Project 2: RISC-V Pipeline

Gabriel Gaspar
Alejandro Guijarro Monerris
Raynner Schnneider Carvalho

TELECOM Paris, Palaiseau

February 9th 2024

Contents

List of Figures

Chapter 1

RISC-V Tool Chain

I Get familiar with the RISC-V tools and the Ripes interface.

A RISC-V Tools

Compilation of *insertion-sort.c* using *make all* executes the following command
riscv64-unknown-elf-gcc -O -g -fno-pic -fno-inline -mcmodel=medlow -mabi=ilp32 -march=rv32im -Wall -static -nostdinc -nostartfiles -nodefaultlibs -nostdlib -o insertion-sort.elf insertion-sort.c

The option "-nostdlib" does the following [1]: "Do not use the standard system startup files or libraries when linking.". For example, according again to the documentation, "The compiler may generate calls to *memcmp*, *memset*, *memcpy* and *memmove*. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.".

The option "-nostartfiles" [1] does the following: "Do not use the standard system startup files when linking."

The compiled code can be disassembled with the command "riscv64-linux-gnu-objdump -d insertion-sort.elf":

Listing 1.1: Disassemble of insertion-sort.elf

```
insertion-sort.elf:     file format elf32-littleriscv

Disassembly of section .text:

00010094 <minIndex>:
   10094:       00050813        mv      a6,a0
   10098:       04b05063        blez    a1,100d8 <minIndex+0x44>
   1009c:       00050693        mv      a3,a0
   100a0:       00000513        li      a0,0
   100a4:       00000713        li      a4,0
```

```
    100a8:      0100006f        j       100b8 <minIndex+0x24>
    100ac:      00170713        add     a4,a4,1
    100b0:      00468693        add     a3,a3,4
    100b4:      02e58063        beq     a1,a4,100d4 <minIndex+0x40>
    100b8:      00251793        sll     a5,a0,0x2
    100bc:      00f807b3        add     a5,a6,a5
    100c0:      0006a603        lw      a2,0(a3)
    100c4:      0007a783        lw      a5,0(a5)
    100c8:      fef652e3        bge     a2,a5,100ac <minIndex+0x18>
    100cc:      00070513        mv      a0,a4
    100d0:      fddff06f        j       100ac <minIndex+0x18>
    100d4:      00008067        ret
    100d8:      00000513        li      a0,0
    100dc:      00008067        ret

000100e0 <main>:
    100e0:      e5010113        add     sp,sp,-432
    100e4:      1a112623        sw      ra,428(sp)
    100e8:      1a812423        sw      s0,424(sp)
    100ec:      1a912223        sw      s1,420(sp)
    100f0:      1b212023        sw      s2,416(sp)
    100f4:      19312e23        sw      s3,412(sp)
    100f8:      000117b7        lui     a5,0x11
    100fc:      1a478793        add     a5,a5,420 # 111a4 <input>
    10100:      00410413        add     s0,sp,4
    10104:      18c78613        add     a2,a5,396
    10108:      00040713        mv      a4,s0
    1010c:      0007a683        lw      a3,0(a5)
    10110:      00d72023        sw      a3,0(a4)
    10114:      00478793        add     a5,a5,4
    10118:      00470713        add     a4,a4,4
    1011c:      fec798e3        bne     a5,a2,1010c <main+0x2c>
    10120:      00000493        li      s1,0
    10124:      06300993        li      s3,99
    10128:      06200913        li      s2,98
    1012c:      409985b3        sub     a1,s3,s1
    10130:      00040513        mv      a0,s0
    10134:      f61ff0ef        jal     10094 <minIndex>
    10138:      00950533        add     a0,a0,s1
    1013c:      00251513        sll     a0,a0,0x2
    10140:      19050793        add     a5,a0,400
    10144:      00278533        add     a0,a5,sp
    10148:      e7452783        lw      a5,-396(a0)
    1014c:      00042703        lw      a4,0(s0)
    10150:      e6e52a23        sw      a4,-396(a0)
```

```
    10154:      00f42023        sw      a5,0(s0)
    10158:      00148493        add     s1,s1,1
    1015c:      00440413        add     s0,s0,4
    10160:      fd2496e3        bne     s1,s2,1012c <main+0x4c>
    10164:      00412503        lw      a0,4(sp)
    10168:      1ac12083        lw      ra,428(sp)
    1016c:      1a812403        lw      s0,424(sp)
    10170:      1a412483        lw      s1,420(sp)
    10174:      1a012903        lw      s2,416(sp)
    10178:      19c12983        lw      s3,412(sp)
    1017c:      1b010113        add     sp,sp,432
    10180:      00008067        ret

00010184 <_start>:
    10184:      ff010113        add     sp,sp,-16
    10188:      00112623        sw      ra,12(sp)
    1018c:      f55ff0ef        jal     100e0 <main>
    10190:      00a00893        li      a7,10
    10194:      00000073        ecall
    10198:      00c12083        lw      ra,12(sp)
    1019c:      01010113        add     sp,sp,16
    101a0:      00008067        ret
```

Find the assembly code corresponding to the first for-loop in function main. At which address range is the code?

The code corresponding to the first for-loop is the following (with added comments explaining how it works):

Listing 1.2: Assembly of first for loop

```
    for(i = 0; i < SIZE; i++)
100f8:  000117b7                lui     a5,0x11
100fc:  1a478793                add     a5,a5,420 # 111a4 <input
   >
# this loads the address of the array we are reading from (input
   )
10100:  00410413                add     s0,sp,4
10104:  18c78613                add     a2,a5,396
# computes the end address (a5 + SIZE*4) and stores it in a2
    {
10108:  00040713                mv      a4,s0
# loads the stack address to a4
    {
        buf[i] = input[i];
1010c:  0007a683                lw      a3,0(a5)
```

```
10110:   00d72023                    sw      a3,0(a4)
         for(i = 0; i < SIZE; i++)
10114:   00478793                    add     a5,a5,4
10118:   00470713                    add     a4,a4,4
1011c:   fec798e3                    bne     a5,a2,1010c <main+0x2c>
# checks the end of the loop
# if a5 and a2 match then we have looped SIZE times
```

Find the address of symbol *input*

```
tp-3a209-10% riscv64-unknown-elf-objdump -t -j.data -j.text
   insertion-sort.elf

insertion-sort.elf:     file format elf32-littleriscv

SYMBOL TABLE:
00010094 l    d  .text  00000000 .text
000111a4 l    d  .data  00000000 .data
00011334 g       .data  00000000 __SDATA_BEGIN__
000111a4 g     O .data  00000190 input
[...]
```

The symbol *input* corresponds to the array treated as the array sorted by of the main function, that performs the insertion sort. Its address is loaded into a register (in instruction at address 0x0c100fc) and then this register is used to load values from memory (array's values) (in instruction at address 0x1010c). The explanation of this particular instructions have been detailed in the previous section.

Explain why PIC-code is essential to many modern security mechanisms. Which role does the instruction *auipc* play on the RISC-V architecture in the context of PIC-code?

Position Independent Code refers to a piece of code that can be executed regardless of its position in memory. For example, if branches in some part of code are done to a relative address, that means that the branch will execute properly regardless of where the code is in memory. By contrast, if the destination address of a branch is an absolute address, then the code where the branch jumps has to be stored in that address in order for the branch to properly work. PIC is commonly enforced by the compiler. Security wise, PIC, in conjunction with the OS, allows for the code to be executed in unpredictable memory addresses, thus avoiding an attacker to make use of attacks that exploit the fact that the code's execution address is known.

The instruction *auipc* takes as an input a destination register as well as an immediate, it then computes the sum of PC and the immediate and stores it in the destination register. This allows for relative jumps in code, thus letting the possibility to produce PIC, as explained before.

Is the compiled binary code position-independent (PIC)?

The code is PIC. If we check for example the instruction at address 0x10134, we have the instruction *jal x1, -160*. The *jal* instruction takes a register to store the return address of the jump (the call will return to PC+4), and a 20 bit immediate that corresponds to the offset of the jump. The jump will be to address PC+offset, so the jump is PC relative, hence being PIC according to the definition we gave before.

B Ripes

Show that the content of the input array within the simulator matches that of the source/ELF file

According to the Symbol Table shown in the section before, the array `input[]` was saved into the `.data` section, and both share the same initial address, which is 0x000111A4.

The text below shows the content of the array from the source file `.C`, while Figure 1.1 presents a section of the content of the input array within the simulator. Scrolling through the entire memory content from the simulator and comparing it with each value from the source file, it is possible to assure that both content of the input array matches.

```
int input[] = {60, 41, 46, 50, 44, 3, 84, 80, 55, 57, 91, 22, 21, 12, 64, 59,
    71, 34, 81, 77, 69, 95, 2, 24, 61, 73, 25, 19, 29, 91, 45, 53, 39, 15, 47,
    58, 3, 62, 81, 0, 33, 83, 12, 64, 75, 59, 32, 68, 98, 68, 53, 74, 88, 30,
    65, 23, 97, 66, 49, 46, 18, 22, 0, 30, 3, 33, 13, 33, 31, 61, 14, 87, 57,
    95, 20, 92, 67, 71, 42, 52, 18, 98, 2, 93, 95, 69, 90, 8, 97, 46, 26, 68,
    69, 84, 73, 35, 44, 88, 79, 65};
```

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00011208	73	73	0	0	0
0x00011204	61	61	0	0	0
0x00011200	24	24	0	0	0
0x000111fc	2	2	0	0	0
0x000111f8	95	95	0	0	0
0x000111f4	69	69	0	0	0
0x000111f0	77	77	0	0	0
0x000111ec	81	81	0	0	0
0x000111e8	34	34	0	0	0
0x000111e4	71	71	0	0	0
0x000111e0	59	59	0	0	0
0x000111dc	64	64	0	0	0
0x000111d8	12	12	0	0	0
0x000111d4	21	21	0	0	0
0x000111d0	22	22	0	0	0
0x000111cc	91	91	0	0	0
0x000111c8	57	57	0	0	0
0x000111c4	55	55	0	0	0
0x000111c0	80	80	0	0	0
0x000111bc	84	84	0	0	0
0x000111b8	3	3	0	0	0
0x000111b4	44	44	0	0	0
0x000111b0	50	50	0	0	0
0x000111ac	46	46	0	0	0
0x000111a8	41	41	0	0	0
0x000111a4	60	60	0	0	0

Figure 1.1: Memory viewer

Determine the address of the array *buf* by examining the register values

At first, a breakpoint was set in the store instruction related to the first `for-loop` of main, in which copy the values from the `input` array to the `buf`. With the help of the Listing 1.2, the store instruction was easily found.

Examining the register values after using the breakpoint and the simulation step-by-step, the `buf` address was found within the register *a4/x14*. The Figure 1.2 shows it content, which is 0x7FFFFE34.

Figure 1.2: Store instruction and Registers Value

Show that the values of array *buf* get actually sorted

The Figure 1.3 presents the content of `buf` array after running the code. It is clearly sorted with `input` array content as expected.

Address	Word	Byte 0
0x7ffffea8	33	33
0x7ffffea4	32	32
0x7ffffea0	31	31
0x7ffffe9c	30	30
0x7ffffe98	30	30
0x7ffffe94	29	29
0x7ffffe90	26	26
0x7ffffe8c	25	25
0x7ffffe88	24	24
0x7ffffe84	23	23
0x7ffffe80	22	22
0x7ffffe7c	22	22
0x7ffffe78	21	21
0x7ffffe74	20	20
0x7ffffe70	19	19
0x7ffffe6c	18	18
0x7ffffe68	18	18
0x7ffffe64	15	15
0x7ffffe60	14	14
0x7ffffe5c	13	13
0x7ffffe58	12	12
0x7ffffe54	12	12
0x7ffffe50	8	8
0x7ffffe4c	3	3
0x7ffffe48	3	3
0x7ffffe44	3	3
0x7ffffe40	2	2

Figure 1.3: Array BUF sorted

Play with different cache options

Concerning about the cache memory, the code was run with the standard values, then it was run again with the modified values selected by us. The Figure 1.4 shows the numbers reached with the differences applied. In the left, the initial cache has 128 words structured in 32 lines and 2 blocks. In the right, our cache has the same 128 words but structured in 16 lines and 8 blocks. Applying this modification, the number of `Misses` almost halved.

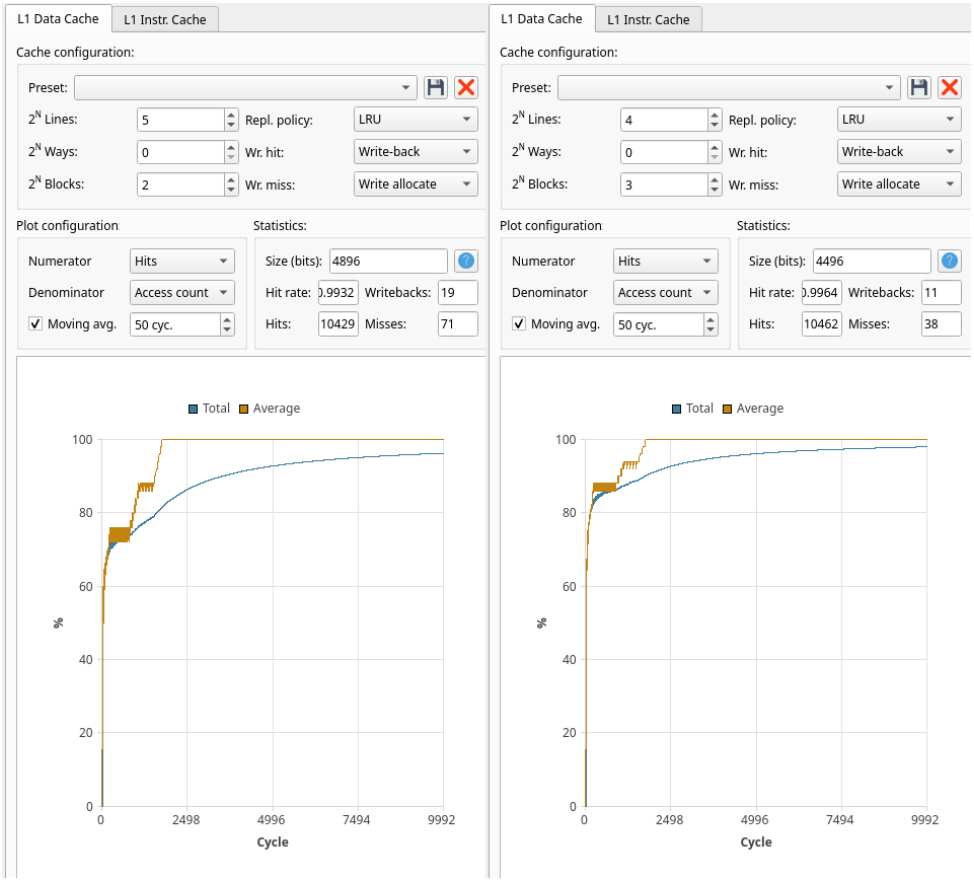

Figure 1.4: Left: Initial cache / Right: Modified cache

Using different processor models

Regarding the different processor models available on Ripes, some RISC-V architectures were selected to run the code and extract their overall performance. The Table 1.1 summarize all data obtained from the simulator, being:

- Number of cycles to execute the code
- Number of instructions executed
- Cycles per instruction
- Instructions per cycle

- Clock rate

The architectures used to compare the performance were:

- Single-cycle processor
- 5-stage processor
- 5-Stage processor w/o forwarding unit
- 5-stage processor w/o hazard detection

Analyzing the Table 1.1, it is possible to infer that the Clock Rate reported by Ripes is not a valid data point. A frequency of 21,15 KHz for a processor with a CPI = 1 is unusually low. By disregarding this aspect and concentrating on others data points, it becomes clear how the implementation of units for forwarding and hazard detection in the architecture contributes to increase the performance.

It is important to mention that the 5-stage processor without forwarding or hazard detection was also tested. However, the code did not executed as expected, as the architecture was unable to handle with the data hazards that occurred.

Table 1.1: Comparation between differents architectures

CPU	Cycles	Instrs. Retired	CPI	IPC	Clock Rate
Single-cycle processor	42770	42770	1	1	21.15 KHz
5-stage processor	58998	42770	1.38	0.725	5.70 KHz
5-Stage processor w/o forwarding unit	84929	42770	1.99	0.504	56.79 KHz
5-stage processor w/o hazard detection	64384	53094	1.21	0.825	45.47 KHz

Chapter 2

Simple Pipelining

I Deepen the understanding of the operation of a simple,but realistic, pipelined processor.

Select the 5-stage processor and simulate the insertion sort ELF binary for the following questions.

The simulated processor supports forwarding because the result of *x2* is not yet registered in the memory yet, it is still in the MEM stage. That is because the SW instruction needs the *x2* value, then it can stores the value inside *x1* in the right place. However *x2* is modified in *addi*, thus it is not registered but it is possible to see that they are not stalled at this point. This explanation is illustrated by the Figure 2.1.

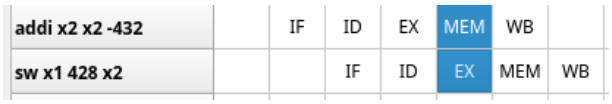

Figure 2.1: Forwarding - Pipeline Diagram

This implies that the processor can forward data instead of stalling. As a result, there is no need to waste two cycles, thus improving performance.

Explain the capabilities of the processor and illustrate them through examples

The processor's capabilities are satisfactory due to minimal stalling, except for exceptional cases where stalling occurs, typically when executing a jump instruction, necessitating the flushing of instructions. This is mitigated by the implementation of forwarding mechanisms.

In our scenario, we achieve a CPIAvg of 1.6 (8/5). The processor performance shown by the Figure 2.2 indicates that simulation get a CPI little bit lower (1.55) because on the first instruction there is practically no jump.

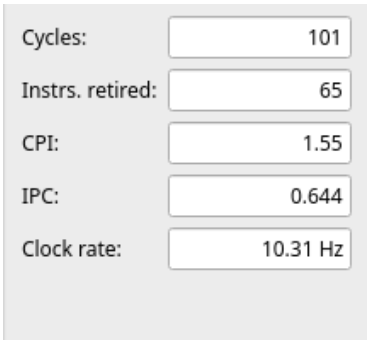

Figure 2.2: Processor Performance

Can you identify special cases for certain kinds of instructions?

It possible to identify a special case for the *LW* instructions for example. The pipeline diagram represented in the Figure 2.3 shows this situation.

lw x13 0 x15				IF	ID	EX	MEM	WB
sw x13 0 x14	MEM	WB			IF	ID	-	EX
addi x15 x15 4	EX	MEM	WB			IF	-	ID

Figure 2.3: Pipeline Diagram - LW Instruction

This occurs because loading a variable happens in the MEM stage instead of the Execute (EX) stage like the other instructions. Consequently, when the variable (for example, x13) is needed in the MEM stage, it must also be available at the same time in the EC stage for another operation (for example, store). This necessitates a one-cycle stall, after which the variable can be forwarded from the Write-back (WB) to the Execute (EX) stage. This process is illustrated by the Figure 2.4.

How does the simulator know how that the program has completed?

The simulator identifies the completion of the program upon encountering the *ecall* instruction in the ”IF” stage, signaling the need to halt.

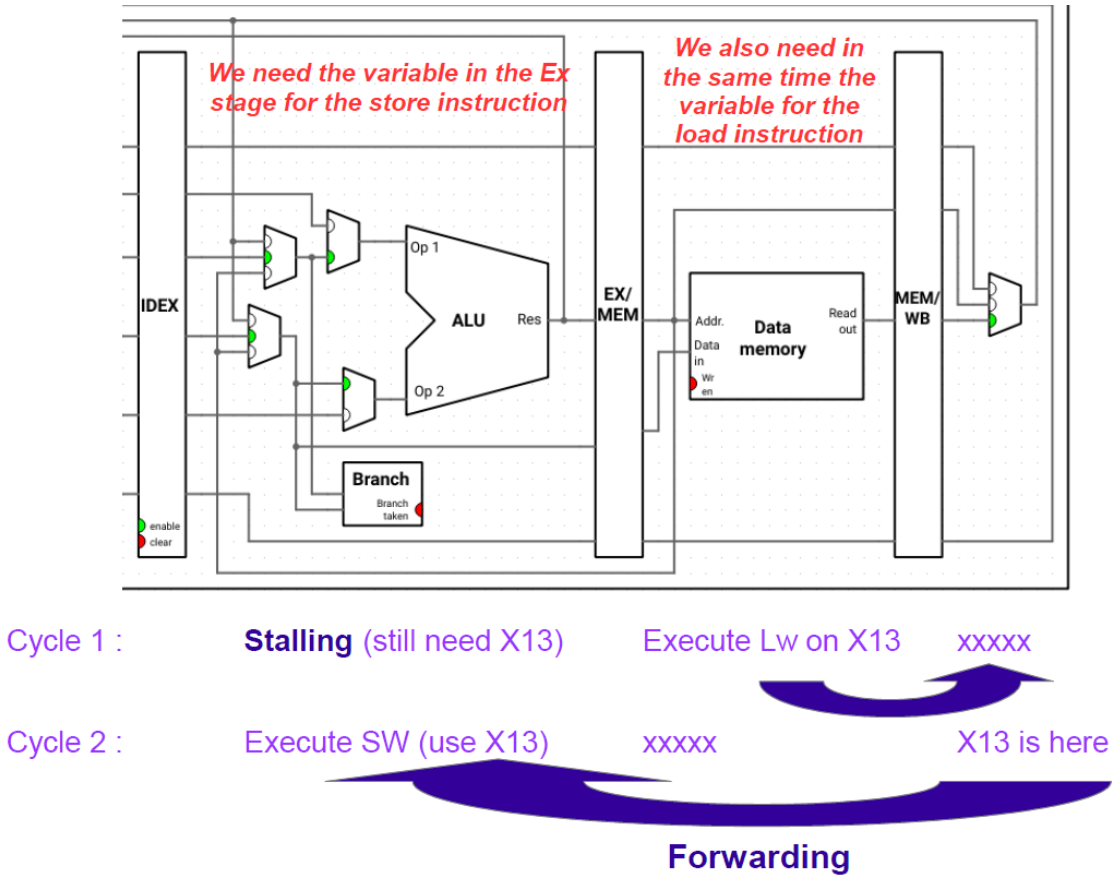

Figure 2.4: Forwarding Process

Chapter 3

Branches and Multiple-Issue

I Understand the impact of branches and more complex pipelines.

A 5-stage processor

In order to simulate the processor running the program, the 5-stage processor which can be visualized in the Figure 3.1 was selected.

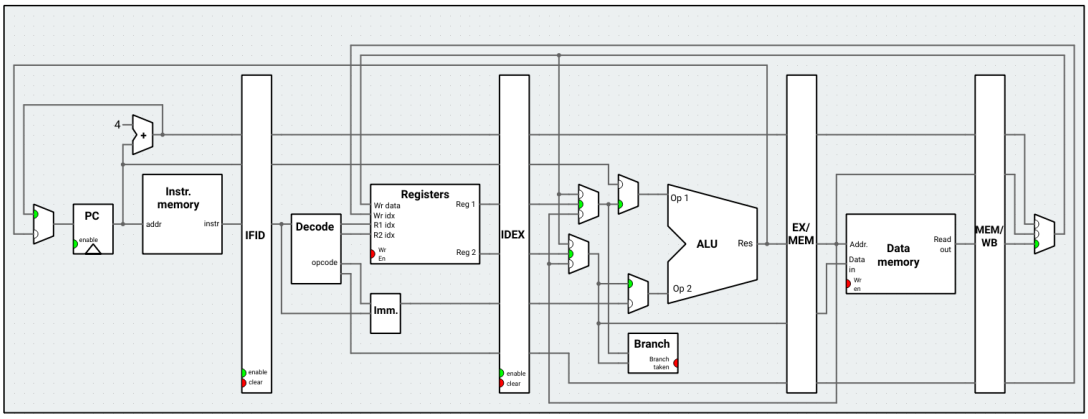

Figure 3.1: 5-Stage RISC-V Processor

Find examples in the pipeline diagram illustrating situations where a few instructions have not been completed

The Figure 3.2 shows the pipeline diagram for the program being run. It is possible to verify that two instructions have not been executed. These instructions are:

- addi x9 x0 0
- addi x19 x0 99

It happens because the previous instruction is a conditional branch: *bne x15 x12 -6*. When a branch is taken, the *PC* register is updated with the target address. However, the decision whether the branch will be taken or not only occurs into EX stage as well as the target address. Due to this delay of two stages, the following instructions are inserted into pipeline. Considering the situation shown in the figure below where the conditional branch is taken, these two instructions between the branch instruction and the target should not have been started. Therefore, the processor flushes them, resulting in two instructions uncompleted, as it shown in Figure 3.2.

	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
addi x8 x2 4	IF	ID	EX	MEM	WB																
addi x12 x15 396		IF	ID	EX	MEM	WB															
addi x14 x8 0			IF	ID	EX	MEM	WB														
lw x13 0 x15				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID
sw x13 0 x14					IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF
addi x15 x15 4						IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB		
addi x14 x14 4								IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB	
bne x15 x12 -16									IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB
addi x9 x0 0										IF	ID							IF	ID		
addi x19 x0 99											IF								IF		

Figure 3.2: Pipeline Diagram - Conditional Branch

Another example can be found in the pipeline diagram which is shown in the Figure 3.3. This new situation concerns about the jump instructions. These instructions are unconditional branches, in other words, it is not necessary to decide whether the branch will be taken or not, because it always will be taken, thus the processor has only to calculate the target address. As explained in the previous context, the new value of PC is calculated into EX stage, as a result, the processor must flushes the following two instructions.

addi x2 x2 -16	IF	ID	EX	MEM	WB		
sw x1 12 x2		IF	ID	EX	MEM	WB	
jal x1 -172 <main>			IF	ID	EX	MEM	WB
addi x17 x0 10				IF	ID		
ecall					IF		

Figure 3.3: Pipeline Diagram - Conditional Branch

Determining whether the processor has branch predictor

To determine whether the processor has branch predictor, it necessary to analyze it architecture. The datapath signal states when a conditional branch is taken, as illustrated by Figure 3.4, shows the address being calculated into EX stage and returning this value to PC register. Besides, there is a module to determine wheter the branch is taken or not. Since these two operations are made in this current stage, it is possible to infer that

the 5-stage processor available in the software Ripes does not have a branch predictor. Aiming to have a branch predictor, the module to determine a branch should be moved to ID stage as well as a signal extension module and a adder should be inserted.

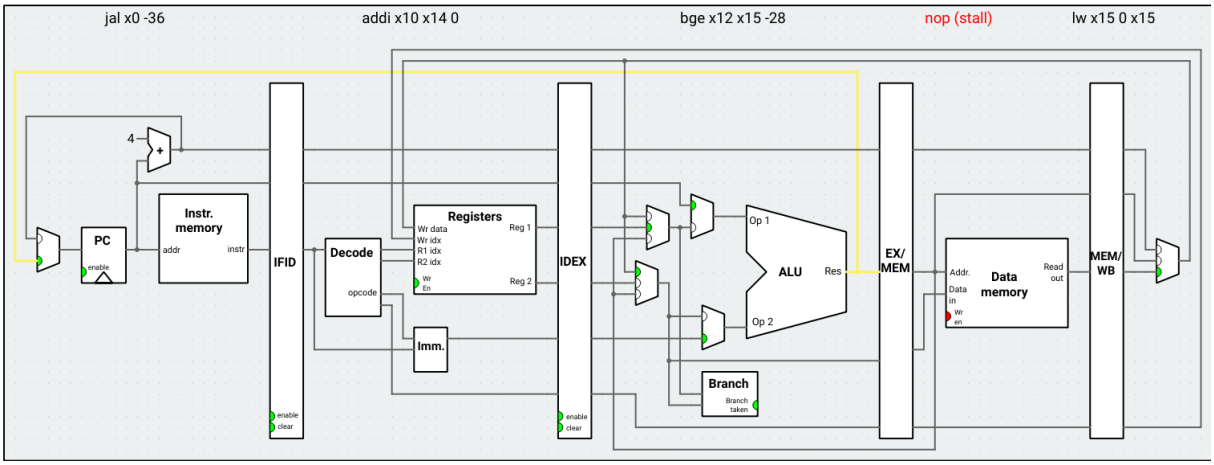

Figure 3.4: Processor Architecture - Conditional Branch

What would the optimal CPI be for such a simple pipelined processor?

In theory, the optimal CPI for a simple pipeline would be of 1. However this number cannot be reached due to many situations. Firstly, when the processor is energized, it requires 5 cycles to fill the pipeline then finishing an instruction. Besides, there are the penalty cycles caused by the hazards. One of them is the data hazard, when the data is not available yet for the next instruction, so the processor needs to stall the pipeline. In addition, as explained in this same section, when a conditional branch is taken the processor flushes two instructions, which contributes to increase the value of the CPI.

B 6-stage dual issue processor

For this section the 6-stage dual-issue processor was selected to simulate the processor running the program. It architecture can be visualized in the Figure 3.5 was selected.

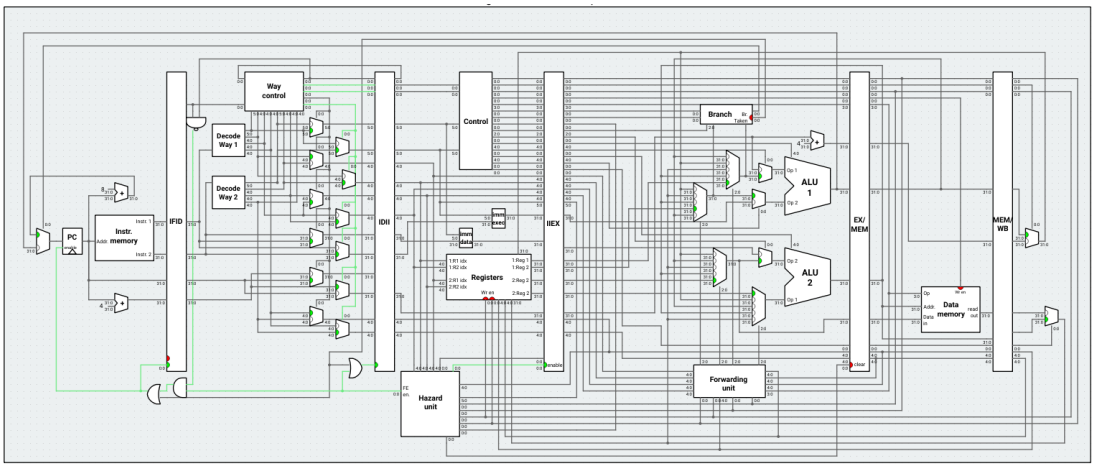

Figure 3.5: 6-Stage Dual-Issue RISC-V Processor

Analyze the pipeline's behaviour

The pipeline diagram for this new architecture is shown in Figure 3.6 and it will be used to analyze the pipeline's behaviour. It is possible to easily visualize how different are the both pipeline diagrams presented in this report. This new architecture allows to execute instructions, thus they are advancing at their pipeline stages simultaneously. Moreover, the diagram indicates an absence of a branch predictor in view of the fact that the following instructions of the instruction *bne x15 x12 -6*. Also, these flushes represent penalty cycles, which will be discussed in the next section.

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
addi x8 x2 4	II	EX	MEM	WB																				
addi x12 x15 396	II	EX	MEM	WB																				
addi x14 x8 0	ID	II	EX	MEM	WB																			
lw x13 0 x15	ID	II	EX	MEM	WB		IF	ID	II	EX	MEM	WB			IF	ID	II	EX	MEM	WB			IF	ID
sw x13 0 x14	IF	ID	II	-	EX	MEM	IF/WB	ID	-	II	-	EX	MEM	WB	IF	ID	-	II	-	EX	MEM	WB	IF	ID
addi x15 x15 4	IF	ID	II	-	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF
addi x14 x14 4		IF	ID	-	II	EX	MEM	IF/WB	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF
bne x15 x12 -16		IF	ID	-	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB
addi x9 x0 0			IF	-	ID	II				IF	-	ID	-	II				IF	-	ID	-	II		
addi x19 x0 99			IF	-	ID	II						IF	-	ID						IF	-	ID		
addi x18 x0 98					IF	ID						IF	-	ID						IF	-	ID		
sub x11 x19 x9					IF	ID								IF								IF		
addi x10 x8 0						IF								IF								IF		
jal x1 -160 <minIndex>						IF																		

Figure 3.6: Pipeline Diagram - 6-Stage Dual-Issue Processor

Another important point to mention about the pipeline's behaviour is when a data hazard occurs. Visualizing the Figure 3.7 we can detect a data hazard between the instructions *addi x2 x2 -432* and *sw x1 428 x2*. The store word instruction needs a value from x2 that are being calculated at the same time. For this reason, the storing process is stalled, consequently all instructions in IF, ID and II stages are equally stalled.

	6	7	8	9	10	11	12	13	14	15	16	17
addi x2 x2 -432	IF	ID	II	EX	MEM	WB						
sw x1 428 x2	IF	ID	-	II	EX	MEM	WB					
sw x8 424 x2		IF	-	ID	II	EX	MEM	WB				
sw x9 420 x2		IF	-	ID	-	II	EX	MEM	WB			
sw x18 416 x2				IF	-	ID	II	EX	MEM	WB		
sw x19 412 x2				IF	-	ID	-	II	EX	MEM	WB	
lui x15 0x11						IF	-	ID	II	EX	MEM	WB

Figure 3.7: Pipeline Diagram (Data Hazard) - 6-Stage Dual-Issue Processor

Discuss the optimal CPI for this processor

The discussion for the optimal CPI considering this 6-stage dual issue processor is similar to that made for the simple pipelined processor. As shown in the Figure 3.5, this architecture contains two modules of execution into EX stage, which would allow the processor to reach a CPI of 0.5. However, as expected, the processor cannot reach this incredible value. It happens because, as in the previous example, it is necessary to fill the pipeline (now there is one more stage) besides the hazards that occur during execution. In addition, this architecture allows to execute two different instructions as long as they do not have to access the memory at the same time, it means that it is not possible to run two load/store instructions simultaneously, for example. In that case, one of them and the instructions in previous stages are stalled.

Chapter 4

Caches

I Understand the performance of caches.

Analyzing the code 1.1 we can determine the number of instructions fetches and memory accesses. For instance, each instruction executed corresponds to an instruction fetch (we have to keep in mind that loops are fetched each time). So we can estimate the total number of instructions fetches to be the number of instructions plus the number of instructions per loop times how many times it's looped. In our case, we have one loop that copies the original content of the array, so executed 99 times; then the loop that calls the function minIndex is executed for 99-1 times; finally the function minIndex loops each time 99-i times, that is a total of $(98 + 1)\frac{99+1}{2} = 4950$. We also take into account the worst case, where the minIndex finds the index at last iteration.

$$n_{if} = 5 + 11 + 99 * 5 + 5 + 98 * 14 + 98 * (6 + 2 + 5) + 4950 * 8 + 8 = 42770 \quad (4.1)$$

For memory accesses we use the same logic to calculate the total:

$$n_{ma} = 1 + 5 + 99 * 2 + 98 * 4 + 4950 * 2 + 6 = 10502 \quad (4.2)$$

If we simulate this scenario using *Ripes* we can verify by looking in the *Cache* tab the numbers shown above. The caches have been configured to only hold one line and one block, to be comparable to the calculations done above (the cache is not useful in this scenario).

Looking at 4.1 confirms the calculated number of memory accesses ($1052 \approx 1033198$). In the same way, 4.7 shows the same number of instruction fetches ($42770 \approx 42771$).

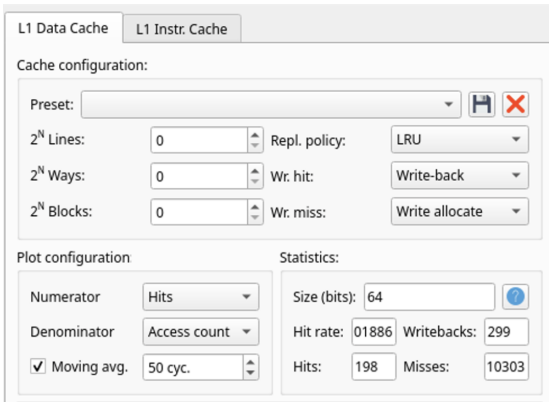

Figure 4.1: Data cache

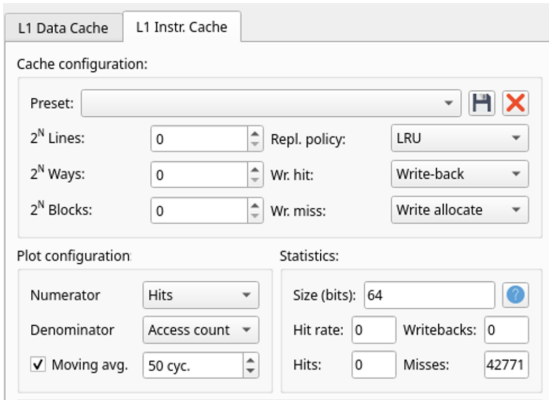

Figure 4.2: Instruction cache

A Describe cache configurations for the data/instruction caches respectively that results in the minimum number of cache misses, while also minimizing the cache size

In order to minimize cache misses and its size (it's important to optimize size as well, otherwise we could have a huge cache, but that would be expensive) we have to reason according to how memory accesses are done by our program. For instruction it should be clear that having the loop code in cache would be very beneficial for avoiding misses. For data, having the array in cache would be beneficial as we are always looping through it for sorting.

Data cache

We have two arrays being copied of int data type of size 99, so we will need at least $99 * 4 * 2 = 792$ bytes in cache (plus some stack variables). There are different ways we can achieve this. The most basic way is to have one line for each array and hold the whole array in the line. We configure the following way: $2^N = 1$ lines and $2^N = 7$ blocks (each block is 32 bits).

In this configuration we get a fairly high amount of misses. Executing step by step and monitoring the cache state we can see that during the copy of the arrays, both addresses map to the same entry, leading to *conflict misses*. For example, load address `0x11204` and store address `0x7ffff9e4` (and the following loop iterations). To avoid this conflict misses we should have a 2-way cache in order to allow both addresses to map to the same set but different entries. As we can see in 4.3, we have only 3 misses but one entry of the first set is not utilized. Let's reduce the number of blocks per entry. In this case, as shown by 4.4 misses only increment by 2 but all entries are used; and we have reduced the cache's size by 2 (8292 bits). Reducing even more the number of blocks (to $2^N = 5$) gives us a good performance as we can see in 4.5, but doesn't meet the criteria given by the exercise, that is, having single digit misses (we got 12). Moreover, both sets have the same tags, so we can suppose that some overlap of addresses occur between blocks en each entry. Let's try removing one line. The result is very promising, as shown by figure 4.6, misses are still one digit numbers (only 7) and we have effectively reduced cache size again by a factor of 2 (4148 bits).

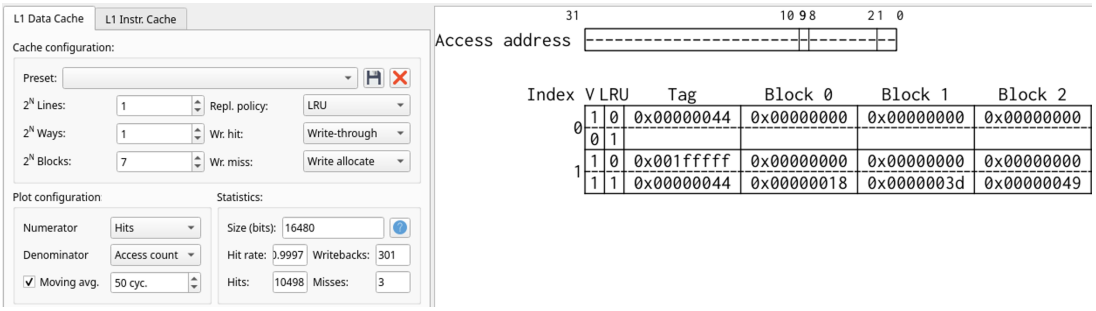

Figure 4.3: 2 way $2^N = 7$

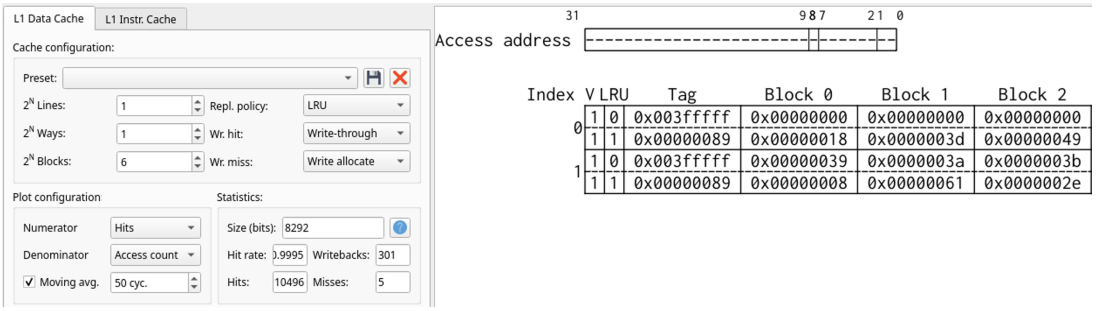

Figure 4.4: 2 way $2^N = 6$

Instruction cache

For the instruction cache we don't need that much memory as the program only has a few instructions (68 instructions). We can build a cache with only one line that holds the complete program ($2^N = 7$ blocks). Examining the results of the execution 4.7, we can see that we only get 1 miss, that corresponds to a *compulsory miss* at the first instruction fetch, then the whole program is available in cache.

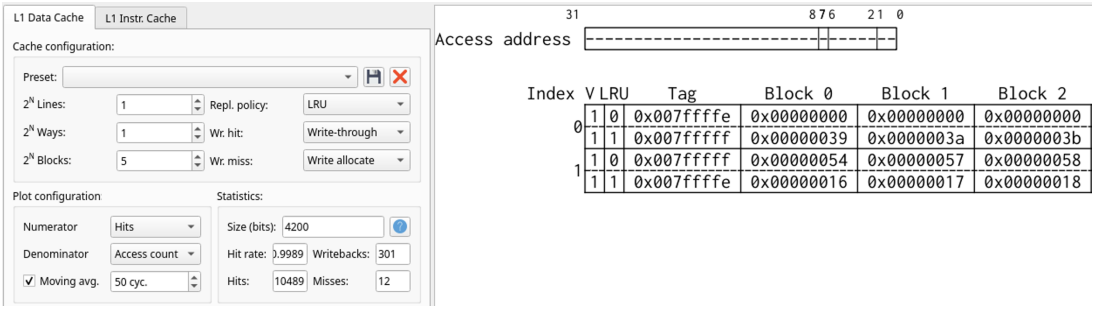

Figure 4.5: 2 way $2^N = 5$

Figure 4.6: 1 way $2^N = 6$

If we look closer at the values stored in cache, we have a lot of unused blocks. 68 instructions is close to 64, so we can look at a way of reducing the cache's size. Reducing the block size is not enough as we get a total of 199 misses (figure not shown to make the report more compact). This misses come from the loading of minIndex code. So each call produces two misses (one to get the function's code and one to get the main program code after the return). To improve this we might think that having two lines can separate loop code in one entry and the rest of the program in the other entry. By doing so we improve on cache misses (only 2), but we effectively double the cache size, so we are back to the starting position. Keeping with the same philosophy as before, that is, trying to use only 64 blocks total, let's reduce the blocks to $2^N = 5$, giving effectively 64 blocks total. As we can see in figure 4.8, we only get 4 misses but we reduced the cache size to 2100 bits. This will be the final design for the instruction cache.

Cache Misses

We have seen already two types of misses happening in our processor: *compulsory misses* when fetching new data/instructions for the first time, and *conflict misses* when two addresses map to the same entry. Let's analyze a *capacity miss* in our program, with the data cache configuration found at the end of the analysis, that is, 1 way and $2^N = 6$ blocks. In this case, when copying array input into buf, eventually we access the last block of the entry while looping through the array, so when we try to store/load a new value (for example storing at address `0x7fffff00`) we have a *capacity miss* because we don't have enough blocks to hold the data that we want (the array for instance). This example can be seen in figure 4.9.

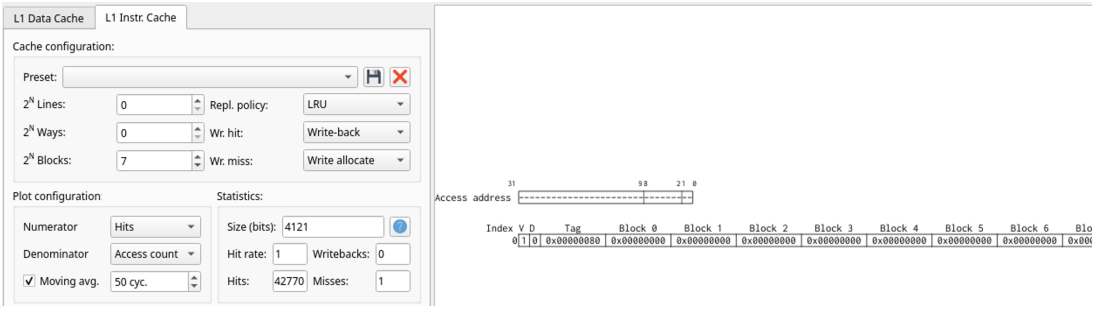

Figure 4.7: 1 line $2^N = 7$

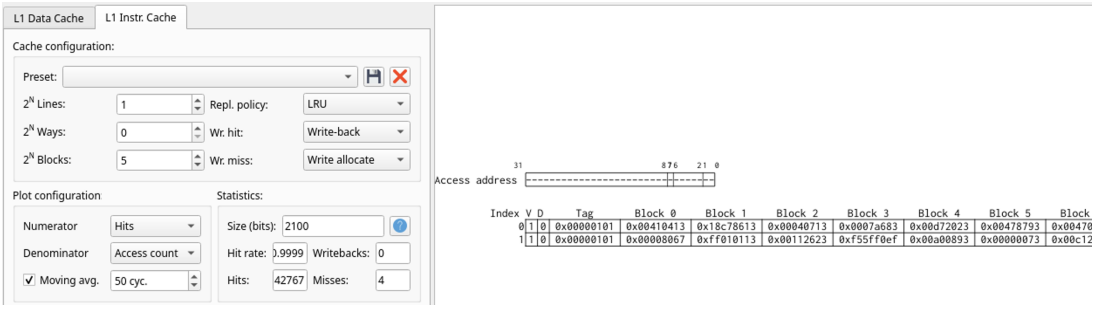

Figure 4.8: 2 lines $2^N = 5$

Cache policy

Cache policy refers to how entries are replaced in cache. There are two available strategies in our simulation: LRU, that considers entries' age, and random, that chooses randomly between entries. In our case it is interesting to use a LRU policy. The accesses made are always to the same addresses (buf array), so it is pertinent to keep the array in cache. If we used a random policy, we risk removing the entry corresponding to the array, hence provoking a new miss next time we access the array.

Write policy

Write policy refers to how memory stores are handled. We will study how policy in case of a cache miss compare.

We have two possible policies:

1. Write allocate: this strategy first loads block from backing storage and then proceeds as in a normal hit (either write through or write back, although usually combined with the latter).

2. Write no allocate: this strategy directly writes into backing storage in case of a miss.

The *write allocate* policy exploits temporal locality, as data is written in cache, expecting it to be read soon, and thus being a faster read. In our case this is beneficial because buf array is being modified in each loop[2].

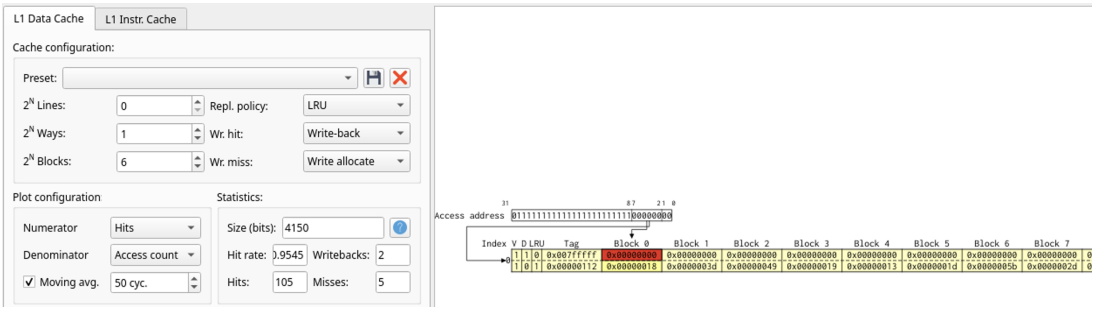

Figure 4.9: Capacity miss

Bibliography

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

[2] UC San Diego, *Computer Science and Engineering 141*