matters.

Now let us see what happens when we compute an *AND* operation between any binary word and 0b1. Every bit of 0b1 except the LSB are 0, so every bit of the result except the LSB are 0 no matter the number we are testing.
The LSB of the result will be 0 if the LSB of the tested number is 0, meaning it’s even. Otherwise, the result will be 1, which means that the number is odd.

If we want to test 23, which is binary encoded by 0b10111, we get:

a = 23	1	0	1	1	1
b = 0b1	0	0	0	0	1
a AND b	0	0	0	0	1

The LSB of *a AND b* is 1, which means that 23 is odd.

d. Labels

Labels are beacons in the code, when we “jump” to a label —using for instance the instruction *j*— it means that the next instruction executed will the one just following the label’s name.

```
(Previous instructions)

label_name:
(Following instructions)

j label_name
```

NB: you can also jump to a label written later in the code.

First code:

Now that you have a little understanding of the basics of the language, you can try to do this simple task: create a code that count from 1 to 10.

For this, you will need the instruction presented in the previous examples.

Try doing this on your own, and then come back here to see the solution!

Solution: Using *addi x12, x0, 0x01*, we set the value of the register *x12* to 1. We define a label lets_count_to_ten. The *add x10, x10, x12* instruction will sum the values in *x10* and *x12* and write it back in *x10*, as the value of *x12* is set to 1, this instruction will increment the value in *x10* by 1. Then, the instruction *blt x10, x11, lets_count_to_ten* will always bring back to the label lets_count_to_ten repeatedly until the value in *x10* becomes equal to the value in *x11* (which is 10).

```
addi x11, x0, 10
addi x12, x0, 1
lets_count_to_ten:
add x10, x10, x12
blt x10, x11, lets_count_to_ten
stop_the_count:
```

e. Pseudo-instructions

Pseudo-instructions are code syntaxes that do not belong to the base instruction set. But **they allow us to rapidly write simple instruction** that would require more symbol using only base instructions. Those instructions are directly translated by the assembler to base instruction.
This is used for clarity in the code, and to simplify instructions that are used all the time.

<u>Examples:</u>

```
# The li instruction stands for load immediate
# It provides a simple way to initiate values into register

# The two following lines have the same behavior
# They actually even have the same binary translation
li a0, 7
addi a0, x0, 7
```

C. Conditions

a. Comparisons

Set instructions	
These instructions will set the value of the return register to 1 if the comparison is true, 0 otherwise. There are three versions of each test: one is to compare two registers, the other one compares directly to zero (those are pseudo instructions), and the third compares a register with an immediate value.	
Operation	Code
rs1 == rs2	*seq rd, rs1, rs2*
rs1 != rs2	*sne rd, rs1, rs2*
rs1 < rs2	*slt rd, rs1, rs2*
rs1 > rs2	*sgt rd, rs1, rs2*
rs1 == 0	*seqz rd, rs1*
rs1 != 0	*snez rd, rs1*
rs1 < 0	*sltz rd, rs1*
rs1 > 0	*sgt rd, rs1*
rs1 < imm	*slti rd, rs1, imm*
Logical operation instructions	
You can also use logical operation instructions to combine different set instructions. Remember that 1 means true and 0 means false.	
Operation	Code
OR	*or rd, rs1, rs2*
OR immediate	*ori rd, rs1, imm*
XOR	*xor rd, rs1, rs2*
XOR immediate	*xori rd, rs1, imm*
AND	*and rd, rs1, rs2*
Branch instructions	
Branch instructions do the same comparisons as set instructions, but when true, they will send us to a label (called *label* in this example) instead of setting the value of the return directory to one. Most of the time you will only use these.	
Operation	Code
rs1 == rs2	*beq rs1, rs2, label*
rs1 != rs2	*bne rs1, rs2, label*
rs1 < rs2	*blt rs1, rs2, label*
rs1 >= rs2	*bge rs1, rs2, label*
rs1 == 0	*beqz rs1, label*
rs1 != 0	*bnez rs1, label*
rs1 < 0	*bltz rs1, label*
rs1 <= 0	*blez rs1, label*
rs1 > 0	*bgtz rs1, label*
rs1 >= 0	*bgez rs1, label*
jump to the label: *label*	*j label*

Examples:

```
#Suppose t4 is 2, and t5 is 3 and label is a label
seq t3, t4, t5 # t3 = 0
slt t3, t4, t5 # t3 = 1
bne t4, t5, label # we are sent to label
```

b. If statements

We do not need more instructions to write an if statements: we will simply use the comparisons and branch instructions described in the last paragraph. Their structure is very simple:

Syntax:

```
(set/logical instruction to make the test, put its value into register xN)
beqz xN, endif #if the test is false, we are sent directly to the end
    (If-body) #Otherwise we compile the (if-body)
endif:
```

⚠**You don't need to indent in RISC-V. We do it in the following examples only to facilitate the comparison with python.**
You can use any branch instruction instead of the *beqz* instruction.

Explanation:
After you do the test, *xN* will have the value 0 or 1. If it's 1, the branch instruction *beqz* is false, so the line won't have any effect and the if-body will run. Otherwise, the branch instruction will send us directly to the endif, thus ignoring the if-body.

Example:

Remember the python code presented previously. In the collatz function we had:

```
if n%2 == 1:
    return 3*n+1
```

In RISC-V:

```
#The n value is stored in register a2
#The return value is stored in register a0
addi t0, x0, 1
and t1, t0, a2 #This equal 1 if a2 is odd, 0 if a2 is even
beqz t1, endif #This instruction branches if t1 is equal to zero
add a0, a0, a2
add a0, a0, a2 # Just adding n three time to the result
add a0, a0, a2
addi a0, a0, 1
endif:
```

Remark: in simple cases, we could also combine the test and the branch by using beq, bne, blt…

c. Else and elif

Now that we know how to do if statements, it will be very simple to add else and elif. Indeed, *endif* is just a naming convention we can name the label as we want. Let us see how it’s done:

Syntax:

```
(Put the test value in xN)
beqz xN, else # If the test if false we jump directly to the else
    (If-body) # Otherwise we compile the (if-body)
    j endif # And then we jump directly to the end to skip the (Else-body)
else: #We compile this part only if the test is false
    (Else-body)
endif:
```

Explanation: it works exactly as the if statement, except that instead of jumping to the end of the code when the test is false, we jump to another label called *else*. If the test is true, after compiling the if-body, we jump to the end of the code to skip the else-body.

Example:

Let us complete the example we began previously:

```
if n%2 == 1:
    return 3*n+1
else:
    return n/2
```

In RISC-V:

```
#The n value is stored in register a2
#The return value is stored in a0
addi t0, x0, 1
and t1, t0, a2
beqz t1, else
add a0, a0, a2
add a0, a0, a2
add a0, a0, a2
addi a0, a0, 1
j endif
else:
srli a0, a2, 1
endif:
```

You can now also probably see how we are going to do elif statements:

Syntax:

```
(Put the test value into register xN1) #use a set instruction
beqz xN1, elif #if xN1=0, we jump directly to the elif
    (If-body) #Otherwise we compile the (if-body)
    j endif #And we then jump directly to the end
elif: #We compile this part only if the first test is false
    (Put the test value into register xN2)
    beqz xN2, else: #if the second test is false, we jump to the else
        (Elif-body)
        j endif
else: #We compile this part only if both previous tests are false
    (Else-body)
endif: #The statement if finished once we compiled either if, elif or else
```

D. While loops

If you have well understood what we did previously, while loops are going to be very easy to understand. We will once again only use branches and conditions:

Syntax:

```
while: #beginning of the loop
    (Put the test value into xN) #use a set instruction
    beqz xN, end_while #if xN=0 we jump to the end of the loop
        (While-body) #Otherwise we compile the (while-body)
        j while #Then we jump back to the beginning and test again
end_while: #Once the loop is finished, we can continue
```

Explanation: This is very similar to if statements. The test is done inside the while loop, and as long as it's true we compile the while-body, and we jump back to the beginning of the loop. When the test is false, we jump outside the loop.

Example:

In python:

```
while n != 1:
    n-=1
    C+=1
```

In RISC-V:

```
#Let n be in a0 and C be in a1
#This is the straightforward while loop, as it follows the python patter
li t0, 1
while:
beq a0, t0, end_while
addi a0, a0, -1
addi a1, a1, 1
j while
end_while:
#Next instruction
```

```
#But this method uses less comparison so it's faster in the end
# You can check by yourself; this does the same thing as before
li t0, 1
j compare
while:
addi a0, a0, -1
addi a1, a1, 1
compare:
bne a0, t0, while
#Next instruction
```

5. Procedures and Stacks

a. Procedure calls

In the previous part, we learned how to do if and while statements. But programming also implies being able to call a function to do a certain task. This is what procedures and stacks are for!

In a RISC-V code, we may need to get to another point of the code to execute instructions written there using parameters and then come back to where we were. This is a **procedure call** (it corresponds to functions in high language programming). To do this, we can use instructions like *jal* that takes in argument:
→a register which stores the address to go after exiting the procedure.
→the address of the called procedure. (given by a label)
The return address is stored in the *x1*(*ra*) register by convention. Registers *x10-x17* (*a0-a7*) are used for the function arguments. Therefore, **a procedure can take up to 8 arguments**. For the return values, we use *a0* and *a1*.

jal and jr:

```
jal ra, label_name
```

is used to call a procedure. The instruction does two things: it stores the return address in the *ra* register and it jumps to the *label_name* label.

```
jr ra
```

on the other hand, is used to return from a procedure call. Indeed, this instruction jumps back to the instruction memory address stored in *ra*. This is a pseudo instruction for *jalr*.

Example:

```
# Main code
# The hexadecimal number correspond the address in the instruction memory
0x14 : jal ra, sum
# later
0x6C: jal ra, sum

# Here is the definition of the sum procedure
# its position in the code doesn’t matter

#In the first call, ra has the value 0x18
#And in the second one, ra has the value 0x70
sum:
add a0, a0, a1
jr ra
```

But we may encounter an issue: let’s imagine that a procedure A is called and that then, this procedure calls another procedure B. A, which was the callee in the first place, became the caller. A has its own return address stored in *ra*, but also need to put a return address for B in it.

That is why we need a place to store data and a convention about who needs to save what.

b. Caller/Callee convention and the stack

Caller/callee conventions

The RISC-V architecture **use a very strict set of rules** when it comes to procedure calls.

Indeed, **the caller** (the procedure that calls a sub-function) **and the callee** (the procedure that is called) **share the same 32 registers**. So, as the callee, you need to know which registers you can use without overwriting valuable data, and as the caller you need to be aware that some registers can be overwritten by the callee.

As 32 registers is not a lot of memory, we use the stack to store some additional information, and to save data during procedure calls.

The stack starts at the higher addresses in the memory (the end of the memory) and grows towards smaller memory addresses. **A procedure can save data on the stack at any time**. But we need to keep in memory a way to find the top of the stack. This is the role of the *x2* register (alias *sp* for stack pointer).

The *sp* register is used to store the memory address that is currently the top of the stack.

If a procedure wants to store some data during a procedure call, the method is the following:
→Modify the *sp* register to fit the new top of the stack.
→Store the data you need in the new available spots in the stack
→Do something else: call the f sub-procedure for instance
→Load back the data stored in the stack
→Bring back the stack pointer to its original value

Note: to add something to the stack, you simply need to add a negative value to the *sp* register. Indeed, the stack grows towards smaller memory addresses, which means that if *sp* is smaller, there is more space in the stack.

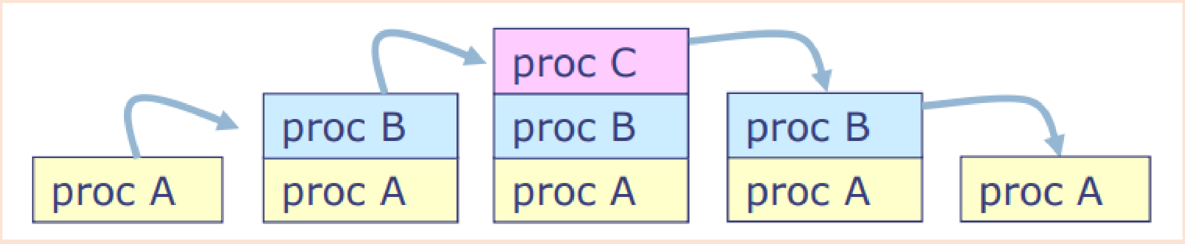


Figure 5: Caller/Callee in procedures[5]

Here the values stored by the procedure A are unchanged by the callee procedures, so when procedure A retrieves control, it can access its saved data as if nothing happened to the stack.

Callee responsibility:

The callee has some responsibility when it comes to returning the register to the caller: **the registers s0 through s11** (registers used specifically because they won't be modified by the callee function) **and the sp register must be returned to their initial state**. Therefore, if you need to modify the value of one of these registers, as the callee, it is your responsibility to store their value somewhere else and to restore it at the end.

This is done by using the stack. Let's say you need to store a temporary value on register s0 –you will overwrite it, how can you do that and respect the caller/callee convention?
You simply need to follow the steps described in the previous section and save the value of register s0. And never forget to return the *sp* register as you received it, since it is also a callee responsibility.

In this image, f is the name of the callee. And it needs to overwrite s0 and s1.

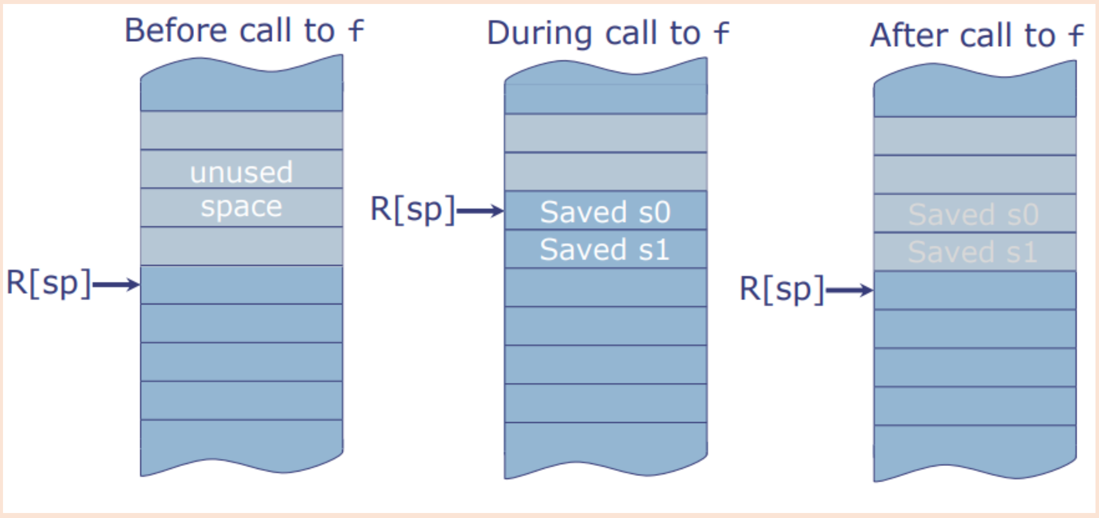


Figure 6 : Callee-saved registers [6]

Example:

```
# NOT a correct procedure since we have overwritten the value of s0 and not retrieve it
callee:
addi s0, a0, 10
sub a0, s0, a1
jr ra
```

```
# This is the correct way
callee:
addi sp, sp, -4 #Adding a spot in the stack
sw s0, 0(sp) #Using it to store s0
addi s0, a0, 10
sub a0, s0, a1
lw s0, 0(sp) #Loading back the original value of s0
addi sp, sp, 4 #Resotring sp at its original state
jr ra
```

Caller responsibility:

Caller-saved registers (arguments registers *a0-a7*, *ra* register, *t0-t6* temporary registers) could be overwritten by the callee across procedure calls. Therefore, **if the caller cares about their values, it must save them on the stack before giving control to the callee**.

The following example show a caller implementation:

Example:

```
# The idea is the same as earlier
# But now we are in the caller procedure, and we want to save ra and a0,
# As it’s in the caller responsibility
caller:
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
(Other instructions)
jal ra, callee
(Other instructions)
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, 8
jr ra # The caller is also often a callee
```

This caller/callee convention allow us to have all sorts of procedure calls, without them interfering with one another. You can check by yourself; **this convention allows several layers of calls and even recursive calls.**

6. Final exercise

Now that you have learn a lot about if-else statements, while loops and procedure calls, you can try to translate the full python code from earlier into RISC-V instructions. The goal here is for you to start working: a solution is given directly under the python code. You can use what you learned in this course to code those basic functions in assembly. Note that many of the previous examples can be used as a basis for this exercise.

The Annex can help you with the download of Ripes!
Also, we wrote **a line-by-line explanation of this procedure**, in the attached PowerPoint. This can help you better understand how this program and a RISC-V program works.

a. Creation of the collatz procedure

Never forget that as a convention the argument of a procedure is stored in a0. If there are two, we use a0 and a1, etc.

Here is a new pseudo instruction to help:

```
mv a0, a1 #This instruction is a pseudo-instruction
addi a0, a1, 0 # It does the same thing as this line
```

```python
def collatz(n):
    if n%2 == 1:
        return 3*n+1
    else:
        return n/2
```

```
collatz: # Beginning of the collatz procedure

mv a2, a0 # a2 takes the value of a0
addi t0, x0, 1 # t0 takes the value 1 (x0 = 0), we could also use “li t0, 1”
and t1, t0, a2 # equals 1 if a2 is odd

beqz t1, else # If a2 is even (i.e., equal 0), we jump to the else
add a0, a0, a2 # Otherwise a0 takes the value 3*a0+1
add a0, a0, a2
addi a0, a0, 1
j endif # And we then jump to the end of the procedure

else: # We compute this part only if a0 is even
srli a0, a2, 1 # We divide a0 by 2

endif:
jr ra #We jump out of the procedure
```

b. *Creation of the sequence procedure*

Again, some new pseudo instructions:

```
ret # which means
jr ra
```

```
def sequence(n):
    C=0
    while n != 1:
        n = collatz(n)
        C+=1
    return C
```

```
sequence:
addi sp, sp, -4 # Here we store the ra register for later use
sw ra, 0(sp) # Same here

li s0, 0 # We load initial values in s0, here s0 correspond to C
# We use s0 because we want to keep its value in between procedure calls

# Here we use on of the while loop syntax
j compare
while:
# a0 already contains n (because it is the argument of the procedure)
# So, no need to change anything, as the collatz procedure also take n as an
# argument
jal ra, collatz # Here is the call to collatz
# The result will be stored in a0, that is what we want

addi s0, s0, 1 # We increase the value of C by one
compare:
li t0, 1 # We temporarily store 1 in the t0 register
bne a0, t0, while # We use the 1 from the line before to make our comparison

mv a0, s0 # We move s0 (C in python) to a0
# because it’s the return value of the procedure

# Then we load back the value of ra that we stored in the stack.
# This operation allowed us not to care about the modification of ra
# register (during the jal instruction)
# We then restore the sp to its original value
lw ra 0(sp)
addi sp, sp, 4
ret #This is the pseudo-instruction for ‘jr ra’
```

c. Creation of the convergence_speed procedure

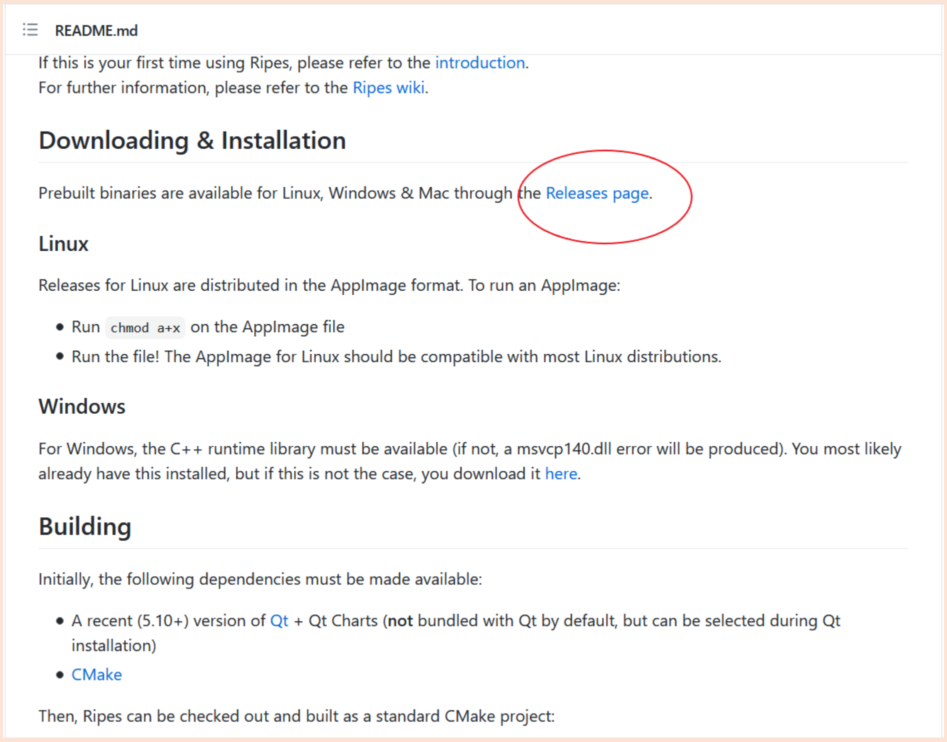
```python
def convergence_speed(a, b):
    i = a
    idx_max = a
    max = sequence(a)
    while i<b:
        i+=1
        temp = sequence(i)
        if temp>max:
            idx_max = i
            max = temp
    return idx_max, max
```

```
convergence_speed:
addi sp, sp, -4 # Store the ra value
sw ra, 0(sp)

# a0 contains a and a1 contains b
mv s0, a0 # s0 will contain the value of i
mv s1, a0 # s1 will contain the value of idx_max
mv s3, a1 # s3 contains the value of b (a1)

jal ra, sequence # sequence called with a0 (correspond to a) as argument
mv s2, a0 # s2 will contain the value of max

while_beginning: # the while loop
bge s0, s3, end_while
addi s0, s0, 1
mv a0, s0
jal ra, sequence

# The if statement, because there is nothing after it, we can jump back
# directly to the top of the while loop if the condition if false
bge s2, a0, while_beginning
mv s1, s0
mv s2, a0
j while_beginning

end_while:
mv a0, s1
mv a1, s2
ret
```

Annex: How to install and use RIPES

In order to get a linter (a tool that flags your syntax errors) for assembly language and a graphical simulator to view the instructions execution in a RISC-V processor, you may want to use *Ripes*.

The prebuilt binaries can be found in *this repository*.
You may access to the releases in the *Downloading & Installation* section of the README or directly *here*.



README.md

If this is your first time using Ripes, please refer to the introduction.
For further information, please refer to the Ripes wiki.

Downloading & Installation

Prebuilt binaries are available for Linux, Windows & Mac through the Releases page.

Linux

Releases for Linux are distributed in the AppImage format. To run an AppImage:

- Run `chmod a+x` on the AppImage file
- Run the file! The AppImage for Linux should be compatible with most Linux distributions.

Windows

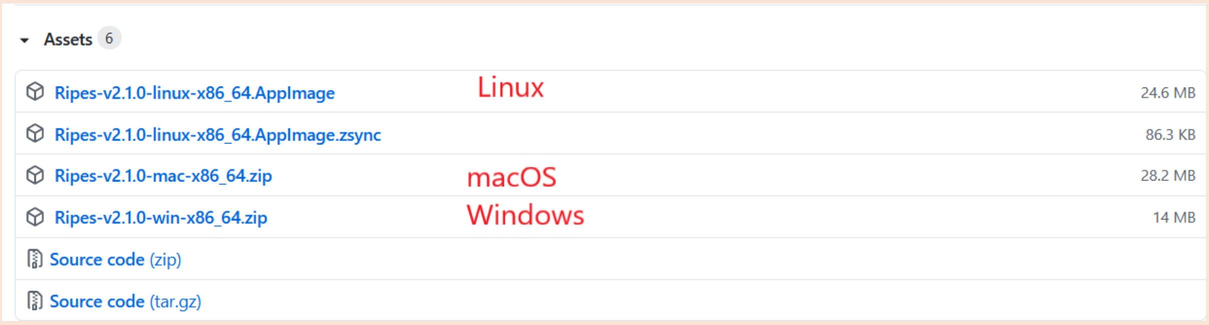
For Windows, the C++ runtime library must be available (if not, a msvcp140.dll error will be produced). You most likely already have this installed, but if this is not the case, you download it here.

Building

Initially, the following dependencies must be made available:

- A recent (5.10+) version of Qt + Qt Charts (**not** bundled with Qt by default, but can be selected during Qt installation)
- CMake

Then, Ripes can be checked out and built as a standard CMake project:

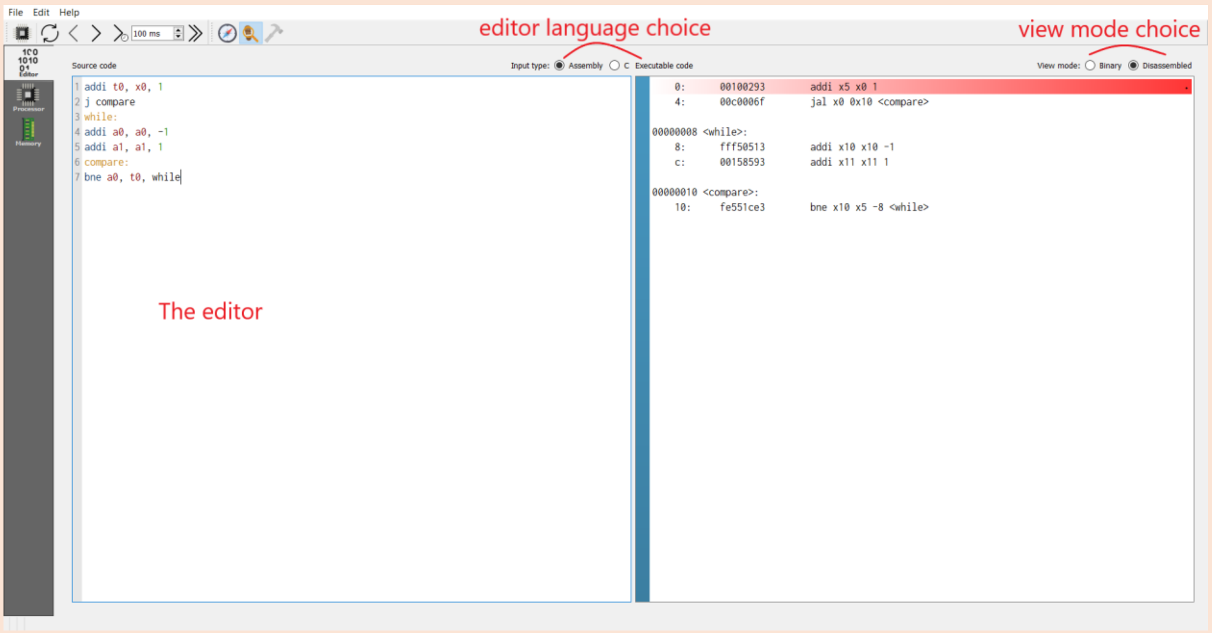


Assets 6

Ripes-v2.1.0-linux-x86_64.AppImage	Linux	24.6 MB
Ripes-v2.1.0-linux-x86_64.AppImage.zsync		86.3 KB
Ripes-v2.1.0-mac-x86_64.zip	macOS	28.2 MB
Ripes-v2.1.0-win-x86_64.zip	Windows	14 MB
Source code (zip)		
Source code (tar.gz)		

You can download the release according to your OS.
For Windows: In the folder you can launch the executable file called Ripes, it may return an “Untrusted source” error, you can execute the file anyway.
On Linux: you just have to run the downloaded file.
For MacOS: You have to execute the Mach-O file in the folder *MacOS*.

Here is the editor interface:



Source code

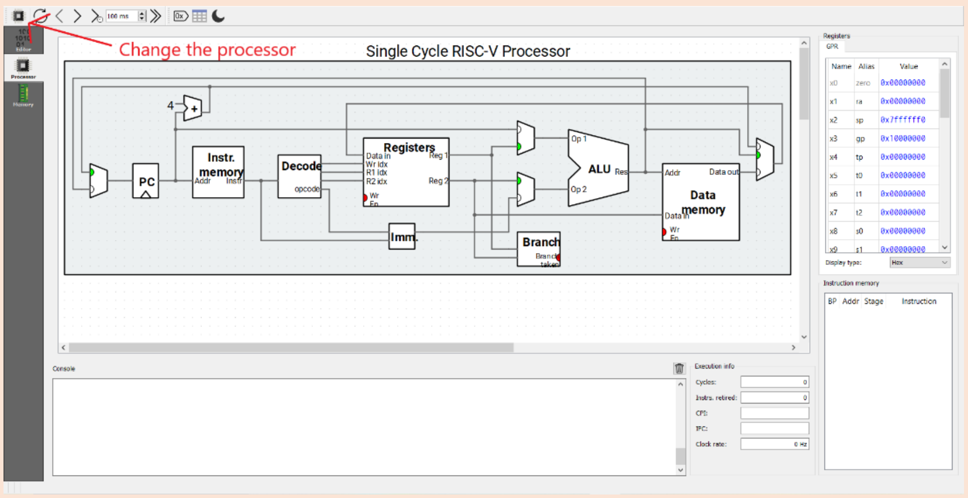
Input type: [✓ Assembly] C Executable code

View mode: Binary [✓ Disassembled]

```
addi t0, x0, 1
j compare
while:
addi a0, a0, -1
addi a1, a1, 1
compare:
bne a0, t0, while
```

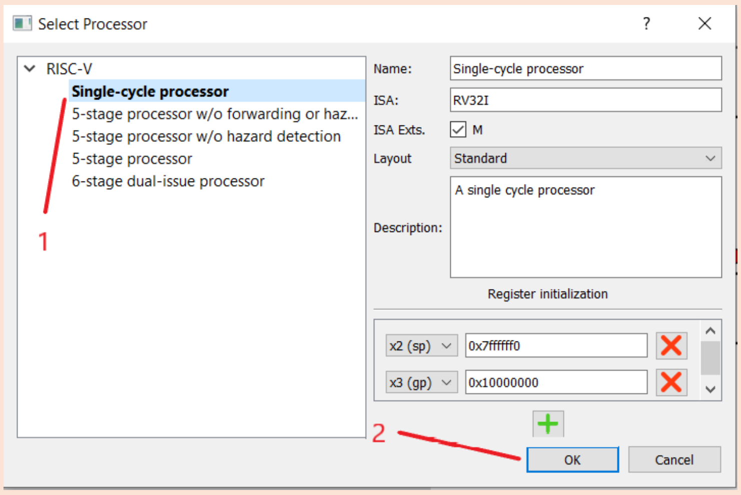
```
    0:      00100293      addi x5 x0 1
    4:      00c0006f      jal x0 0x10 <compare>

00000008 <while>:
    8:      fff50513      addi x10 x10 -1
    c:      00158593      addi x11 x11 1

00000010 <compare>:
   10:      fe551ce3      bne x10 x5 -8 <while>
```

As we do not use pipelining, you should use a Single Cycle Processor. To do that, click on the chip icon at the upper left of the window. Then, choose *Single-cycle processor* and validate by clicking on *OK.*



Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7ffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000000
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000

Cycles: 0
Instrs. retired: 0
Clock rate: 0 Hz



- Single-cycle processor
- 5-stage processor w/o forwarding or haz...
- 5-stage processor w/o hazard detection
- 5-stage processor
- 6-stage dual-issue processor

Name: Single-cycle processor
ISA: RV32I
ISA Exts. ✓ M
Layout: Standard
Description: A single cycle processor

Register initialization

x2 (sp)	0x7ffffff0
x3 (gp)	0x10000000

Sources

→Ripes, Graphical Simulator. https://github.com/mortbopet/Ripes [1]
→Building a RISC-V CPU Core, edx.org course. Cours | Building a RISC-V CPU Core | edX
→The RISC-V Instruction Set Manual https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf [2] [4]
→MIT course on RISC-V, available on YouTube:
MIT 6.004 L02: Introduction to Assembly and RISC-V - YouTube
MIT 6.004 L03: Compiling Code, Procedures, and Stacks - YouTube [3]
MIT 6.004 L04: Procedures, Stacks, and MMIO - YouTube [5] [6]