Project 1: RISC-V Instruction Set

Gabriel Gaspar
Alejandro Guijarro Monerris
Raynner Schnneider Carvalho

TELECOM Paris, Palaiseau

January 11th 2024

Contents

Chapter 1

RISC-V Instruction Set

I ISA and encoding of the RISC-V processor

We have the following program represented by the binary code of a simple function:

Listing 1.1: Function binary code

```
 0: 00050893
 4: 00068513
 8: 04088063
 c: 04058263
10: 04060063
14: 04d05063
18: 00088793
1c: 00269713
20: 00e888b3
24: 0007a703
28: 0005a803
2c: 01070733
30: 00e62023
34: 00478793
38: 00458593
3c: 00460613
40: ff1792e3
44: 00008067
48: fff00513
4c: 00008067
50: fff00513
54: 00008067
```

A Determine the instruction format and the operands for each instruction

In the table 1.1 we have written the corresponding instructions, as well as its format and the operands for each instruction (registers and immediate values). The format can be determined using the Risc-V instruction format detailed in 1.1.

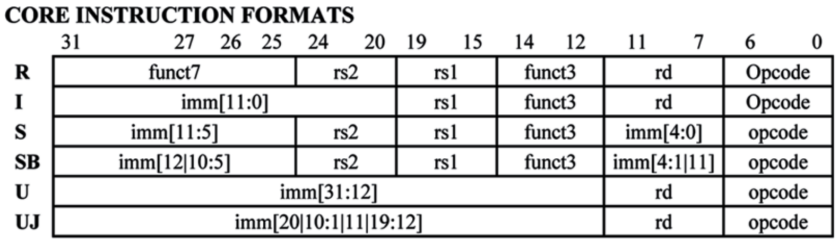

Figure 1.1: Core instruction formats

Table 1.1: Program instructions

Addr	Instr (Hex)	Instruction (asm)	Format	rd	rs1	rs2	imm
0:	00050893	addi a7, a0, 0	I	a7 (x17)	a0 (x10)	-	0x0
4:	00068513	addi a0, a3, 0	I	a0 (x10)	a3 (x13)	-	0x0
8:	04088063	beq a7, x0, 64	B	-	a7 (x17)	x0	0x40
c:	04058263	beq a1, x0, 68	B	-	a1 (x11)	x0	0x44
10:	04060063	beq a2, x0, 64	B	-	a2 (x12)	x0	0x40
14:	04d05063	bge x0, a3, 64	B	-	x0	a3 (x13)	0x40
18:	00088793	addi a5, a7, 0	I	a5 (x15)	a7 (x17)	-	0x0
1c:	00269713	slli a4, a3, 2	I	a7 (x14)	a3 (x13)	-	0x2
20:	00e888b3	add a7, a7, a4	R	a7 (x17)	a7 (x17)	a4 (x14)	
24:	0007a703	lw a4, 0(a5)	I	a4 (x14)	a5 (x15)	-	0x0
28:	0005a803	lw a6, 0(a1)	I	a6 (x16)	a1 (x11)	-	0x0
2c:	01070733	add a4, a4, a6	R	a4 (x14)	a4 (x14)	a6 (x16)	
30:	00e62023	sw a4, 0(a2)	S	-	a2 (x12)	a4 (x14)	0x0
34:	00478793	addi a5, a5, 4	I	a5 (x15)	a5 (x15)	-	0x4
38:	00458593	addi a1, a1, 4	I	a1 (x11)	a1 (x11)	-	0x4
3c:	00460613	addi a2, a2, 4	I	a2 (x12)	a2 (x12)	-	0x4
40:	ff1792e3	bne a5, a7, -28	B	-	a5 (x15)	a7 (x17)	-28
44:	00008067	jalr x0, 0(ra)	I	x0	ra (x1)	-	0x0
48:	fff00513	addi a0, x0, -1	I	a0 (x10)	x0	-	-1
4c:	00008067	jalr x0, 0(ra)	I	x0	ra (x1)	-	0x0
50:	fff00513	addi a0, x0, -1	I	a0 (x10)	x0	-	-1
54:	00008067	jalr x0, 0(ra)	I	x0	ra (x1)	-	0x0

The first instruction is 0x00050893, which is 00000000000001010000100010010011 in binary. We can split the code into different part:
000000000000|01010|000|10001|0010011: It tells us that it is an addition in I-format:

- 000000000000: it is the immediate value to add
- 01010: the source register for the addition, register a0 (x10)
- 000: it is the funct3 of the I-format
- 10001: the destination register of the operation, register a7 (x17)
- 0010011: it is the opcode, this indicates that the instruction is of I-format. Together with funct3 defines the instruction that should be executed. In this case an addi instruction.

The decoding of the rest of the instructions follow this same procedure, with the help of the instruction's format 1.1.

B Determine to which instructions the branch instruction jump

A branch instruction follows the SB-format 1.1. The destination address is computed from the immediate value as follows:

$$PC = PC + imm * 2 \tag{1.1}$$

The immediate is encoded in the instruction omitting the LSB, that is why we multiply by 2 (shift left by one bit) its value from the machine code.

With this in mind we can determine to which instruction a branch instruction jumps to. For each branch instruction its destination is indicated in the table 1.2.

Table 1.2: Branch instruction destination

Addr	Instruction	imm	Destination
8:	beq a7, x0, 64	0x40	0x48
c:	beq a1, x0, 68	0x44	0x50
10:	beq a2, x0, 64	0x40	0x50
14:	bge x0, a3, 64	0x40	0x54
40:	bne a5, a7, -28	-28	0x24

C What is the function actually doing? What is the return value?

This functions receives as input 3 values used as pointers (a0 to a2), and one value used as an integer (a3). After checking that the pointers do not hold the value 0x0, it loops *a3* times. During each iteration, it copies the sum from *(a0) and *(a1) to *(a2), and increments every pointer in order to point to the next word (4 bytes).

D Branch delay slots

In a pipeline processor, several instructions can be executed concurrently, though not in the same stages. It's similar to the process of doing laundry, as illustrated in the figure 1.2 below.

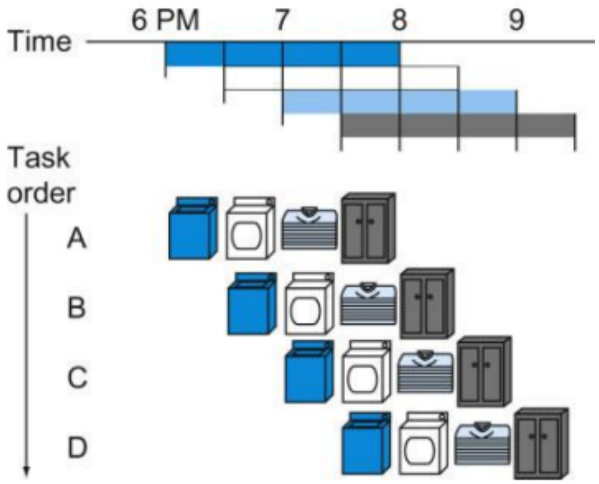

Figure 1.2: Pipeline analogy

In this example, we execute 4 instructions simultaneously. However, in electronics, there are instances where, during an instruction, a decision needs to be made regarding the next step. In such cases, particularly when encountering a branch instruction, one of the instructions may need to wait, leading to a potential delay, since it is necessary to verify if the branch will be taken or not. This is where the concept of a branch delay slot comes into play. Branch delay slots consist of instructions that can be executed independently while the branch instruction is waiting for it resolution. For each possible choice within an instruction, we execute a corresponding instruction. Once the actual choice is determined, we delay the execution of the unused instruction and replace it with the correct one at the address of the initial instruction (preceding the choice).

Chapter 2

RISC-V Tool Chain

I Interplay between compiler and computer architecture

A Write a C program matching the program above

Following the description of the behavior of the machine code, a C code implementation that follows the described behavior could be:

Listing 2.1: C code of the function

```
int foo(int* a0, int* a1, int* a2, int a3)
{
    if(a0 == 0 || a1 == 0 || a2 == 0)
        return -1;
    for(int i = 0; i < a3; i++)
        a2[i] = a0[i] + a1[i];
    return a3;
}
```

B Compile the program using the RISC-V compiler

Compiling with the command ‘riscv64-unknown-elf-gcc -g -O0 -mcmodel=medlow -mabi=ilp32 -march=rv32im -Wall -c -o se201-prog.o se201-prog.c‘ we get the following code:

Listing 2.2: Compiled code -O0

```
se201-prog.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <foo>:
   0:   fd010113                add     sp,sp,-48
```

```
   4:   02812623            sw      s0,44(sp)
   8:   03010413            add     s0,sp,48
   c:   fca42e23            sw      a0,-36(s0)
  10:   fcb42c23            sw      a1,-40(s0)
  14:   fcc42a23            sw      a2,-44(s0)
  18:   fcd42823            sw      a3,-48(s0)
  1c:   fdc42783            lw      a5,-36(s0)
  20:   00078a63            beqz    a5,34 <.L2>
  24:   fd842783            lw      a5,-40(s0)
  28:   00078663            beqz    a5,34 <.L2>
  2c:   fd442783            lw      a5,-44(s0)
  30:   00079663            bnez    a5,3c <.L3>

00000034 <.L2>:
  34:   fff00793            li      a5,-1
  38:   0680006f            j       a0 <.L4>

0000003c <.L3>:
  3c:   fe042623            sw      zero,-20(s0)
  40:   0500006f            j       90 <.L5>

00000044 <.L6>:
  44:   fec42783            lw      a5,-20(s0)
  48:   00279793            sll     a5,a5,0x2
  4c:   fdc42703            lw      a4,-36(s0)
  50:   00f707b3            add     a5,a4,a5
  54:   0007a683            lw      a3,0(a5)
  58:   fec42783            lw      a5,-20(s0)
  5c:   00279793            sll     a5,a5,0x2
  60:   fd842703            lw      a4,-40(s0)
  64:   00f707b3            add     a5,a4,a5
  68:   0007a703            lw      a4,0(a5)
  6c:   fec42783            lw      a5,-20(s0)
  70:   00279793            sll     a5,a5,0x2
  74:   fd442603            lw      a2,-44(s0)
  78:   00f607b3            add     a5,a2,a5
  7c:   00e68733            add     a4,a3,a4
  80:   00e7a023            sw      a4,0(a5)
  84:   fec42783            lw      a5,-20(s0)
  88:   00178793            add     a5,a5,1
  8c:   fef42623            sw      a5,-20(s0)

00000090 <.L5>:
  90:   fec42703            lw      a4,-20(s0)
  94:   fd042783            lw      a5,-48(s0)
```

```
  98:    faf746e3                blt     a4,a5,44 <.L6>

0000009c <.LBE2>:
  9c:    fd042783                lw      a5,-48(s0)

000000a0 <.L4>:
  a0:    00078513                mv      a0,a5
  a4:    02c12403                lw      s0,44(sp)
  a8:    03010113                add     sp,sp,48
  ac:    00008067                ret
```

It is considerably different from the original assembly code that the assignments gives us. The first thing that we can notice is the addition of the prologue, that is, a piece of code executed just after calling the function, in order to preserve the values of the used registers during the call, complying with the calling conventions. We also see that the compiler uses register a5 as the register used to do the initials comparisons, loading each time the required value onto this register from the stack (previously saved) and then doing the comparison to zero.
Depending on the compiler used, and the flags used, the machine code produced can vary.

Compiling with the command 'riscv64-unknown-elf-gcc -g -O -mcmodel=medlow -mabi=ilp32 -march=rv32im -Wall -c -o se201-progO.o se201-prog.c' we get the following code:

Listing 2.3: Compiled code -O

```
se201-progO.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <foo>:
   0:    00050893                mv      a7,a0
   4:    00068513                mv      a0,a3

00000008 <.LVL1>:
   8:    04088063                beqz    a7,48 <.L4>
   c:    04058263                beqz    a1,50 <.L5>
  10:    04060463                beqz    a2,58 <.L6>

00000014 <.LBB2>:
  14:    04d05463                blez    a3,5c <.L2>
  18:    00088793                mv      a5,a7
  1c:    00269713                sll     a4,a3,0x2
  20:    00e888b3                add     a7,a7,a4

00000024 <.L3>:
  24:    0007a703                lw      a4,0(a5)
```

```
  28:    0005a803                lw      a6,0(a1)
  2c:    01070733                add     a4,a4,a6
  30:    00e62023                sw      a4,0(a2)
  34:    00478793                add     a5,a5,4
  38:    00458593                add     a1,a1,4
  3c:    00460613                add     a2,a2,4
  40:    ff1792e3                bne     a5,a7,24 <.L3>
  44:    00008067                ret

00000048 <.L4>:
  48:    fff00513                li      a0,-1
  4c:    00008067                ret

00000050 <.L5>:
  50:    fff00513                li      a0,-1

00000054 <.LVL6>:
  54:    00008067                ret

00000058 <.L6>:
  58:    fff00513                li      a0,-1

0000005c <.L2>:
  5c:    00008067                ret
```

This time, we get almost the same code as the one we are given. The only exception is that the compiler added two extra lines for the function return, hence also changing some branches addresses. Other than that, the code is the same.

Compiling with the command 'riscv64-unknown-elf-gcc -g -O3 -mcmodel=medlow -mabi=ilp32 -march=rv32im -Wall -c -o se201-prog3.o se201-prog.c' we get the following code:

Listing 2.4: Compiled code -O3

```
se201-prog3.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <foo>:
   0:    00050793                mv      a5,a0
   4:    04050063                beqz    a0,44 <.L8>
   8:    02058e63                beqz    a1,44 <.L8>
   c:    02060c63                beqz    a2,44 <.L8>

00000010 <.LBB2>:
  10:    00269893                sll     a7,a3,0x2
```

```
  14:    011508b3                add     a7,a0,a7
  18:    02d05263                blez    a3,3c <.L5>

0000001c <.L4>:
  1c:    0007a703                lw      a4,0(a5)
  20:    0005a803                lw      a6,0(a1)
  24:    00478793                add     a5,a5,4
  28:    00458593                add     a1,a1,4
  2c:    01070733                add     a4,a4,a6
  30:    00e62023                sw      a4,0(a2)
  34:    00460613                add     a2,a2,4
  38:    ff1792e3                bne     a5,a7,1c <.L4>

0000003c <.L5>:
  3c:    00068513                mv      a0,a3

00000040 <.LVL3>:
  40:    00008067                ret

00000044 <.L8>:
  44:    fff00513                li      a0,-1

00000048 <.LVL5>:
  48:    00008067                ret
```

This code is an optimized version of the one we got previously. As we can see, the return instructions (0xfff00513, 0x00008067) only appear once, making the code shorter. Moreover, some initial register loads for comparing are skipped, saving some instructions.

Chapter 3

RISC-V Architecture

I Program Flow

The table 3.2 provides a list of all instructions that are executed to run the program from Question 1. It shows a brief explanation of what is being executed in each instruction and the value of registers *a0* to *a7*. Besides, the text highlighted by the color RED indicates that the content of this register changed and it is the new value.

It is important to mention that the memory content was previously considered according to the table 3.1.

Table 3.1: Initial memory content

Address	**Value**
0x200	0x61
0x204	0x20
0x208	0x62
0x20C	0x00
0x210	0x00

The address computations of branches followed what was explained in Chapter 1 Section B. Therefore, according to the expression 1.1, the new value of PC is calculated from the sum between the current address pointed by the PC and the immediate contained into the instruction.

Concerning the memory access in this program, it is done through the specific instructions for load word (*lw*) and store word (*sw*). The addresses that will be accessed are calculated from the sum between the value of *rs1* and the immediate.

The hazards and their mechanisms to solve them were explicitly mentioned in the Explanation column. In the following Section, Pipeline Diagram, it is possible to see in which stage of pipeline the instructions got a hazard.

Table 3.2: Instructions executed

PC	Instruction	a0	a1	a2	a3	a4	a5	a6	a7	Explanation
0x00:	addi a7, a0, 0	0x200	0x200	0x200	0x2	0x0	0x0	0x0	**0x200**	Sum between value of a0 and the immediate 0 into a7
0x04:	addi a0, a3, 0	**0x2**	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Sum between value of a3 and the immediate 0 into a0
0x08:	beq a7, x0, 64	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Does not branch to the address PC + immediate 64 (0x40) because the value of a7 is not equal to 0. Besides, there is a data hazard in this instruction because the result of *a7* in the instruction pointed by the PC = 0x00 is not available yet. Then, the solution is to use the *forwarding* from WB stage to EX stage.
0x0C:	beq a1, x0, 68	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Does not branch to the address PC + immediate 68 (0x40) because the value of a1 is not equal to 0
0x10:	beq a2, x0, 64	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Does not branch to the address PC + immediate 64 (0x40) because the value of a1 is not equal to 0
0x14:	bge x0, a3, 64	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	Does not branch to the address PC + immediate 64 (0x40) because 0 is not greater than or equal to the value of a3

0x18:	addi a5, a7, 0	0x2	0x200	0x200	0x2	0x0	**0x200**	0x0	0x200	Sum between value of a7 and the immediate 0 into a5
0x1C:	slli a4, a3, 2	0x2	0x200	0x200	0x2	**0x8**	0x200	0x0	0x200	Shift the bits of a3 by 2 bits to the left and store into a4, in others words, multiplication between value of a3 and 4 into a4
0x20:	add a7, a7, a4	0x2	0x200	0x200	0x2	0x8	0x200	0x0	**0x208**	Sum between value of a7 and value of a4 into a7. There is a data hazard in this instruction because the result of *a4* in the instruction pointed by the PC = 0x1C is not available yet. Then, the solution is to use the *forwarding* from MEM stage to EX stage.
0x24:	lw a4, 0(a5)	0x2	0x200	0x200	0x2	**0x61**	0x200	0x0	0x208	Load word from memory address in a5 and store it in a4
0x28:	lw a6, 0(a1)	0x2	0x200	0x200	0x2	0x61	0x200	**0x61**	0x208	Load word from memory address in a1 and store it in a6
0x2C:	add a4, a4, a6	0x2	0x200	0x200	0x2	**0xC2**	0x200	0x61	0x208	Sum between value of a4 and value of a6 into a4. There is a data hazard in this instruction because the result of *a6* in the instruction pointed by the PC = 0x28 is not available yet. Since that instruction is a *lw*, it was necessary to use the *stalling* in ID stage then a *forwarding* from WB stage to EX stage

0x30:	sw a4, 0(a2)	0x2	0x200	0x200	0x2	0xC2	0x200	0x61	0x208	Store word from a4 into memory address in a2. This instruction has a data hazard due to the instruction before, so it was necessary to use the *stalling* in IF stage. Besides, the *forwarding* was used from MEM stage to EX stage to use the value of *a4* updated
0x34:	addi a5, a5, 4	0x2	0x200	0x200	0x2	0xC2	**0x204**	0x61	0x208	Sum between value of a5 and the immediate 4 into a5
0x38:	addi a1, a1, 4	0x2	**0x204**	0x200	0x2	0xC2	0x204	0x61	0x208	Sum between value of a1 and the immediate 4 into a1
0x3C:	addi a2, a2, 4	0x2	0x204	**0x204**	0x2	0xC2	0x204	0x61	0x208	Sum between value of a2 and the immediate 4 into a2
0x40:	bne a5, a7, -28	0x2	0x204	0x204	0x2	0xC2	0x204	0x61	0x208	Branch to the address PC + immediate -28 because the value of a5 is not equal to value of a7. Since the branch was taken, a control hazard was generated.
0x44:	jalr x0, 0(ra)	-	-	-	-	-	-	-	-	Flushed due to a Control Hazard
0x48:	addi a0, x0, -1	-	-	-	-	-	-	-	-	Flushed due to a Control Hazard
0x24:	lw a4, 0(a5)	0x2	0x204	0x204	0x2	**0x20**	0x204	0x61	0x208	Load word from memory address in a5 and store it in a4
0x28:	lw a6, 0(a1)	0x2	0x204	0x204	0x2	0x20	0x204	**0x20**	0x208	Load word from memory address in a1 and store it in a6

0x2C:	add a4, a4, a6	0x2	0x204	0x204	0x2	**0x40**	0x204	0x20	0x208	Sum between value of a4 and value of a6 into a4. There is a data hazard in this instruction because the result of *a6* in the instruction pointed by the PC = 0x28 is not available yet. Since that instruction is a *lw*, it was necessary to use the *stalling* in ID stage then a *forwarding* from WB stage to EX stage
0x30:	sw a4, 0(a2)	0x2	0x204	0x204	0x2	0x40	0x204	0x20	0x208	Store word from a4 into memory address in a2. This instruction has a data hazard due to the instruction before, so it was necessary to use the *stalling* in IF stage. Besides, the *forwarding* was used from MEM stage to EX stage to use the value of *a4* updated
0x34:	addi a5, a5, 4	0x2	0x204	0x204	0x2	0x40	**0x208**	0x20	0x208	Sum between value of a5 and the immediate 4 into a5
0x38:	addi a1, a1, 4	0x2	**0x208**	0x204	0x2	0x40	0x208	0x20	0x208	Sum between value of a1 and the immediate 4 into a1
0x3C:	addi a2, a2, 4	0x2	0x208	**0x208**	0x2	0x40	0x208	0x20	0x208	Sum between value of a2 and the immediate 4 into a2
0x40:	bne a5, a7, -28	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208	Does not branch to the address PC + immediate -28 because the value of a5 is equal to value of a7

0x44:	jalr x0, 0(ra)	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208	Jump to the address saved in ra, in others words, it is the return. Since the branch was taken, a control hazard was generated.
0x48:	addi a0, x0, -1	-	-	-	-	-	-	-	-	Flushed due to a Control Hazard
0x4C:	jalr x0, 0(ra)	-	-	-	-	-	-	-	-	Flushed due to a Control Hazard

II Pipeline Diagram

The tables 3.3, 3.4, 3.5, 3.6 and 3.7 present the pipeline diagram showing all the instructions executed as shown by the table 3.2. All types of hazards that occurred, along with their respective resolutions, were highlighted in the tables to facilitate the interpretation of information.

Along the execution of this function is possible to find two types of hazards, being them Control Hazards, represented by the color BLUE, and Data Hazards, represented by the color ORANGE.

Intending to distinguish all mechanisms adopted to resolve these hazards, some colors were established to indicate them. At first, the color GREEN was used for FORWARDING. In sequence, the color YELLOW for STALLING. Finally, the color RED was used to represent FLUSH.

Table 3.3: Pipeline diagram highlighting the hazards - part 1

PC	Instruction	Cycle									
0x00:	addi a7, a0, 0	IF	ID	EX	MEM	**WB**					
0x04:	addi a0, a3, 0		IF	ID	EX	MEM	WB				
0x08:	beq a7, x0, 64			IF	**ID**	**EX**	MEM	WB			
0x0C:	beq a1, x0, 68				IF	ID	EX	MEM	WB		
0x10:	beq a2, x0, 64					IF	ID	EX	MEM	WB	
0x14:	bge x0, a3, 64						IF	ID	EX	MEM	WB
0x18:	addi a5, a7, 0							IF	ID	EX	MEM	WB

Table 3.4: Pipeline diagram highlighting the hazards - part 2

PC	Instruction	Cycle									
0x1C:	slli a4, a3, 2	IF	ID	EX	**MEM**	WB					
0x20:	add a7, a7, a4		IF	**ID**	**EX**	MEM	WB				
0x24:	lw a4, 0(a5)			IF	ID	EX	MEM	WB			
0x28:	lw a6, 0(a1)				IF	ID	EX	MEM	**WB**		
0x2C:	add a4, a4, a6					IF	**ID**	**ID**	**EX**	**MEM**	WB
0x30:	sw a4, 0(a2)						IF	**IF**	**ID**	**EX**	MEM	WB

,

Table 3.5: Pipeline diagram highlighting the hazards - part 3

PC	Instruction	Cycle									
0x34:	addi a5, a5, 4	IF	ID	EX	MEM	WB					
0x38:	addi a1, a1, 4		IF	ID	EX	MEM	WB				
0x3C:	addi a2, a2, 4			IF	ID	EX	MEM	WB			
0x40:	bne a5, a7, -28				IF	ID	**EX**	MEM	WB		
0x44:	jalr x0, 0(ra)					IF	**ID**	xx	xx	xx	
0x48:	addi a0, x0, -1						**IF**	xx	xx	xx	xx

Table 3.6: Pipeline diagram highlighting the hazards - part 4

PC	Instruction	Cycle									
0x24:	lw a4, 0(a5)	IF	ID	EX	MEM	WB					
0x28:	lw a6, 0(a1)		IF	ID	EX	MEM	**WB**				
0x2C:	add a4, a4, a6			IF	**ID**	**ID**	**EX**	**MEM**	WB		
0x30:	sw a4, 0(a2)				IF	**IF**	**ID**	**EX**	MEM	WB	
0x34:	addi a5, a5, 4						IF	ID	EX	MEM	WB
0x38:	addi a1, a1, 4							IF	ID	EX	MEM	WB

Table 3.7: Pipeline diagram highlighting the hazards - part 5

PC	Instruction	Cycle								
0x3C:	addi a2, a2, 4	IF	ID	EX	MEM	WB				
0x40:	bne a5, a7, -28		IF	ID	EX	MEM	WB			
0x44:	jalr x0, 0(ra)			IF	ID	**EX**	MEM	WB		
0x48:	addi a0, x0, -1				IF	**ID**	xx	xx	xx	
0x4C:	jalr x0, 0(ra)					**IF**	xx	xx	xx	xx

Chapter 4

Processor Design

I Instruction Set Architecture

We need to encode the instruction in 16 bits, and each instruction contain maximum 2 operands, 1 result and a code which define the operation. Each operand need to have 4 bits, so the result will have 4 bits too, and their is 4 bit left for the code of the operation, the Opcode.

A We can define 2 arithmetical operation:

addition (Opcode = 00) : rd = rs1 + rs2
subtraction (Opcode = 00) : rd = rs1 - rs2

B And 1 logical instruction:

AND (Opcode = 00) : rd = rs1 & rs1

C Define the instruction Load:

LOAD (Opcode: 01): rd = Mem[rs1 + SignExtended(imm)]

D Define the instruction Load:

STORE (Opcode: 01): Mem[rs1 + SignExtended(imm)] = rs2

E copy Immediate instruction in a register (rd):

(Opcode 10) : rd = imm

F Conditionnal branch instruction:

BEQZ (Opcode 11) : if (rs1 != 0) then PC = PC + (imm * 2); else PCnew = PCold + 2; (+2 Ã cause branch delay slot)

G Unconditionnal jump instruction:

(Opcode 11) : PC = rs1

H Call instruction:

Opcode (11) : r15(fixed register) = PC; PC = imm (9bits)

I Use of the register

- **R0-R3**: They are using for have variable needed in the functions but there is only 4 of them so just in the case that we have less than 5 argument otherwise we need to stock them in the pile and the are gonna call them one at the time (it take more time)
- **R4-R7**: They are used as temporary register internal for example if the function need a local variable
- **R8-R15**: Use so that we don't loose the variable already saved and that we are gonna need after the function is over by saving them in register we can be sure that they will keep the same value, the functions won't modified them, at the end of the function the variables are restored
- **R fixed**: We also need a fixed register to save the PC so that we can go back after the function is over (we saw in SE203 that we usually take R15), at the end of the function we go back to PC that have the return address inside, (we also saw in SE203 that if there is function inside functions, we will need a stack pointer because PC can only have one value)

J Describing the instructions

Instruction	Explication	Human form	Binary form	Format
ADD (Addition)	put inside rd the value of Rs1 + Rs2	add Rd, Rs1, Rs2	0000 DDDD SSSS RRRR	R-Type
SUB (Substraction)	put inside rd the value of Rs1 - Rs2	SUB Rd, Rs1, Rs2	0001 DDDD SSSS RRRR	R-Type
AND	put result in rd of a bitwise operation AND between RS1 and RS2	AND Rd, Rs1, Rs2	0010 DDDD SSSS RRRR	R-Type
LOAD	put the result of what is on the register Rs1 + the immediate value Imm	lw Rd, Rs1, Imm	0011 DDDD SSSS IIIII	I-Type
STORE	put Rd in the register number (value in Rs1) + Imm	sw Rs1, Rs2, Imm	0100 DDDD SSSS IIIII	I-Type
Copy Immediate	put the immediate value Imm inside Rd	COPY Rd, Imm	0101 DDDD IIIII	U-Type
Conditionnal branch	if Rs1 is different from 0 then PC = Pc + 2*Imm, else just add 2 to PC	BEQZ Rs1, Imm	0110 RRRR IIIIIIIIII	SB-Type
Unconditionnal jump	PC take Rs1 value	JUMP Rs1	0111 RRRR IIIIIIIIII	SB-Type
Call Instruction	PC take the value of Imm	CALL R[FixedRegister], Imm	1000 RRRR IIIIIIIIII	SB-Type

Figure 4.1: Instructions of our simple processor

K Instruction nop with the previous instructions

We can do the a nop with a AND, we just need to put in rd the result af rd & rd :
AND, Rd, Rd, Rd

L Assembly code of a function that add 2 arguments

SUM:
ADD R2, R0, R1 ; Result in R2
RET

M Assembly code that use the function before

MAIN:
COPY R0, 65408
COPY R1, 134
CALL R[FixedRegister], SUM;

N Translate the C-code from Question 1 to corresponding instructions of your processor.

ADDI a7, a0, 0
load a7, a0, 0; (because the immediate is 0)

ADDI a0, a3, 0
load a0, a3, 0;

BEQ a7, x0, 64
beqz a7, 64;

BEQ a1, x0, 68
beqz a1, 68;

BEQ a2, x0, 64
beqz a2, 64;

BGE x0, a3, 64
bge x0, a3, 64;

ADDI a5, a7, 0
load a5, a7, 0

SLLI a4, a3, 2
copy a4, 2
slli a4, a3, a4;

ADD a7, a7, a4
add a7, a7, a4;

LW a4, 0(a5)
load a4, a5, 0;

LW a6, 0(a1)
load a6, a1, 0;

ADD a4, a4, a6
add a4, a4, a6;

SW a4, 0(a2)
store a4, a2, 0;

ADDI a5, a5, 4
load a5, a5, 4;

ADDI a1, a1, 4
load a1, a1, 4;

ADDI a2, a2, 4
load a2, a2, 4;

BNE a5, a7, -28
bne a5, a7, -28;

JALR x0, 0(ra)
call x0, 0;

II Pipelining

Define the differents stages of the pipeline

IF Stage:
- Fetch the instruction from the memory location pointed to by the PC.

https://www.overleaf.com/project/65699909a760ef4d742fc722

ID Stage:
- (op code 01) Decode the instruction and determine the operation to be performed.
- Read register values needed for the execution.
- (op code 11)For conditional branches, unconditional jumps, and calls, perform the operation and update the PC accordingly.

EX Stage:
For arithmetic and logic instructions:
- Perform the operation.
- Write the result to the destination register.

* For memory-related instructions:
- Compute the effective address (for LOAD and STORE).
- Access memory using the computed address.
- Write the result to the destination register (for LOAD).

A Drawing of the processor

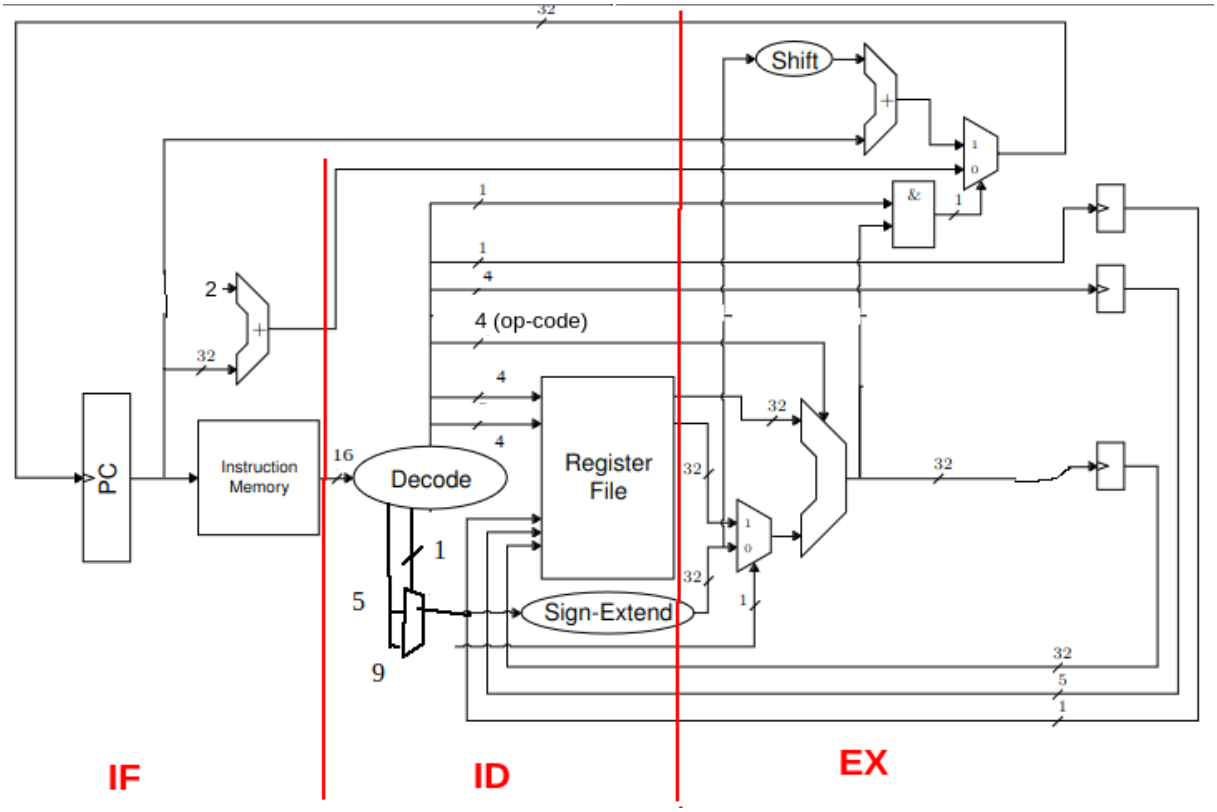

Figure 4.2: processor

Bibliography

[1] David A. Patterson and John L. Hennessy, *Computer Organization and Design RISC-V Edition*, Elsevier, 2017.

[2] *Understanding branch delay slot and branch prediction prefetch in instruction pipelining*, https://electronics.stackexchange.com/questions/433704/understanding-branch-delay-slot-and-branch-prediction-prefetch-in-instruction-pi, (accessed: 20.12.2023), keywords = "branch delay slot pipelining".