Télécom Paris

SE201: Execution Platforms
Project 1

Alaf do Nascimento Santos
Alessandro Mandrile
Ivan Luiz de Moura Matos

2022/2023

Contents

1 RISC-V Instruction Set

Aims: Understand the instruction set architecture and encoding of the RISC-V processor.

Table 1 shows the first results obtained, where considering the given RISC-V program represented by a hex code of a simple function, we have translated the instructions to binary so we could determine which RISC-V instructions appear in the program.

Address	Hex Instruction	Bin Instruction
0	00050893	00000000000000101000010001010011
4	00068513	00000000000000110100001010010011
8	04088063	00000100000010001000000001100011
C	04058263	00000100000001011000001001100011
10	04060063	00000100000001100000000001100011
14	04d05063	00000100110100000101000001100011
18	00088793	00000000000010001000011110010011
1C	00269713	00000000001001101001011100010011
20	00e888b3	00000000111010001000100010110011
24	0007a703	00000000000001111010011100000011
28	0005a803	00000000000001011010100000000011
2C	01070733	00000001000001110000011100110011
30	00e62023	00000000111001100010000000100011
34	00478793	00000000010001111000011110010011
38	00458593	00000000010001011000010110010011
3C	00460613	00000000010001100000011000010011
40	ff1792e3	11111111000101111001001011100011
44	00008067	00000000000000001000000001100111
48	fff00513	11111111111100000000010100010011
4C	00008067	00000000000000001000000001100111
50	fff00513	11111111111100000000010100010011
54	00008067	00000000000000001000000001100111

Table 1: Given RISC-V instructions

Table 2 summarises the second step of our methodology. It consisted in determining the format for each instruction and its operands. The method to get this information was based on the RISC-V documentation [2], starting by separating the binary code in order to identify each instruction format, we were able to define what the binary code really means for different types of instructions.

Address	funct7	funct3	opcode	function type	mne	rd	rs1	rs2	imm
0	-	000	0010011	I	addi	a7	a0	-	0x00
4	-	000	0010011	I	addi	a0	a3	-	0x00
8	-	000	1100011	SB	beq	-	a7	zero	0x40
C	-	000	1100011	SB	beq	-	a1	zero	0x44
10	-	000	1100011	SB	beq	-	a2	zero	0x40
14	-	101	1100011	SB	bge	-	zero	a3	0x40
18	-	000	0010011	I	addi	a5	a7	-	0x00
1C	-	001	0010011	I	slli	a4	a3	-	0x02
20	0000000	000	0110011	R	add	a7	a7	a4	-
24	-	010	0000011	I	lw	a4	a5	-	0x00
28	-	010	0000011	I	lw	a6	a1	-	0x00
2C	0000000	000	0110011	R	add	a4	a4	a6	-
30	-	010	0100011	S	sw	-	a2	a4	0x00
34	-	000	0010011	I	addi	a5	a5	-	0x04
38	-	000	0010011	I	addi	a1	a1	-	0x04
3C	-	000	0010011	I	addi	a2	a2	-	0x04
40	-	001	1100011	SB	bne	-	a5	a7	0x1fe4
44	-	000	1100111	I	jalr	zero	ra	-	0x00
48	-	000	0010011	I	addi	a0	zero	-	0xfff
4C	-	000	1100111	I	jalr	zero	ra	-	0x00
50	-	000	0010011	I	addi	a0	zero	-	0xfff
54	-	000	1100111	I	jalr	zero	ra	-	0x00

Table 2: Finding assembly instructions

Figure 1 illustrates the core instruction formats which gave us the starting point to develop the assembly code.

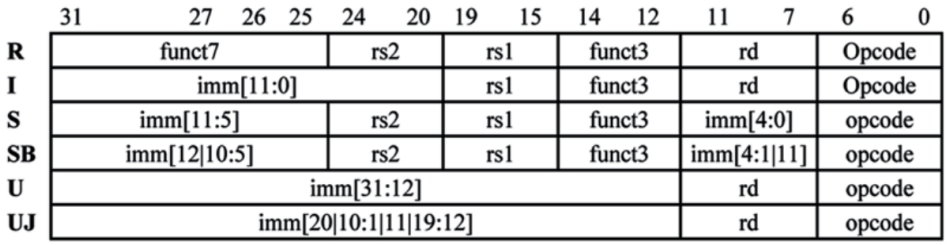

Figure 1: Core instruction formats

It is important to note that the conditional branch instructions SB already have their immediate offset extended with the implicit 0 bit and the `jalr` instructions, since they discard `PC` + 2 value and jump to `ra`, should be the expansion of the pseudo instruction `ret`. The spreadsheets that we developed in this step of the work are available on the file "SE201 - Project 1.xlsx" sent with this report. In the end of the analysis, we found the following code:

```
1       .text
```

```
2       addi    a7,a0,0x00
3       addi    a0,a3,0x00
4       beq     a7,zero, a0_eq_0 #[0x40 + PC] #0x48
5       beq     a1,zero, a1a2_eq_0 #0x50
6       beq     a2,zero, a1a2_eq_0 #0x50
7       bge     zero,a3, a3_le_0
8       addi    a5,a7,0x00
9       slli    a4,a3,0x02
10      add     a7,a7,a4
11  if_5:
12      lw      a4,0(a5)
13      lw      a6,0(a1)
14      add     a4,a4,a6
15      sw      a4,0(a2)
16      addi    a5,a5,0x04
17      addi    a1,a1,0x04
18      addi    a2,a2,0x04
19      bne     a5,a7, if_5 #0x24
20      ret     #jalr x0,ra
21  a0_eq_0:
22      li      a0,-1 #pseudo to addi a0,x0,-1
23      ret     #jalr x0,ra
24  a1a2_eq_0:
25      li      a0,-1 #pseudo to addi a0,x0,-1
26  a3_le_0:
27      ret     #jalr x0,ra
```

1.1 There are conditional branches in the function. Determine to which instructions they branch

We have put the obtained code in a file called program1.s to verify if it would generate the expected hexadecimal instructions (given in the assignment). For this step we created a Makefile, which can be read in Annex A.

By typing `make objdump`, we can generate the object file from the written assembly and then analyze the code generated by the compiler. Doing so and after discussing with the professors, we have realised that the branching instructions follow the Equation 1. It means that the next `PC` value in case of a validated condition will be the current `PC` (current address) added to twice the immediate value. This last part is caused by a shift left made on the immediate in order to reach the offset value.

$$PC_{next} = PC_{current} + 2 \times immediate \tag{1}$$

When we take a look on the assembly code generate by the compiler, we see branching addresses that seem to be wrong. Theoretically they are, but we would say the *objdump*

tool make some changes in the assembly code that it presents, in order to help us in the case of analyzing big programs. For example, the expectation according to the table we set up, for the branch of line 8, would be an offset of 40, but, instead, the compiler gave us 48 (which is, in fact, where it is located the exact target line of the branching).

```
 8:   04088063                beqz    a7,48 <a0_eq_0>
 c:   04058263                beqz    a1,50 <a1a2_eq_0>
10:   04060063                beqz    a2,50 <a1a2_eq_0>
14:   04d05063                blez    a3,54 <a3_le_0>
```

1.2 What is the function actually doing? What is its return value?

The function receives four parameters: three pointers to integer values and one integer. If one of the pointers is `null`, the function returns -1, otherwise it checks if the integer is greater than zero and then uses this value as a position counter for a loop. In this loop, one of the vectors pointed to by one of the pointers will receive the sum of the other two vectors pointed to by the remaining two pointers. In the end, the function returns the value of the integer that was passed as a parameter.

1.3 Branch Delay Slots

The branch delay slot was a feature of conditional branches on old MIPS architecture, which exploited the delays with which a branch was taken. With this feature instruction placed after the branch could be eventually executed while the branch was "waiting" to know its own verdict. The feature had to be supported by smarter compilers and consisted in placing after the branch instruction that logically should have been before, knowing that their effect would have effect before it. The kind of instructions that could be moved where those incrementing counters at the end of cycles (except if the counter is part of the condition) or instructions that would have been eventually executed in any condition. The size of the delay slots (number of instruction that could fit after a branch) was highly dependent on the complexity of the branch encoding and the length of the pipeline.

2 RISC-V Tool Chain

Aims: Understand the interplay between compiler and computer architecture

We start by writing the following C program matching the previous Assembly code. This version is faithful to the assembly code, a direct translation, but not very readable.

```
1  int func(int a0, int a1, int a2, int a3) {
2      int a7 = a0;
3
4      a0 = a3;
5
6      int a4, a5;
7
8      if (a7 != 0) {
9          if (a1 != 0 && a2 != 0) {
10             if (a3 > 0) {
11                 a5 = a7;
12                 a4 = a3 << 2;
13                 a7 += a4;
14
15                 while (a5 != a7) {
16                     a4 = *((int*)a5);
17                     int a6 = *((int*)a1);
18                     a4 += a6;
19                     *((int*)a2) = a4;
20                     a5 += 4;
21                     a1 += 4;
22                     a2 += 4;
23                 }
24
25                 return a0;
26
27             }
28
29             else
30                 return a0;
31         }
32
33         else
34             return -1;
35     }
36
37     else
38         return -1;
39 }
```

Then we summarize the previous version to make the code more readable. Both functions act in the same way, the second being written in a shorter way and for that reason we will call it short version.

```
1 int func(int* a0, int* a1, int* a2, int a3) {
2     if (a0 == 0 || a1 == 0 || a2 == 0)
3         return -1;
4     if (a3 > 0) {
5         for (int i = 0; i < a3; i++)
6             a2[i] = a0[i] + a1[i];
7     }
8     return a3;
9 }
```

2.1 Analysis of written codes

We want to see how different types of optimizations in the compilation of our codes can act on the generated assembly. Then, using our makefile A to facilitate the use of the compilation commands, we have checked the different generated codes via objdump.

By typing `make se201-prog.o`, we generated the object file from the written assembly for the first code C (long version), which gave us:

```
riscv64-unknown-elf-gcc -g -O0 -mcmodel=medlow -mabi=ilp32
-march=rv32im -Wall -c -o se201-prog.o se201-prog.c
```

So it was possible to disassemble the compiled program (se201-prog.o) with the objdump tool using the `make objdump1` command line, resulting in:

```
riscv64-unknown-elf-objdump -d se201-prog.o
```

In order to verify how the compiler works over different optimization options and to compare the resulting assembly code obtained from the `objdump` tool with the previous code, we tried the options -O0, -O and -O3 with our C codes.

The result obtained with -O0 for the long code can be seen in Appendix B, as follow for -O in Appendix C and finally -O3 in Appendix D. We did the same for the second code we have wrote, so the result obtained with -O0 for the long code can be seen in Appendix E, as follow for -O in Appendix G and finally -O3 in Appendix H.

We have applied for each optimization the command `size`, which lists the section sizes and total size of the binary files. For each optimization, we could verify that only the size of the `text` section that changes. Table 3 shows the different sizes we found. Since the data remains the same in all cases, the optimization will act on the number of instructions executed (`text` section) only.

	Text Section Sizes	
Optimization	**Short Version**	**Long Version**
-O0	184	260
-O	96	108
-O3	76	88

Table 3: Text Section Sizes

The code looks so different because the compiler can decide the best way to compile according to the given optimization option. These decisions depend on the developed algorithm, the processor and the architecture. The compiler will analyze many processor and architecture issues, such as latency between cycles, cache memory, cycles per clock, etc. Performance does not necessarily change when using different optimization options. This only has effect if the algorithm really makes sense for a given optimization, whose decision is exclusive to the compiler [1]. For example, some code breaks when using -O2 (not the case explored here).

As a conclusion of the analysis, we noticed that by using the option -O0, a lot of instructions are lost in the *push-pop* from the stack, so the option -O0 should not be used in this case. Finally, we tried to explore other optimization options, looking for one that would produce a result similar to the code we wrote. By doing so, we noticed that, when using the -O1 optimization option with the short version code, the resulting assembly code (which can be found in appendix F) was similar to the one corresponding to the hexadecimal code originally provided.

3 RISC-V Architecture

Aims: Understand RISC-V program execution on a pipelined processor.
We assume,

- The pipeline consists of 5 stages (IF, ID, EX, MEM, WB).
- Registers are read in the ID stage and written in the WB stage.
- Memory accesses are performed in the MEM stage.
  - The address computation is performed in the EX stage.
  - Data hazards between a memory load (in the MEM stage) and another instruction immediately using its results (in the EX stage) are resolved by stalling in the ID stage.
- Branches are performed in the EX stage. The two instructions following a branch are flushed when the branch is taken.
- For arithmetic instructions forwarding is performed as explained in the lecture

Based on the program from section 1, in to provide a list of instructions that are executed, along with a brief explanation of the processor/program state, we have assumed that the program starts with the following initial processor state:

- Registers `a0`, `a1`, and `a2` all have the value 0x200.
- Register `a3` has the value 0x2.
- All other registers have the value zero (0x0).
- Table 4 shows the memory contents at the address range 0x200 through 0x210 is given as follows:
- All other memory cells have a value of zero (0x0).

Address	**Value**
0x200	0x61
0x204	0x20
0x208	0x62
0x20C	0x0
0x210	0x0

Table 4: Memory contents at the address range 0x200 through 0x210

3.1 Program Flow

Table 5 shows the sequence of instructions that are computed, as well as the state of the registers `a0` to `a7` after the execution of each line. For each line, the cells with blue text indicate that the value of the corresponding registers has changed, when compared to the state before the execution of the instruction.

PC	Instruction	a0	a1	a2	a3	a4	a5	a6	a7
0x00	mv a7,a0	0x200	0x200	0x200	0x2	DC	DC	DC	0x200
0x04	mv a0,a3	0x2	0x200	0x200	0x2	DC	DC	DC	0x200
0x08	beqz a7,48 <a0_eq_0>	0x2	0x200	0x200	0x2	DC	DC	DC	0x200
0x0c	beqz a1,50 <a1a2_eq_0>	0x2	0x200	0x200	0x2	DC	DC	DC	0x200
0x10	beqz a2,50 <a1a2_eq_0>	0x2	0x200	0x200	0x2	DC	DC	DC	0x200
0x14	blez a3,54 <a3_le_0>	0x2	0x200	0x200	0x2	DC	DC	DC	0x200
0x18	mv a5,a7	0x2	0x200	0x200	0x2	DC	0x200	DC	0x200
0x1c	slli a4,a3,0x2	0x2	0x200	0x200	0x2	0x8	0x200	DC	0x200
0x20	add a7,a7,a4	0x2	0x200	0x200	0x2	0x8	0x200	DC	0x208
0x24	lw a4,0(a5)	0x2	0x200	0x200	0x2	0x61	0x200	DC	0x208
0x28	lw a6,0(a1)	0x2	0x200	0x200	0x2	0x61	0x200	0x61	0x208
0x2c	add a4,a4,a6	0x2	0x200	0x200	0x2	0xc2	0x200	0x61	0x208
0x30	sw a4,0(a2)	0x2	0x200	0x200	0x2	0xc2	0x200	0x61	0x208
0x34	addi a5,a5,4	0x2	0x200	0x200	0x2	0xc2	0x204	0x61	0x208
0x38	addi a1,a1,4	0x2	0x204	0x200	0x2	0xc2	0x204	0x61	0x208
0x3c	addi a2,a2,4	0x2	0x204	0x204	0x2	0xc2	0x204	0x61	0x208
0x40	bne a5,a7,24 <if_5>	0x2	0x204	0x204	0x2	0xc2	0x204	0x61	0x208
0x44	ret	-	-	-	-	-	-	-	-
0x48	li a0, -1	-	-	-	-	-	-	-	-
0x24	lw a4,0(a5)	0x2	0x204	0x204	0x2	0x20	0x204	0x61	0x208
0x28	lw a6,0(a1)	0x2	0x204	0x204	0x2	0x20	0x204	0x20	0x208
0x2c	add a4,a4,a6	0x2	0x204	0x204	0x2	0x40	0x204	0x20	0x208
0x30	sw a4,0(a2)	0x2	0x204	0x204	0x2	0x40	0x204	0x20	0x208
0x34	addi a5,a5,4	0x2	0x204	0x204	0x2	0x40	0x208	0x20	0x208
0x38	addi a1,a1,4	0x2	0x208	0x204	0x2	0x40	0x208	0x20	0x208
0x3c	addi a2,a2,4	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208
0x40	bne a5,a7,24 <if_5>	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208
0x44	ret	0x2	0x208	0x208	0x2	0x40	0x208	0x20	0x208
0x48	li a0, -1	-	-	-	-	-	-	-	-
0x4c	ret	-	-	-	-	-	-	-	-

Table 5: List of instructions that are executed

Table 6 presents a brief explanation of what is performed by each instruction.

PC	Instruction	Explanation
0x00	mv a7,a0	overwrite a7 with content of a0
0x04	mv a0,a3	overwrite a0 with content of a3
0x08	beqz a7,48 <a0_eq_0>	a7 is not equal 0, branch not taken
0x0c	beqz a1,50 <a1a2_eq_0>	a1 is not equal 0, branch not taken
0x10	beqz a2,50 <a1a2_eq_0>	a2 is not equal 0, branch not taken
0x14	blez a3,54 <a3_le_0>	a3 is not equal 0, branch not taken
0x18	mv a5,a7	overwrite a5 with content of a7
0x1c	slli a4,a3,0x2	shift left by 2 bits a3 into a4
0x20	add a7,a7,a4	add a7 to result of previous operation, data forward from EX
0x24	lw a4,0(a5)	load from memory at address (0 + a5) into a4
0x28	lw a6,0(a1)	load from memory at address (0 + a1) into a6
0x2c	add a4,a4,a6	add into a4 = a4 + a6
0x30	sw a4,0(a2)	save a4 into mem[0 + a2]
0x34	addi a5,a5,4	add into a5 = a5 + a4
0x38	addi a1,a1,4	add into a1 = a1 + a4
0x3c	addi a2,a2,4	add into a2 = a2 + a4
0x40	bne a5,a7,24 <if_5>	a7 is not equal a5, branch taken with value evaluated in EX stage with 0x40 +0x1FE4 = 0x40 - 0x1C = 0x24
0x44	ret	instruction is not completely executed
0x48	li a0, -1	instruction is not completely executed
0x24	lw a4,0(a5)	load from memory at address (0 + a5) into a4
0x28	lw a6,0(a1)	load from memory at address (0 + a1) into a6
0x2c	add a4,a4,a6	add into a4 = a4 + a6
0x30	sw a4,0(a2)	save a4 into mem[0 + a2]
0x34	addi a5,a5,4	add into a5 = a5 + a4
0x38	addi a1,a1,4	add into a1 = a1 + a4
0x3c	addi a2,a2,4	add into a2 = a2 + a4
0x40	bne a5,a7,24 <if_5>	a7 is equal to a5, branch not taken
0x44	ret	branch to the value in ra
0x48	li a0, -1	instruction is not completely executed
0x4c	ret	instruction is not completely executed

Table 6: List of instructions that are executed - Explanation

3.1.1 Hazards explanation

In the following, the hazards and their resolution are briefly explained:

- In the first execution of the instruction corresponding to the value 0x08 of `PC`, we have the occurrence of a *data hazard* during the execution of the `beqz` instruction, since the new value of the register `a7` (modified by the `mv` instruction corresponding to `PC` = 0x00) is not yet available to be read. This is resolved by doing a *forwarding* from the 0x00 instruction's MEM stage to the 0x08 instruction's EX stage.
- In the first execution of the instruction corresponding to the value 0x20 of `PC`, a *data hazard* occurs during the execution of the corresponding instruction (`add`), since the

new value of the register `a4` (used as an operand and modified by the previous `slli` instruction, at 0x1c) is not yet available to be read. This is resolved by doing a *forwarding* from 0x1c instruction's EX stage to 0x20 instruction's EX stage.

- In the first execution of the instruction corresponding to the value 0x2c of `PC`, a *data hazard* occurs during the execution of the corresponding instruction (`add`), since the new value of the register `a6` (modified by the previous `lw` instruction, at 0x28) is not yet available to be read. This is resolved by *forwarding* `a6` from the 0x28 instruction's MEM stage, but a *stall* during one clock cycle is needed.

- In the first execution of the instruction corresponding to the value 0x30 of `PC`, a *data hazard* occurs during the execution of the corresponding instruction (`sw`). This happens because the new value of `a4` (modified during the previous `add` instruction, at 0x2c) is not yet available. It is resolved by *forwarding* `a4` from the 0x2c instruction's MEM stage, and also by *stalling* at the IF stage, because ID is occupied.

- In the first execution of the instruction corresponding to the value 0x44 of `PC`, there is a *control hazard*, since the branch corresponding to the `bne` instruction at 0x40 is taken (because the tested condition is true). The instruction at 0x44 is *flushed* at the ID stage.

- In the first execution of the instruction corresponding to the value 0x48 of `PC`, there is a *control hazard*, since the branch corresponding to the `bne` instruction at 0x40 is taken. The instruction at 0x48 is *flushed* at the IF stage.

- In the second execution of the instructions corresponding to the values 0x2c and 0x30 of the `PC`, there are the same hazards described for the first execution of those instructions, and they are resolved in the same way.

- In the second execution of the instruction corresponding to the value 0x48 of `PC`, there is a *control hazard*, since the `ret` instruction at 0x44 is (completely) executed. The instruction at 0x48 is *flushed* at the ID stage.

- In the execution of the instruction corresponding to the value 0x4c of `PC`, there is a *control hazard*, since the `ret` instruction at 0x44 is (completely) executed. The instruction at 0x4c is *flushed* at the IF stage.

The graphical description of the hazards is shown in the tables presented in the Section 3.2, as well as the indication of the respective resolution mechanisms.

3.1.2 Address computations of branches

The target address computation of branches is made as described before, in Section 1.1: if the tested condition is true, the address of the next instruction (that is, the one that the branch must lead to) is calculated by summing the *offset* value (which is encoded by the value stored in the immediate field) to the current address (which is the address of the branch instruction, pointed by the `PC` register).

3.1.3 Memory accesses

Concerning the load and store instructions for memory accesses, which are used for transferring values between the registers and memory, the loads are I-type and stores are S-type. In the given function, we can find only `lw` and `sw` for memory accesses, where the `lw` loads a 32-bit (a word) value from memory into `rd`, while the `sw` store a 32-bit value from the register `rs2` to memory. Basically, the effective byte address is obtained by adding register `rs1` to the sign-extended 12-bit offset. The load copy a value from memory to register `rd`. Stores copy the value in register `rs2` to memory. Due to performance reasons, the effective address for those functions should be naturally aligned on a four-byte boundary.

3.2 Pipeline Diagram

In this section we want to show all the instructions executed by the function as determined before (see section 3.1). The way we found to represent that was drawing a pipeline diagram, represented in Tables 7 to 11. We have highlighted all forms of hazards that occur and graphically distinguished its resolution mechanisms. The blue cells represent *forwarding*, the gray ones represent *stalling* and the green ones represent *jumps*. In order to represent the hazards, we have painted with specific colors the text present in the cell where each hazard happened. We have decided to use red for *Control Hazards* and orange for *Data Hazard*. We decided to show hazards' indications one cycle before their respective solutions, because, by doing so, we have a clear vision of the state where our process would be if the observed problem had not been treated.

PC	Instruction	Cycle									
0x00	mv a7,a0	IF	ID	EX	MEM	WB					
0x04	mv a0,a3		IF	ID	EX	MEM	WB				
0x08	beqz a7,48			IF	ID	EX	MEM	WB			
0x0c	beqz a1,50				IF	ID	EX	MEM	WB		
0x10	beqz a2,50					IF	ID	EX	MEM	WB	
0x14	blez a3,54						IF	ID	EX	MEM	WB

Table 7: Pipeline diagram part 1

PC	Instruction	Cycle									
0x18	mv a5,a7	IF	ID	EX	MEM	WB					
0x1c	slli a4,a3,0x2		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

Table 8: Pipeline diagram part 2

PC	Instruction	Cycle									
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		
0x3c	addi a2,a2,4					IF	ID	EX	MEM	WB	
0x40	bne a5,a7,24						IF	ID	EX	MEM	WB
0x44	ret							IF	ID		
0x48	li a0, -1								IF		

Table 9: Pipeline diagram part 3

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

Table 10: Pipeline diagram part 4

PC	Instruction	Cycle							
0x38	addi a1,a1,4	IF	ID	EX	MEM	WB			
0x3c	addi a2,a2,4		IF	ID	EX	MEM	WB		
0x40	bne a5,a7,24			IF	ID	EX	MEM	WB	
0x44	ret				IF	ID	EX	MEM	WB
0x48	li a0, -1					IF	ID		
0x4c	ret						IF		

Table 11: Pipeline diagram part 5

Here, we consider that it is important to calculate the function's CPI. For simplicity, we'll consider the entire program as wrote above and its cycles represented in the pipeline diagram. Since we have 30 instructions and 34 clock cycles, we can find the result directly as follow:

$$CPI = \frac{N_c}{N_i} = \frac{34}{30} \approx 1.13$$

Where,

- N_c is the number of clock cycles that are executed in our function;
- N_i is the number of instructions we have in our function.

There are some reasons, in the hardware level, that explain the value obtained for the CPI. For example, it could be due to the fact that in the given architecture there are separated memories for data and instructions, since it is based on the Harvard Machine architecture. With that architecture, differently from the von Neumann, there's no bottleneck problem, because of the relative ability of processors compared to top rates of data transfer. In the von Neumann Machine, the processor is idle while memory is accessed, because physically we have only one memory.

4 Processor Design

Aims: Explain and understand the instruction set of a processor and its implementation using a simple pipeline.

4.1 Instruction Set Architecture

In this section we will define an instruction set and the binary representation of the instructions of a simple processor, considering the following characteristics: all instructions are encoded in 16 bits; the processor has 16 registers, where each register is 32-bit wide; the values of the immediate operands can be sign-extended for some instruction formats; conditional branches, unconditional jumps, and calls in our instruction set architecture have a branch delay slot for a single instruction. Below we describe the characteristics in more detail as well as others features of our processor based on the Table 12.

Firstly, we started by the `bnz` instruction, in other words, the definition of our only *SB-type* instruction. For that part, we have defined our immediate as being 10-bit wide, since we have a 2-bit wide opcode and it was needed 4 bits to encode the `rs1`. Therefore, the opcode 11_2 is done.

Next, for the arithmetic instructions, or *R-type* instructions, we have considered the opcode 01_2. It was needed 12 bits to encode the three registers: the `rd` (destination register), the `rs1` (source register one) and the `rs2` (source register two). We are left, then, with only 2 bits to work with, so we decided to create a `funct2` section that can only assume 3 different values, because the value 00_2 is already reserved for the next instruction type (RS).

In the *RS-type* instructions, instead of having 3 registers, we use only `rd` and `rs1`, so it was possible to set up a `funct4` field in order to have 16 possibilities of operations. Here the first register is the destination, but also a source. The opcode here is still 01_2 and the previous `funct2` section will always be 00_2 (the reason why there are only 3 R-type instructions).

For the memory accesses instructions (store and load, or S and L types, respectively) the encoding method is almost the same, except by the opcode, which, for these instructions, is a 3-bit wide field, in order to distinguish them from the arithmetic and conditional branch instructions. Both store and load have 5 bits for immediate, 2 register fields, and the 2 least significant bits of the opcode are 10_2.

After that, there is the immediate instructions, or *I-type* ones. For that type, we are using the opcode 00_2. Here we have 4 bits for the `rd`, which is the destination and source register, 2 bits (`funct2` field) to decide which function use, and here again we never use the combination 00_2 for this field since it's reserved for the jump instructions (J and JI). Two of our I-type instructions (`slli` and `li`) aren't sign-extended, while `addi` is. Therefore, the former ones' immediate ranges from 0 to 255 and the latter's goes from -128 to 127. The reason is explained in Section 4.1.1.

Finally, concerning the jump instructions, the first is the jump immediate, or *JI-type*, where we have a 11-bit wide immediate, which is sign-extended, and the other bits are used for the identification of the instruction. The last one is the *J-type* instruction, where

the `rs1` register must have an address which is a multiple of 2, since we are working with instructions that are encoded in 16 bits.

It is important to emphasize that, with exception to the `slli` and `li` instructions, in all the instructions that contains an `immediate`, this one is treated as a two's complement number and is sign-extended.

15 12	11 8	7 6 5 4	3 2	1	0	
imm[9:6]	rs1	imm[5:0]		1	1	**SB**
rd	rs1	rs2	funct2	0	1	**R**
rd	rs1	funct4	0 0	0	1	**RS**
imm[4:1]	rs1	rs2	imm[0] 0	1	0	**S**
rd	rs1	imm	1	1	0	**L**
rd	imm		funct2	0	0	**I**
immediate		1	0 0	0	0	**JI**
-	rs1	- 1 0	0 0	0	0	**J**

Table 12: Core Instructions Formats for our architecture

Table 13 lists the assembler mnemonics for the 16 registers we have in our architecture, as well as their description.

Register	**Name**	**Description**	**Saver**
x0	zero	Zero source register	N.C.
x1-x4	a0-a3	Argument register, return value	caller
x5-x10	s0-s5	Register preserved by subprograms	callee
x11-x12	t0-t1	Temporary registers	caller
x13	s0/fp	Saved register 0 or frame pointer	callee
x14	sp	Stack pointer	callee
x15	lr	Link register	caller

Table 13: Assembler mnemonics for our architecture registers

4.1.1 Exercises

- **Define how the 16 registers have to be used by the programmer. In particular define how arguments are passed on function calls for functions with up to 4 arguments. Define how a to return from a function call and how the returned result of the function call can be retrieved. Which registers are preserved/or potentially modified during a function call.**

  Answer:

  As mentioned before, Table 13 shows our definition of how the 16 registers have to be used by the programmer. Firstly, the register `x0`, or `zero`, has always a value of 0 inside it. Next, we have defined the argument registers as being `x1` to `x4`, or `a0` to `a3`. It can be observed that they are used in a similar way as in RISC-V architecture, where the arguments are passed on function calls for functions with up

to 4 arguments by using those registers in order and the return value will be in the same way (either in `a0`, or `a1` and so on). The state of the data into the registers before calling a function is saved on the stack by using the instruction `push` and recovered in the end of the function before returning the control to the caller. We decided to use a stack full descendent, so the `push` and `pop` must always be called in a inverted way – for example, the first pushed register in the caller should be the last popped register in the callee. Then, we have the registers `x5` to `x10` that are preserved by subprograms, the temporary registers `x11` to `x12` and, finally, we have the saved register `s0` or frame pointer (`x13`), the stack pointer (`x14`) and the link register (`x15`).

- **Describe each instruction of your processor. Explain what the instruction is doing, how it can be written in human readable form (assembly), and how it is encoded in binary form.**

  Answer:

  Table 14 presents the assembly and binary formats of the instructions of the processor. In the last column, it is indicated the type of each instruction, and the binary format of each type was detailed before, in Table 12.

funct4	**func2**	**opcode**	**Mnemonical name**	**Assemby usage**	**Type**
-	-	11	bnz	bnz rs1, <label or offset>	SB
-	00	01	add	add rd, rs1, rs2	R
-	01	01	sub	sub rd, rs1, rs2	
-	10	01	and	and rd, rs1, rs2	
0x0	00	01	sll	sll rd, rs1	RS
0x1	00	01	srl	srl rd, rs1	
0x2	00	01	sra	sra rd, rs1	
0x3	00	01	orr	orr rd, rs1	
0x4	00	01	xor	xor rd, rs1	
0x5	00	01	not	not rd, rs1	
0x6	00	01	neg	neg rd, rs1	
0x7	00	01	sez	sez rd, rs1	
0x8	00	01	slz	slz rd, rs1	
0x9	00	01	sgz	sgz rd, rs1	
0xa	00	01	sgez	sgez rd, rs1	
0xb	00	01	slez	slez rd, rs1	
-	-	010	sw	sw rs2, imm(rs1)	S
-	-	110	lw	lw rd, imm(rs1)	L
-	01	00	addi	addi rd, rs1, <imm>	I
-	10	00	slli	slli rd, rs1, <imm>	
-	11	00	li	li rd, <imm>	
-	-	10000	call	call <label or offset>	JI
-	-	100000	jr	jr rs1	J

Table 14: Instruction codes

In the following, it is presented a description of each instruction of the processor.

- `bnz` ("Branch if not zero"): The value of the register `rs1` is verified, and if it is different from 0, then a branch occurs. In this case, the target address (i.e. the next value of `PC`) is equal to `PC` + offset, where the offset is equal to 2 × `immediate`. If a label is informed, the target corresponds to it. If the branch is not taken, the next value of `PC` is simply `PC` + 2.
- `add`: The values of the registers `rs1` and `rs2` are added and the result is stored in `rd`.
- `sub`: The value of the register `rs2` is subtracted from the value of `rs1`, and the result is stored in `rd`.
- `and`: The bitwise logical AND operation between the values of the registers `rs1` and `rs2` is calculated, and the result is stored in `rd`.
- `sll` ("Shift Left Logical"): The value stored in `rd` is shifted to the left by a number of times equal to the value stored in the least significant byte of `rs1`, considering it as unsigned. The result is overwritten in `rd`. The new bits introduced from the right side are always 0.
- `srl` ("Shift Right Logical"): The value stored in `rd` is (logically) shifted to the right by a number of times equal to the value stored in the least significant byte of `rs1`, considering it as unsigned. The result is overwritten in `rd`. Since it is a logical shift, the new bits introduced from the left side are always 0.
- `sra` ("Shift Right Arithmetic"): The value stored in `rd` is arithmetically shifted to the right by a number of times equal to the value stored in the least significant byte of `rs1`, considering it as unsigned. The result is overwritten in `rd`. Since it is an arithmetical shift, the new bits introduced from the left side are always equal to the most significant bit of the initial value of `rd`, in order to keep the sign.
- `orr`: The bitwise logical OR operation between the values of the registers `rs1` and `rd` is computed, and the result is overwritten in `rd`.
- `xor`: The bitwise logical XOR operation between the values of the registers `rs1` and `rd` is computed, and the result is overwritten in `rd`.
- `not`: The bitwise logical NOT operation of the content of `rs1` is computed and the result is stored in `rd`.
- `neg`: The two's complement of `rs1` is computed, and the result is stored in `rd`.
- `sez` ("Set if equal to zero"): Write 0x1 to `rd` if `rs1` is equal to 0; otherwise, write 0x0 to `rd`.
- `slz` ("Set if less than zero"): Write 0x1 to `rd` if `rs1` is less than 0; otherwise, write 0x0 to `rd`.
- `sgz` ("Set if greater than zero"): Write 0x1 to `rd` if `rs1` is greater than 0; otherwise, write 0x0 to `rd`.

– `sgez` ("Set if greater than or equal to zero"): Write 0x1 to `rd` if `rs1` is greater than or equal to 0; otherwise, write 0x0 to `rd`.

– `slez` ("Set if less than or equal to zero"): Write 0x1 to `rd` if `rs1` is less than or equal to 0; otherwise, write 0x0 to `rd`.

– `sw` ("Store Word"): The value of `rs2` is stored in the memory, in the address equal to the sum of the `immediate` and the value of `rs2`.

– `lw` ("Load Word"): The register `rd` receives the content from the memory position with address equal to the sum of the `immediate` and the value of `rs1`.

– `addi` ("Add immediate"): The sum of `rs1` and the `immediate` is stored in `rd`.

– `slli` ("Shift Left Logical Immediate"): The value in `rs1` is shifted to the left by a number of times equal to the value of the `immediate`, considering it as unsigned. The result is stored in `rd`. The new bits introduced from the right side are always 0. In this case, the `immediate` value is unsigned (and, therefore, not sign-extended), since we can only shift a positive number of times.

– `li` ("Load immediate"): The value of the `immediate` is stored in the least significant byte of `rd`. Since the `immediate` field has 8 bits, its value is not sign-extended, because we are interested in using it to change only the least significant byte of `rd`.

– `call`: The next value of `PC` is equal to `PC` + offset, where the offset is equal to $2 \times$ `immediate`. The offset is either directly passed as argument, or obtained from the label, if one is indicated. The return address (equal to the original `PC` + 2) is stored in the `lr` register.

– `jr` ("Jump Register"): The next value of `PC` is equal to the content of `rs1` (that is, the register `rs1` stores the address of the target instruction of the branch). There is no storage of a return address.

Some pseudo-instructions were also defined: `ret` ("return"), `mv` ("move"), `nop` ("no operation"), `push` and `pop`. Table 15 shows the equivalent instructions that implement the desired behaviour.

Pseudo-instruction	**Equivalent instruction(s)**
`ret`	`jr lr`
`mv rd, rs1`	`add rd, rs1, zero`
`nop`	`addi zero,0`
`push rs`	`sw rs, -4(sp)` `addi sp, -4`
`pop rd`	`lw rd, 0(sp)` `addi sp, 4`

Table 15: Description of the pseudo-instruction

- **Group instructions into binary formats, similar to the I-, R-, . . . , and SB-format discussed for RISC-V in the lecture. Illustrate the formats using figures in your report.**

  Answer:

  As explained in the begin of the section 4.1, Table 12 describes our instructions into binary format. There are the following types: *SB* for branch, *R* for register operation, *RS* for short register operation, *S* for store, *L* for load, *I* for immediate, *JI* for jump-immediate and *J* for jump.

- **Define a no-operation instruction (similar to the nop instruction of RISC-V using one of the above instructions. This instruction should be a pseudo instruction that does not modify any registers.**

  Answer:

  The `nop` instruction is defined as being equal to the instruction `add zero,zero,0`. This corresponds to adding 0 to the hard-wired `zero` register (`x0`), which has no effect.

- **Provide the assembly code of a function that takes two arguments and returns the sum of those arguments. In addition, provide the code of a function which does not take any arguments and calls your previously defined function in order to compute the sum of 65408 and 134. Try to make good use of the branch delay slot and recall to save the return address!**

  Answer:

  In order to provide the answer for what was asked, we have decided to put in the same block of code the assembly code of the two required functions, because the second one calls the first one.

  Therefore, we started by writing the function called *sum* that takes two arguments (using the register `a0` and `a1`) and returns the sum of those arguments (`add` function). Here it is important to mention the use of branch delay slot which was done by placing the operation instruction after the `jr` instruction (which returns control to the caller).

  Next, we wrote the *main* which doesn't take any arguments (we clean the argument register in that case, using a `mv` instruction that places zero inside `a0` and `a1`). After, We want to calculate the sum of 65408 and 134. Starting by the simplest part, since 134 is bigger than 127 we used the unsigned version of the load immediate instruction (which takes values from 0 until 255). For the second value, we needed to make an expansion of the load instruction, since it cannot load values above 255, in order to put the value 0x0000ff80 (65408) inside `a1` (because all register are is 32-bit wide and the `li` loads a immediate in the least significant byte). So we first put 0xff in that register, after we shift it by 8 using `slli` and then we call the sum function and finally we load the 0x80 into `a1` (swapped with `call` to exploit delay slot).

```
1  .text
2
3  main:
4      //clean a0 register, expand to add a0,zero,zero
5      mv a0, zero
6      //since 134 is bigger than 127 we must use the unsigned version
7      li a0, 134
8
9      mv a1, zero // add a1,zero,zero
10     //expansion of li a1, 65408 // 0x0000ff80
11     li a1, 0xff
12     slli a1,8
13
14     //li a1, 0x80 // using the delay slot
15     call sum
16     li a1, 0x80
17
18 sum:
19    //add a0,a0,a1 using the delay slot
20    jr lr
21    add a0,a0,a1
```

- **Translate the C-code from Question 1 to corresponding instructions of your processor. Arguments and the return value of the function are communicated through registers. The return address is likewise stored in a register. Your code should respect the register usage conventions that you have defined in the previous exercise from above. This may also require saving/register register values on the stack – depending on your register usage convention. Try to use the instructions of your processor as good as possible in order to minimize the number of instructions.**

  Answer:

  Considering the final version (short version) of the C-code from section 2, we have translated it to our instruction set architecture. Here the register usage convention is applied as described in Table 13, which includes the arguments, the return address from subroutines, and so on. The obtained code can be find in the following block of code:

```
1  .text
2      sez t1,a0
3      bnz t1, arg_0
4      sez t1,a1
5      bnz t1, arg_0
6      sez t1,a2
7      bnz t1, arg_0
```

```
 8      slez t1,a3
 9      bnz t1, bad_a3
10
11      push s0 //expands to:
12              //sw s0, -4(sp)
13              //addi sp, -4
14
15      mv s0, a3 // add s0, a3, zero
16  for_loop: // count to 0
17
18      lw t0,0(a0)
19      lw t1,0(a1)
20      add t0, t1
21      sw t0, 0(a2)
22
23      addi s0, -1
24      addi a0, 4
25      addi a1, 4
26      //addi a2, 4
27      bnz s0, for_loop
28      addi a2, 4
29
30      pop s0  //expands to:
31              //lw s0, 0(sp)
32              //addi sp, 4
33      ret //jr lr
34      mv a0, a3
35
36  arg_0:
37      //li a0, -1 //swapped with ret to exploit delay slot
38      ret
39      li a0, -1
40
41  bad_a3:
42      ret
43      mv a0,a3
```

4.2 Pipelining

In this section we will define the pipeline of our processor, while respecting the following characteristics:

- Pipeline stages: instruction fetch (IF), instruction decode (ID), and execute (EX).

- For arithmetic the three pipeline stages correspond to the pipeline stages of the RISC-V processor.
- The address computation and the memory access are both performed in the EX stage.
- Conditional branches, unconditional jumps, and calls are executed in the ID stage.
- The processor registers are written at the beginning of the EX stage and read at the end of the ID stage.

4.2.1 Exercises

- **Draw a diagram of your processor's design. Use registers, pipeline registers, multiplexers, ALUs, . . . , as you need them. Describe relevant parts of the diagram. The diagram should contain everything that is necessary to execute all instructions that you have defined!**

  Figure 2 shows the diagram of our processor's design.

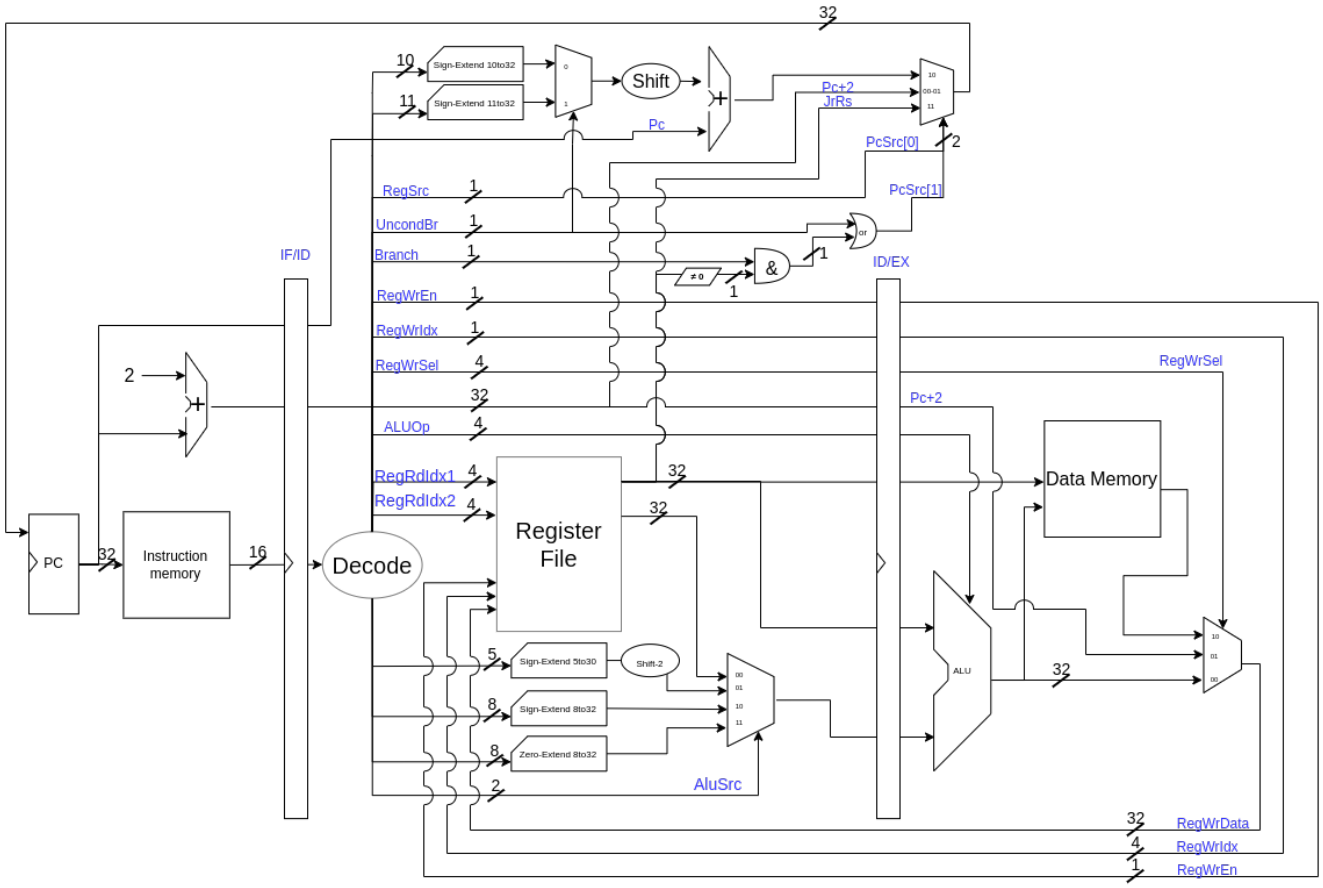

Figure 2: Diagram of your processor's design

For that part, the most complex task was the branch evaluation; for it three signals are used:

- **Branch** activates whenever a branch instruction (either conditional or unconditional) is decoded;
- **UncondBr** activates when a unconditional instruction(jr, call) is decoded and it is used to ignore the result of the !=0 block and to select as source of the PC offset the 11 bit immediate;
- **RegSrc** is activated only when a `jr` is decoded, select the register file as source of next PC.

- **Make a copy of your drawing that specifically highlights how a call instruction is executed by your pipeline design. Explain what happens in each pipeline stage. Notably, explain which control signals are used to control multiplexers, read/write registers, the ALU, et cetera. Also explain when and how these control signals are computed.**

  Figure 3 shows a version of the previous diagram adapted to the execution of the `call` instruction in our pipeline design.

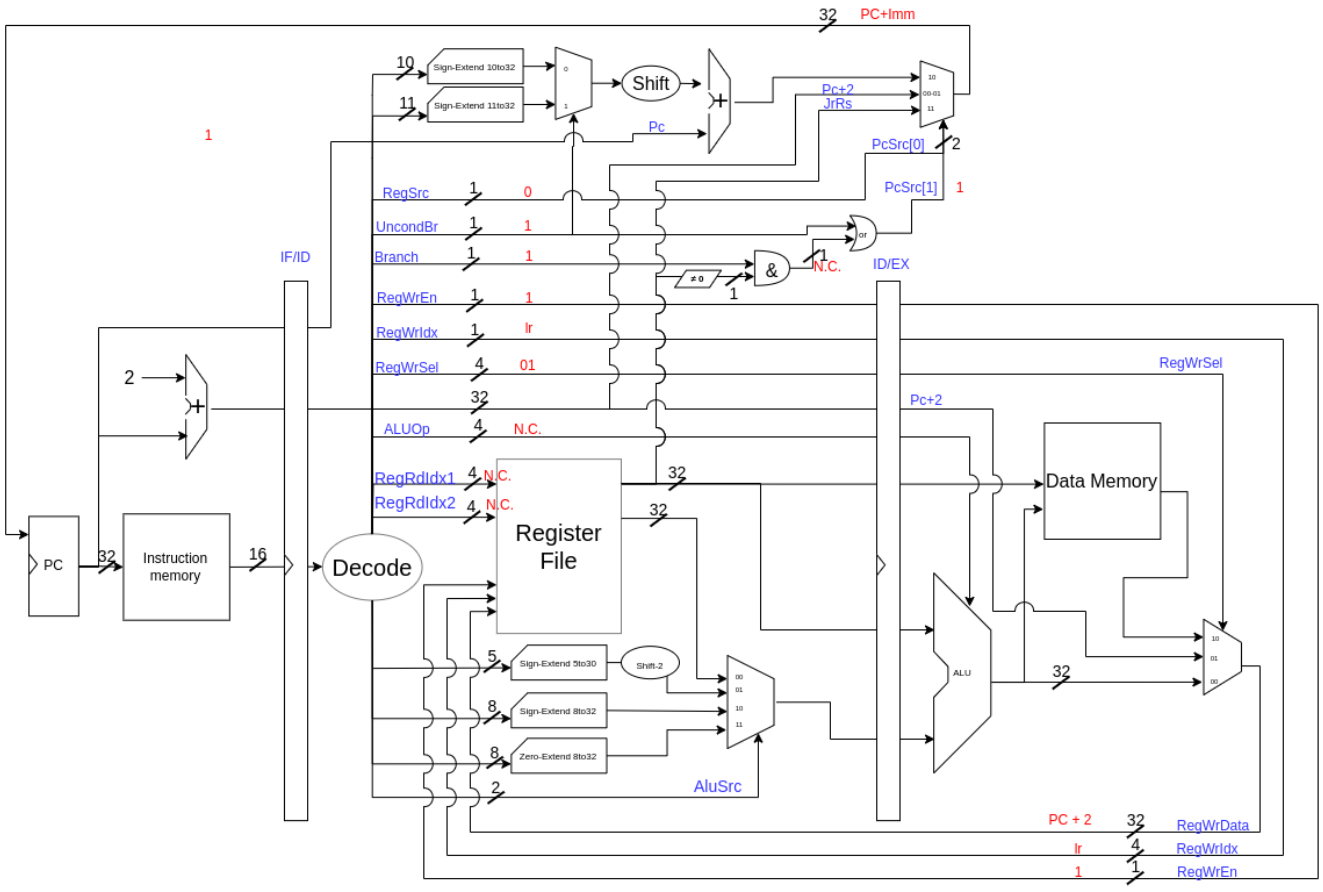

Figure 3: Call instruction pipeline design

The control signal for executing a call have been explained in last point; to them it have to be added RegWrEn (1), RegWrIdx(identifying `lr` the link register) and RegWrSel(01), selecting PC+2 as source for write to the register file. Regarding the stage:

- **IF stage**: the instruction memory is read at the address contained in the PC;
- **ID stage**: the 16 bit instruction is decoded, the 11 bit immediate is sign-extended and added to the PC, then it is chosen as source of the next PC. At the clock the value is saved to the PC, which means the instruction currently in IF continues its execution and the new one will be at the new PC. The result of the and block is ignored as in OR with **UncondBr**. The important signals are those in red;
- **EX stage**: former PC + 2 is saved to register file, inside `lr`.

- **Which kinds of hazards (data, control, or structural) can you encounter for your processor? Explain under which circumstances these hazards occur. How are these hazards resolved?**

  Answer:

  Our processor does not have *control hazard* since we are using branch delay slot, so the architecture has as feature the fact that the instruction after a branch is executed. In the case in which we don't want the instruction after a branch to be executed, we must put a `nop` on it. Briefly, is not a *control hazard* in the sense of the next PC is not yet available when needed.

  Concerning *structural hazard*, we don't have this possibility because different resources are used in different stages so the instructions cannot use any resource at the same time.

  Finally, about *data hazard*, Figure 4 illustrates the fact that we don't have data hazards since the processor registers are written at the beginning of the EX stage (blue arrow) and read at the end of the ID stage (green arrow), so even if we have 2 instructions in sequence and the first write something in a register and the second one needs this value in the same register, the data will always be available when needed. Also if we follow the data path of an instruction, the only flip flop we encounter is the ID/EX register, like in the initial serial risc-v architecture saw at lesson.

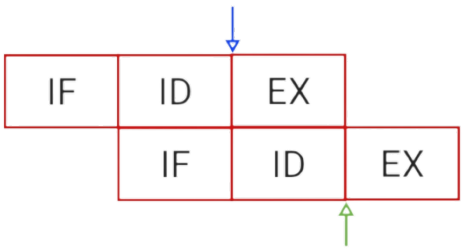

Figure 4: Writing/Reading processor registers

- **Does your processor need logic to flush instructions from the pipeline (as discussed in the lecture)? Explain why this logic is needed or why it is not needed.**

  Answer:

  In our processor, there is no need to flush because we make use of the branch delay slot.

References

[1] M Tim Jones. “Optimization in GCC”. In: *Linux journal* 2005.131 (2005), p. 11.

[2] Andrew Waterman et al. “The RISC-V instruction set manual”. In: *Volume I: User-Level ISA', version* 2 (2014).

A Makefile

```
PREFIX = riscv64-unknown-elf-
CC = $(PREFIX)gcc
AS = $(PREFIX)as
OBJDUMP = $(PREFIX)objdump

WARNINGS = -Wall
CFLAGS = -g -O0 -mcmodel=medlow -mabi=ilp32 -march=rv32im $(WARNINGS) -c

OBJ1 = se201-prog.o
SOURCE_C1 = se201-prog.c

OBJ2 = se201-prog-short.o
SOURCE_C2 = se201-prog-short.c

OBJ = program1.o
SOURCE = $(wildcard *.s)

CLEAN = $(OBJ) $(OBJ1) $(OBJ2)

.PHONY: clean

all: $(OBJ)

$(OBJ):$(SOURCE)
$(AS) $^ -o $@

objdump:$(OBJ)
 $(OBJDUMP) -d $^

$(OBJ1):$(SOURCE_C1)
 $(CC) $(CFLAGS) $^ -o $@

objdump1:$(OBJ1)
 $(OBJDUMP) -d $^

$(OBJ2):$(SOURCE_C2)
 $(CC) $(CFLAGS) $^ -o $@

objdump2:$(OBJ2)
 $(OBJDUMP) -d $^

clean:
```

```
rm -f $(CLEAN)

info:
@echo "\t\t----COMMAND LIST-----"
@echo "\n\tmake program1.o: generates the program1.o file"
@echo "\n\tmake objdump: shows the assembly related to the file program1.o"
@echo "\n\tmake se201-prog.o: generates the se201-prog.o file"
@echo "\n\tmake objdump1: shows the assembly related to the file se201-prog.o"
@echo "\n\tmake se201-prog-short.o: generates the se201-prog-short.o file"
@echo "\n\tmake objdump2: shows the assembly related to the file se201-prog-short.o"
@echo "\n\tmake clean: cleans up the created object files"
@echo "\n\tmake info: shows this command list"
```

B Assembly code with -O0 option (Long version)

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


Disassembly of section .text:

00000000 <func>:
   0:   fd010113                addi    sp,sp,-48
   4:   02812623                sw      s0,44(sp)
   8:   03010413                addi    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:   fef42423                sw      a5,-24(s0)
  24:   fd042783                lw      a5,-48(s0)
  28:   fcf42e23                sw      a5,-36(s0)
  2c:   fe842783                lw      a5,-24(s0)
  30:   0c078063                beqz    a5,f0 <.L2>
  34:   fd842783                lw      a5,-40(s0)
  38:   0a078863                beqz    a5,e8 <.L3>
  3c:   fd442783                lw      a5,-44(s0)
  40:   0a078463                beqz    a5,e8 <.L3>
  44:   fd042783                lw      a5,-48(s0)
  48:   08f05c63                blez    a5,e0 <.L4>
  4c:   fe842783                lw      a5,-24(s0)
  50:   fef42623                sw      a5,-20(s0)
  54:   fd042783                lw      a5,-48(s0)
  58:   00279793                slli    a5,a5,0x2
  5c:   fef42223                sw      a5,-28(s0)
  60:   fe842703                lw      a4,-24(s0)
  64:   fe442783                lw      a5,-28(s0)
  68:   00f707b3                add     a5,a4,a5
  6c:   fef42423                sw      a5,-24(s0)
  70:   05c0006f                j       cc <.L5>

00000074 <.L6>:
  74:   fec42783                lw      a5,-20(s0)
  78:   0007a783                lw      a5,0(a5)
  7c:   fef42223                sw      a5,-28(s0)
  80:   fd842783                lw      a5,-40(s0)
  84:   0007a783                lw      a5,0(a5)
```

```
  88:   fef42023                sw      a5,-32(s0)
  8c:   fe442703                lw      a4,-28(s0)
  90:   fe042783                lw      a5,-32(s0)
  94:   00f707b3                add     a5,a4,a5
  98:   fef42223                sw      a5,-28(s0)
  9c:   fd442783                lw      a5,-44(s0)
  a0:   fe442703                lw      a4,-28(s0)
  a4:   00e7a023                sw      a4,0(a5)
  a8:   fec42783                lw      a5,-20(s0)
  ac:   00478793                addi    a5,a5,4
  b0:   fef42623                sw      a5,-20(s0)
  b4:   fd842783                lw      a5,-40(s0)
  b8:   00478793                addi    a5,a5,4
  bc:   fcf42c23                sw      a5,-40(s0)
  c0:   fd442783                lw      a5,-44(s0)
  c4:   00478793                addi    a5,a5,4
  c8:   fcf42a23                sw      a5,-44(s0)

000000cc <.L5>:
  cc:   fec42703                lw      a4,-20(s0)
  d0:   fe842783                lw      a5,-24(s0)
  d4:   faf710e3                bne     a4,a5,74 <.L6>
  d8:   fdc42783                lw      a5,-36(s0)
  dc:   0180006f                j       f4 <.L7>

000000e0 <.L4>:
  e0:   fdc42783                lw      a5,-36(s0)
  e4:   0100006f                j       f4 <.L7>

000000e8 <.L3>:
  e8:   fff00793                li      a5,-1
  ec:   0080006f                j       f4 <.L7>

000000f0 <.L2>:
  f0:   fff00793                li      a5,-1

000000f4 <.L7>:
  f4:   00078513                mv      a0,a5
  f8:   02c12403                lw      s0,44(sp)
  fc:   03010113                addi    sp,sp,48
 100:   00008067                ret
```

C Assembly code with -O option (Long version)

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


Disassembly of section .text:

00000000 <func>:
   0:   00050713                mv      a4,a0
   4:   00068513                mv      a0,a3

00000008 <.LVL1>:
   8:   04070663                beqz    a4,54 <.L4>
   c:   04058863                beqz    a1,5c <.L5>
  10:   04060a63                beqz    a2,64 <.L6>
  14:   04d05a63                blez    a3,68 <.L2>

00000018 <.LVL2>:
  18:   00269313                slli    t1,a3,0x2

0000001c <.LVL3>:
  1c:   00e30333                add     t1,t1,a4

00000020 <.LVL4>:
  20:   04670463                beq     a4,t1,68 <.L2>
  24:   00070793                mv      a5,a4

00000028 <.LBB2>:
  28:   40e60633                sub     a2,a2,a4

0000002c <.LVL5>:
  2c:   40e585b3                sub     a1,a1,a4

00000030 <.L3>:
  30:   0007a883                lw      a7,0(a5)
  34:   00f60833                add     a6,a2,a5
  38:   00f58733                add     a4,a1,a5
  3c:   00072703                lw      a4,0(a4)
  40:   01170733                add     a4,a4,a7
  44:   00e82023                sw      a4,0(a6)
  48:   00478793                addi    a5,a5,4

0000004c <.LBE2>:
  4c:   fef312e3                bne     t1,a5,30 <.L3>
```

```
  50:   00008067                ret

00000054 <.L4>:
  54:   fff00513                li      a0,-1
  58:   00008067                ret

0000005c <.L5>:
  5c:   fff00513                li      a0,-1

00000060 <.LVL11>:
  60:   00008067                ret

00000064 <.L6>:
  64:   fff00513                li      a0,-1

00000068 <.L2>:
  68:   00008067                ret
```

D Assembly code with -O3 option (Long version)

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


Disassembly of section .text:

00000000 <func>:
   0:   00050793                mv      a5,a0

00000004 <.LVL1>:
   4:   04050663                beqz    a0,50 <.L8>
   8:   04058463                beqz    a1,50 <.L8>
   c:   04060263                beqz    a2,50 <.L8>
  10:   02d05c63                blez    a3,48 <.L4>

00000014 <.LVL2>:
  14:   00269513                slli    a0,a3,0x2

00000018 <.LVL3>:
  18:   00f50533                add     a0,a0,a5

0000001c <.LVL4>:
  1c:   02a78663                beq     a5,a0,48 <.L4>
  20:   40f60633                sub     a2,a2,a5

00000024 <.LVL5>:
  24:   40f585b3                sub     a1,a1,a5

00000028 <.L5>:
  28:   00f58733                add     a4,a1,a5
  2c:   0007a883                lw      a7,0(a5)

00000030 <.LVL7>:
  30:   00072703                lw      a4,0(a4)
  34:   00f60833                add     a6,a2,a5
  38:   00478793                addi    a5,a5,4

0000003c <.LVL8>:
  3c:   01170733                add     a4,a4,a7
  40:   00e82023                sw      a4,0(a6)

00000044 <.LBE2>:
  44:   fef512e3                bne     a0,a5,28 <.L5>
```

```
00000048 <.L4>:
  48:   00068513                mv      a0,a3
  4c:   00008067                ret

00000050 <.L8>:
  50:   fff00513                li      a0,-1

00000054 <.LVL12>:
  54:   00008067                ret
```

E Assembly code with -O0 option (Short version)

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


Disassembly of section .text:

00000000 <func>:
   0:   fd010113                addi    sp,sp,-48
   4:   02812623                sw      s0,44(sp)
   8:   03010413                addi    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:   0700006f                j       a8 <.L4>

0000003c <.L3>:
  3c:   fd042783                lw      a5,-48(s0)
  40:   06f05263                blez    a5,a4 <.L5>

00000044 <.LBB2>:
  44:   fe042623                sw      zero,-20(s0)
  48:   0500006f                j       98 <.L6>

0000004c <.L7>:
  4c:   fec42783                lw      a5,-20(s0)
  50:   00279793                slli    a5,a5,0x2
  54:   fdc42703                lw      a4,-36(s0)
  58:   00f707b3                add     a5,a4,a5
  5c:   0007a683                lw      a3,0(a5)
  60:   fec42783                lw      a5,-20(s0)
  64:   00279793                slli    a5,a5,0x2
  68:   fd842703                lw      a4,-40(s0)
  6c:   00f707b3                add     a5,a4,a5
```

```
  70:   0007a703                lw      a4,0(a5)
  74:   fec42783                lw      a5,-20(s0)
  78:   00279793                slli    a5,a5,0x2
  7c:   fd442603                lw      a2,-44(s0)
  80:   00f607b3                add     a5,a2,a5
  84:   00e68733                add     a4,a3,a4
  88:   00e7a023                sw      a4,0(a5)
  8c:   fec42783                lw      a5,-20(s0)
  90:   00178793                addi    a5,a5,1
  94:   fef42623                sw      a5,-20(s0)

00000098 <.L6>:
  98:   fec42703                lw      a4,-20(s0)
  9c:   fd042783                lw      a5,-48(s0)
  a0:   faf746e3                blt     a4,a5,4c <.L7>

000000a4 <.L5>:
  a4:   fd042783                lw      a5,-48(s0)

000000a8 <.L4>:
  a8:   00078513                mv      a0,a5
  ac:   02c12403                lw      s0,44(sp)
  b0:   03010113                addi    sp,sp,48
  b4:   00008067                ret
```

F Assembly code with -O1 option (Short version)

```
se201-prog-short.o:     file format elf32-littleriscv
Disassembly of section .text:

00000000 <func>:
   0:   00050813                mv      a6,a0
   4:   00068513                mv      a0,a3

00000008 <.LVL1>:
   8:   04080063                beqz    a6,48 <.L4>
   c:   04058263                beqz    a1,50 <.L5>
  10:   04060463                beqz    a2,58 <.L6>
  14:   04d05463                blez    a3,5c <.L2>
  18:   00080793                mv      a5,a6
  1c:   00269713                slli    a4,a3,0x2
  20:   00e80833                add     a6,a6,a4

00000024 <.L3>:
  24:   0007a703                lw      a4,0(a5)
  28:   0005a883                lw      a7,0(a1)
  2c:   01170733                add     a4,a4,a7
  30:   00e62023                sw      a4,0(a2)
  34:   00478793                addi    a5,a5,4
  38:   00458593                addi    a1,a1,4
  3c:   00460613                addi    a2,a2,4
  40:   ff0792e3                bne     a5,a6,24 <.L3>
  44:   00008067                ret

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

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

00000054 <.LVL5>:
  54:   00008067                ret

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

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

G Assembly code with -O option (Short version)

```
se201-prog-short.o:     file format elf32-littleriscv
Disassembly of section .text:

00000000 <func>:
   0:   00050813                mv      a6,a0
   4:   00068513                mv      a0,a3

00000008 <.LVL1>:
   8:   04080063                beqz    a6,48 <.L4>
   c:   04058263                beqz    a1,50 <.L5>
  10:   04060463                beqz    a2,58 <.L6>
  14:   04d05463                blez    a3,5c <.L2>
  18:   00080793                mv      a5,a6
  1c:   00269713                slli    a4,a3,0x2
  20:   00e80833                add     a6,a6,a4

00000024 <.L3>:
  24:   0007a703                lw      a4,0(a5)
  28:   0005a883                lw      a7,0(a1)
  2c:   01170733                add     a4,a4,a7
  30:   00e62023                sw      a4,0(a2)
  34:   00478793                addi    a5,a5,4
  38:   00458593                addi    a1,a1,4
  3c:   00460613                addi    a2,a2,4
  40:   ff0792e3                bne     a5,a6,24 <.L3>
  44:   00008067                ret

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

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

00000054 <.LVL5>:
  54:   00008067                ret

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

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

H Assembly code with -O3 option (Short version)

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


Disassembly of section .text:

00000000 <func>:
   0:   00050793                mv      a5,a0
   4:   04050063                beqz    a0,44 <.L8>
   8:   02058e63                beqz    a1,44 <.L8>
   c:   02060c63                beqz    a2,44 <.L8>
  10:   00269893                slli    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                addi    a5,a5,4
  28:   00458593                addi    a1,a1,4
  2c:   01070733                add     a4,a4,a6
  30:   00e62023                sw      a4,0(a2)
  34:   00460613                addi    a2,a2,4
  38:   ff1792e3                bne     a5,a7,1c <.L4>

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

00000040 <.LVL2>:
  40:   00008067                ret

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

00000048 <.LVL4>:
  48:   00008067                ret
```