Télécom Paris

SE201: Execution Platforms
Project 2

Alaf do Nascimento Santos
Alessandro Mandrile
Ivan Luiz de Moura Matos

2022/2023

Contents

1 Setup

The project was started by downloading the base files provided with the Project Assignment. In summary, there was a C code file called `insertion-sort.c` to be analyzed throughout this work and a Makefile for its compilation.

After that, we launched a terminal and entered the project directory containing the Makefile. By typing `make`, the compilation was started and we could get the binary file `insertion-sort.elf`, as the result from the compilation process, to be analyzed using the *Ripes* simulator.

Since Ripes provides a graphical user interface, it offers a visualization of the execution of the program, allows us to simulate the program cycle-by-cycle, and permits to inspect the register and memory values. In this work we are going to use this tool as the main resource in our analysis.

For more information and clarifications about Ripes, it is important to consult the official Ripes documentation and the RISC-V instruction set manual [1] which has been the main reference in our projects.

2 RISC-V Tool Chain

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

2.1 RISC-V Tools

By running `make`, what is actually executed is

```
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
```

About the commands to force the build, `make -B` forces to make the target even if its recipe is already done. On the other hand, `make clean all` will before execute clean, deleting the binary file so that `make all`, which makes the elf file, will see it as missing and so will compile it.

About the compiler options we discovered that:

- `-nostdlib` tells the compiler not to link the C standard library, which would be needed only if we wanted to use `printf`, `malloc`, etc.
- `-nostartfiles` is used to tell the compile not to link the standard system startup files. These files contain directives and labels used to prepare the environment before the main(), like the `_start`, `_init`, `_fini` or the creation of stack, heap and other memory zone.

The two directives do similar jobs, which is to help linking programs for environments with particular constraints. One example are embedded systems running bare metal C, not having the standard C library, most of time for low memory reasons.

In this section we will further analyse the produced executable. The first loop copies the array `input` to a local, stack-defined `buff`.

```
     for(i = 0; i < SIZE; i++)
   100f8: 000117b7          lui a5,0x11
   100fc: 1a478793          addi a5,a5,420 # 111a4 <input>
   10100: 00410413          addi s0,sp,4
   10104: 18c78613          addi a2,a5,396
{
   10108: 00040713          mv a4,s0
```

We can see the load of the two addresses at 0x100f8-0x100fc and at 0x10100-0x10108. In the first it loads in `a5` 0x111a4, the address of input in `.data` section and the second loads the starting address of buff at `sp[4]`. In 0x10104 loads the cycle counter, by evaluating the arrive address of `a5` (0x111a4 + 396 = 0x111a4 + 99*4).

```
     buf[i] = input[i];
   1010c: 0007a683          lw a3,0(a5)
   10110: 00d72023          sw a3,0(a4)
```

This part copies values from input to buff.

```
         for(i = 0; i < SIZE; i++)
   10114: 00478793          addi a5,a5,4
   10118: 00470713          addi a4,a4,4
   1011c: fec798e3          bne a5,a2,1010c <main+0x2c>
}
```

This part is on the end of the cycle. It increments the access addresses to input and buff. If `a5` is at the arrive address, it returns to 1010c, the `lw`. Launching,

```
riscv64-unknown-elf-objdump -t insertion-sort.elf
```

we observe the line

```
000111a4 g     O .data 00000190 input
```

that confirms our previous ideas about how input is located and used.

PIC independent code is a way of writing assembly code where the execution doesn't depend on where the code is placed in memory. To accomplish that, every instruction referring memory, like store, load and branch targets, must point to an address either passed by a supervisor (like how it happens with sp, initially given by OS) or be related to the position of PC, which means related to the position of the code. In general, compiler always try to produce PIC, for example by avoiding branches to absolute addresses and, when not possible, they add a label to the program symbol table, that will be linked either by the linker, if in a statically linked program, or by a system library, like how it happens for syscalls on every OS. PIC code is important for security because with it allows the OS to randomize the address space layout of the process, so that an attacker could never guess the exact position where existing executable code is.

The problem of branching to relative addresses is that immediates are most of the times reduced in the length of the branch and may not reach their destination in case of very long codes. On risc-v an apposite instruction has been invented, the `auipc`, which loads in a register the upper part of pc + a 19 bit immediate. Its use is similar to that of the `lui`, but the latter would load an absolute value, while in the former the final address is dependent of the PC. Together with an `auipc`, it is used the `jalr` that can jump to any 32-bit value by adding a register to a 12 bit immediate value.

In the executable under examination we can see examples of PIC in the `bne` at 0x1011c and 0x10160 or with the `jal`. Another interesting way of writing PIC are the instructions

```
10190: 00a00893          li a7,10
10194: 00000073          ecall
```

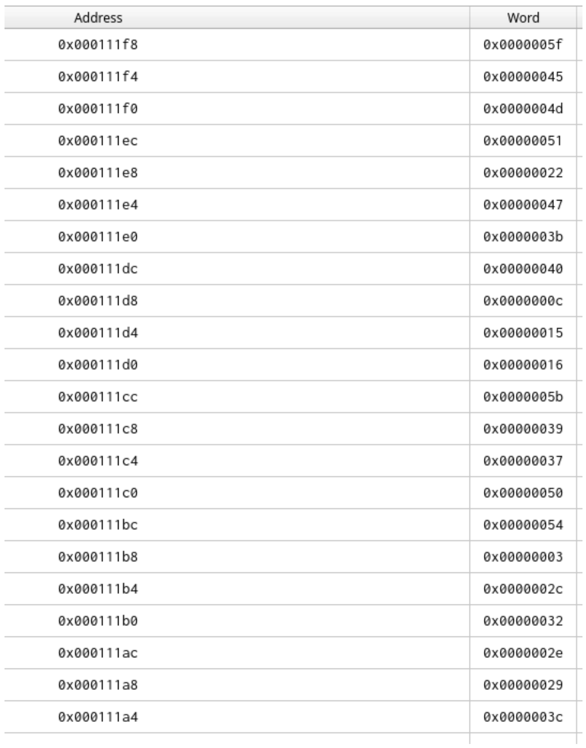
where the program does a system call, without having to call a function, like `open()` or `write()`, which in realty do hide an interrupt call.

The only part of this code that could seem position relevant is the _start label, which has a fixed address, but knowing how the elf execution works, a loader would take the symbol table first and look at _start label, which gives an offset from the start of the .text session, then it will evaluate the effective address by adding it to where it placed `.text`. Another proof is that running the command `readelf` on it will return, among the other things "There is no dynamic section in this file" and "There are no relocations in this file" which would appear on dynamic objects. A static object is by definition position independent.

2.2 Ripes

Aims: Explore the main functionalities of Ripes emulator.

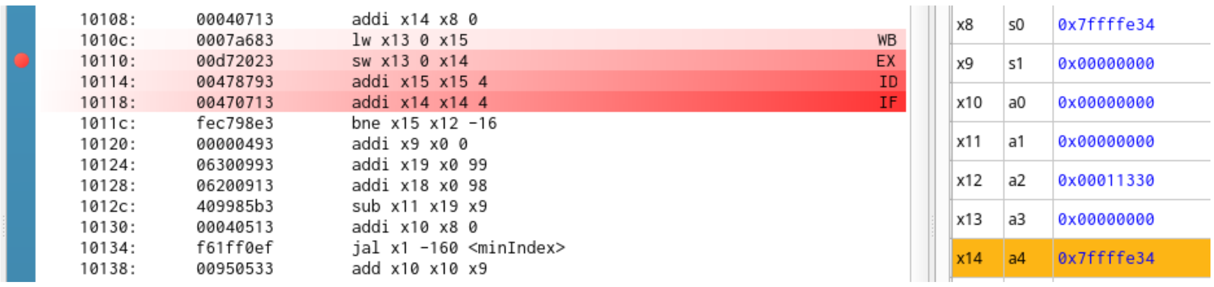
We ran the command
`riscv64-unknown-elf-objdump -s -j.data  insertion-sort.elf`
to analyse the content of `.data` section and appeared the same as in Memory section the emulator, as shown in Figure 1.



Address	Word
0x000111f8	0x0000005f
0x000111f4	0x00000045
0x000111f0	0x0000004d
0x000111ec	0x00000051
0x000111e8	0x00000022
0x000111e4	0x00000047
0x000111e0	0x0000003b
0x000111dc	0x00000040
0x000111d8	0x0000000c
0x000111d4	0x00000015
0x000111d0	0x00000016
0x000111cc	0x0000005b
0x000111c8	0x00000039
0x000111c4	0x00000037
0x000111c0	0x00000050
0x000111bc	0x00000054
0x000111b8	0x00000003
0x000111b4	0x0000002c
0x000111b0	0x00000032
0x000111ac	0x0000002e
0x000111a8	0x00000029
0x000111a4	0x0000003c

Figure 1: .data section in Memory view

To extract the address of **buff** we had to continue the execution for 3 clock cycle, so that after the WB stage of the previous **addi** the value, 0x7ffffe34 is accessible. The process is proved in Figure 2

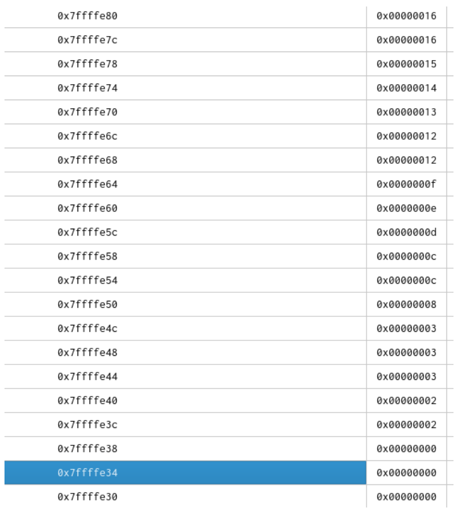


```
10108:   00040713   addi x14 x8 0
1010c:   0007a683   lw x13 0 x15          WB
10110:   00d72023   sw x13 0 x14          EX
10114:   00478793   addi x15 x15 4        ID
10118:   00470713   addi x14 x14 4        IF
1011c:   fec798e3   bne x15 x12 -16
10120:   00000493   addi x9 x0 0
10124:   06300993   addi x19 x0 99
10128:   06200913   addi x18 x0 98
1012c:   409985b3   sub x11 x19 x9
10130:   00040513   addi x10 x8 0
10134:   f61ff0ef   jal x1 -160 <minIndex>
10138:   00950533   add x10 x10 x9
```

x8	s0	0x7ffffe34
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00011330
x13	a3	0x00000000
x14	a4	0x7ffffe34

Figure 2: Address written to register after addi WB

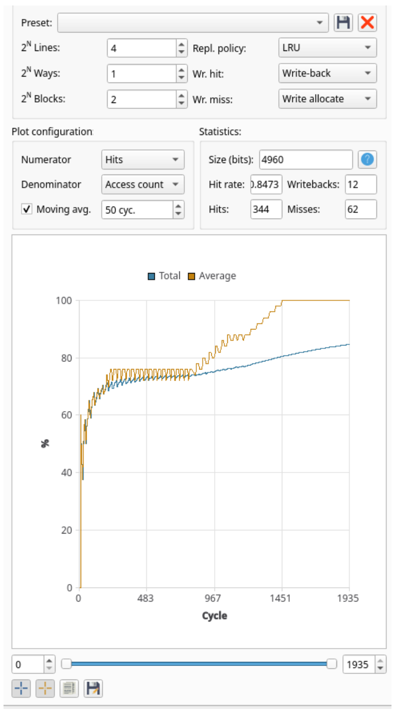
We let the program run and we observed that, starting from 0x7ffffe34 all the values were ordered, as shown in Figure 3



0x7ffffe80	0x00000016
0x7ffffe7c	0x00000016
0x7ffffe78	0x00000015
0x7ffffe74	0x00000014
0x7ffffe70	0x00000013
0x7ffffe6c	0x00000012
0x7ffffe68	0x00000012
0x7ffffe64	0x0000000f
0x7ffffe60	0x0000000e
0x7ffffe5c	0x0000000d
0x7ffffe58	0x0000000c
0x7ffffe54	0x0000000c
0x7ffffe50	0x00000008
0x7ffffe4c	0x00000003
0x7ffffe48	0x00000003
0x7ffffe44	0x00000003
0x7ffffe40	0x00000002
0x7ffffe3c	0x00000002
0x7ffffe38	0x00000000
0x7ffffe34	0x00000000
0x7ffffe30	0x00000000

Figure 3: Array sorted, on the stack

Regarding caches, we tried a cache of 128 words, to be able to contain all the 99 values, divided in 2 ways to avoid conflicts while copying from one array to the other. For blocks we used a set of 4 blocks, which brings to 16 lines, a level under whom we were unsafe to go. In Figure 4 you can see the Cache hit percentage over the number of accesses after one cycle of main's sorting `for`.



Preset:

2^N Lines:	4	Repl. policy:	LRU
2^N Ways:	1	Wr. hit:	Write-back
2^N Blocks:	2	Wr. miss:	Write allocate

Plot configuration

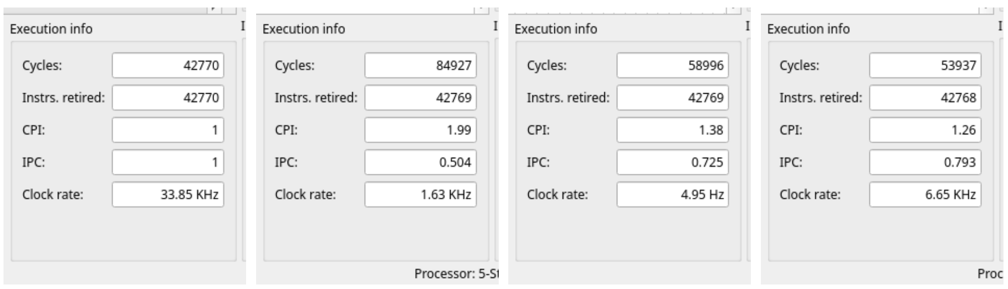
Numerator	Hits
Denominator	Access count
✓ Moving avg.	50 cyc.

Statistics:

Size (bits):	4960		
Hit rate:	).8473	Writebacks:	12
Hits:	344	Misses:	62

Figure 4: Cache hit chart during sort

We tested overall performance of different architectures for the same program. In Figure 5 you can see, from left to right, the average CPI and clock frequency at which



Execution info	Execution info	Execution info	Execution info
Cycles: 42770	Cycles: 84927	Cycles: 58996	Cycles: 53937
Instrs. retired: 42770	Instrs. retired: 42769	Instrs. retired: 42769	Instrs. retired: 42768
CPI: 1	CPI: 1.99	CPI: 1.38	CPI: 1.26
IPC: 1	IPC: 0.504	IPC: 0.725	IPC: 0.793
Clock rate: 33.85 KHz	Clock rate: 1.63 KHz	Clock rate: 4.95 Hz	Clock rate: 6.65 KHz
	Processor: 5-St		Proc

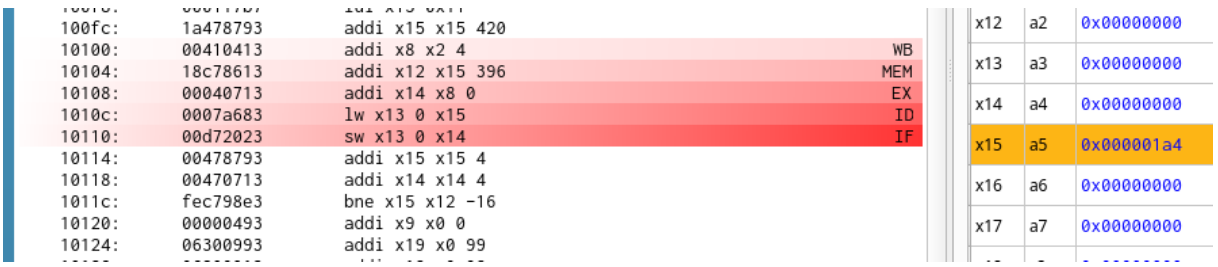
Figure 5: Processor performance comparison

they work. We then evaluated the performance gain, with respect to the single cycle CPU, as:

$$\frac{P_x}{P_s} = \frac{Clk_x}{Clk_s}\frac{CPI_s}{CPI_X} \tag{1}$$

Unfortunately the clock rates represented by Ripes are either invalid or imprecise, because we have the single cycle CPU, which has inherently a CPI of 1, showing the higher clock rate, 33 KHz. That would mean that 50 years of development have been wasted, or that old, simple pipeline processors like those seen in the course had hazard management logic so sophisticated that they would erase any gain obtained by splitting the data path in stages.

We also noticed that 5-stage processor without forwarding or hazard detection wasn't able to execute the program as intended, because as Figure 6 shows, the lack of either a stall or forwarding, make it impossible to manage correctly the data hazard happening at 0x10108, were `x8` should be used but it it isn't yet written, leading `x14` to a wrong value.



```
100fc:    1a478793    addi x15 x15 420
10100:    00410413    addi x8 x2 4          WB
10104:    18c78613    addi x12 x15 396      MEM
10108:    00040713    addi x14 x8 0         EX
1010c:    0007a683    lw x13 0 x15          ID
10110:    00d72023    sw x13 0 x14          IF
10114:    00478793    addi x15 x15 4
10118:    00470713    addi x14 x14 4
1011c:    fec798e3    bne x15 x12 -16
10120:    00000493    addi x9 x0 0
10124:    06300993    addi x19 x0 99
```

x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x000001a4
x16	a6	0x00000000
x17	a7	0x00000000

Figure 6: Cache hit chart during sort

3 Simple Pipelining

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

In the previous section we had a first contact with the functionalities that Ripes can offer. In this section we will use different functions of this tool to explore the processor pipeline. Using the Ripes simulator, we have selected the 5-stage processor, as illustrated by Figure 7, in order to simulate the insertion sort ELF binary and explore the operation of the processor pipeline.

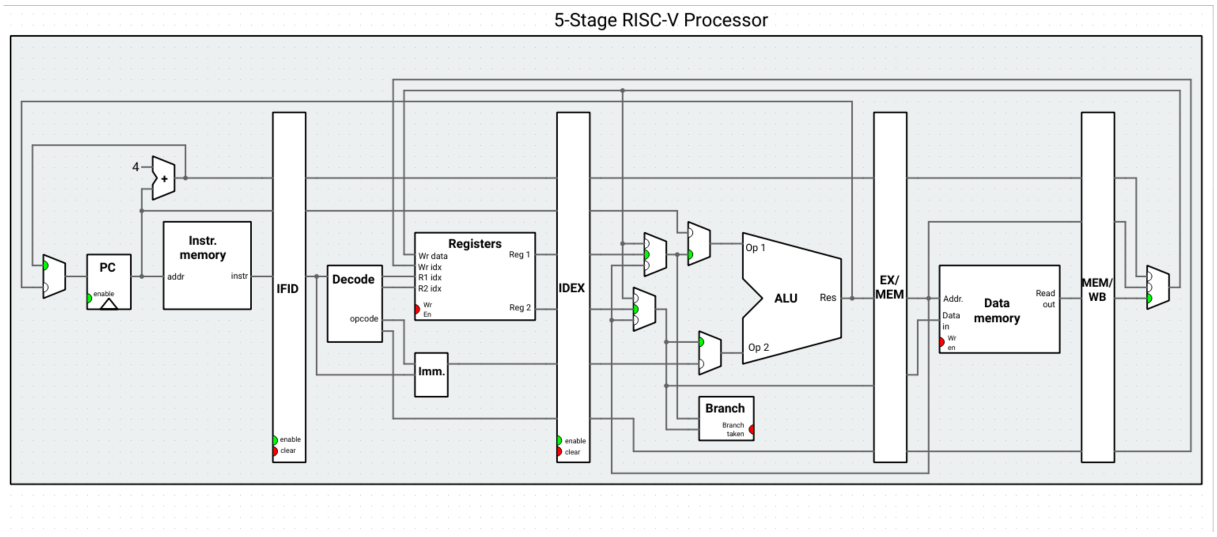
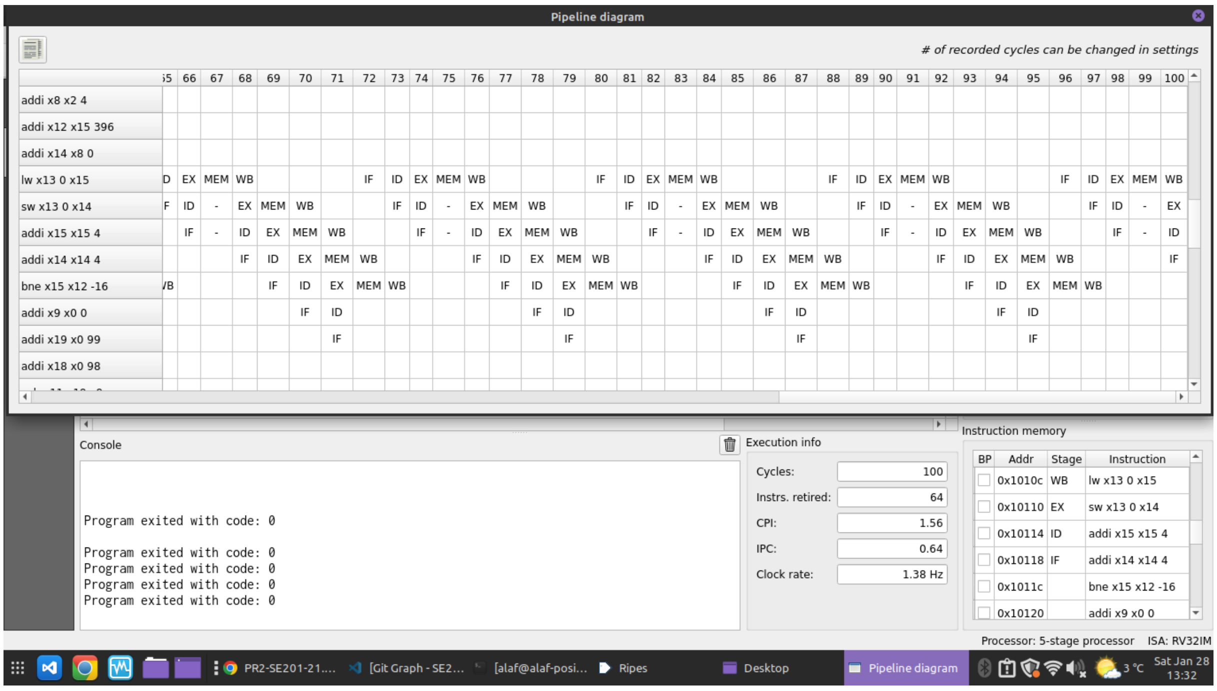


Figure 7: 5-stage processor

First of all, looking for the pipeline diagram after 100 executed cycles, we can see that our simulated processor supports forwarding. Since the use of data forwarding is a kind of optimization used to minimise the deficits due to stalls, when for example we have a data hazard because of a operation that needs a previous results and the earlier operation has not been finished, there are some cases where the processor must detect that we can just use the data from MEM or WB in EX, in order to not spend time waiting for a result (which in fact already exists, i.e., that was already computed).

Figure 8 shows the obtained diagram, where it is possible to see the forwarding operation acting for example in the instruction `sw x13, 0, x14` because it is needed the `x13` value from the previous instruction (`lw x13, 0, x15`) in order to store it at the address `mem[0 + x14]`. The previous instruction has not been done yet, therefore the current instruction needs to forward the data from the ALU. In other words, it can be seen that the `sw x13, 0, x14` instruction doesn't wait for the previous instruction to arrive at the `WB` stage before executing. It is possible to see, based on the number of stalls, that the processor implements forwarding from the MEM stage to the EX stage and in the case the forward was implemented at the EX stage there would not be a stall.


Pipeline diagram

# of recorded cycles can be changed in settings

	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
addi x8 x2 4																																				
addi x12 x15 396																																				
addi x14 x8 0																																				
lw x13 0 x15	D	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB
sw x13 0 x14	F	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF	ID	-	EX
addi x15 x15 4		IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB			IF	-	ID
addi x14 x14 4				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF
bne x15 x12 -16	/B				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB			
addi x9 x0 0						IF	ID							IF	ID							IF	ID							IF	ID					
addi x19 x0 99							IF								IF								IF								IF					
addi x18 x0 98																																				

Console

```
Program exited with code: 0

Program exited with code: 0
Program exited with code: 0
Program exited with code: 0
Program exited with code: 0
```

Execution info

Cycles:	100
Instrs. retired:	64
CPI:	1.56
IPC:	0.64
Clock rate:	1.38 Hz

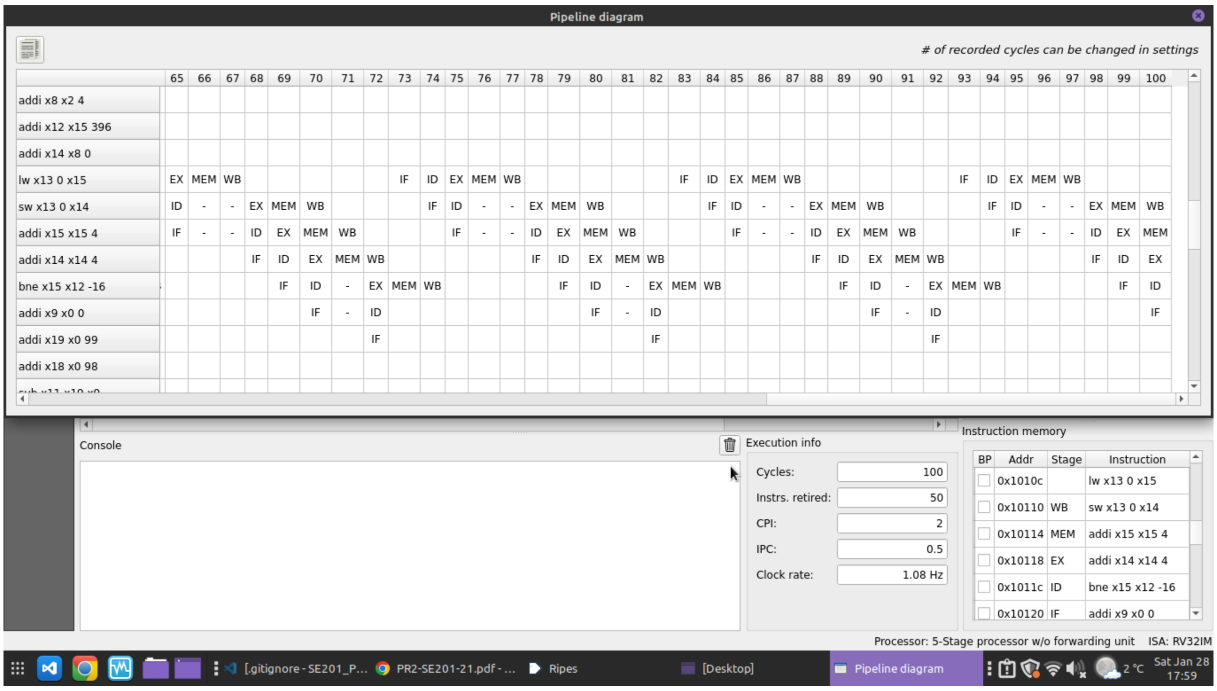
Instruction memory

BP	Addr	Stage	Instruction
☐	0x1010c	WB	lw x13 0 x15
☐	0x10110	EX	sw x13 0 x14
☐	0x10114	ID	addi x15 x15 4
☐	0x10118	IF	addi x14 x14 4
☐	0x1011c		bne x15 x12 -16
☐	0x10120		addi x9 x0 0

Processor: 5-stage processor ISA: RV32IM

Figure 8: Pipeline diagram for 100 cycles performed

Figure 9 illustrates the case where we simulated the program using another processor, that one is exactly as the previous one, but no forwarding unit. Now we have clearly more stalls compared to the previous simulation.


Pipeline diagram

# of recorded cycles can be changed in settings

	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
addi x8 x2 4																																				
addi x12 x15 396																																				
addi x14 x8 0																																				
lw x13 0 x15	EX	MEM	WB						IF	ID	EX	MEM	WB						IF	ID	EX	MEM	WB						IF	ID	EX	MEM	WB			
sw x13 0 x14	ID	-	-	EX	MEM	WB				IF	ID	-	-	EX	MEM	WB				IF	ID	-	-	EX	MEM	WB				IF	ID	-	-	EX	MEM	WB
addi x15 x15 4	IF	-	-	ID	EX	MEM	WB				IF	-	-	ID	EX	MEM	WB				IF	-	-	ID	EX	MEM	WB				IF	-	-	ID	EX	MEM
addi x14 x14 4				IF	ID	EX	MEM	WB						IF	ID	EX	MEM	WB						IF	ID	EX	MEM	WB						IF	ID	EX
bne x15 x12 -16					IF	ID	-	EX	MEM	WB					IF	ID	-	EX	MEM	WB					IF	ID	-	EX	MEM	WB					IF	ID
addi x9 x0 0						IF	-	ID								IF	-	ID								IF	-	ID								IF
addi x19 x0 99								IF										IF										IF								
addi x18 x0 98																																				

Console

Execution info

Cycles:	100
Instrs. retired:	50
CPI:	2
IPC:	0.5
Clock rate:	1.08 Hz

Instruction memory

BP	Addr	Stage	Instruction
☐	0x1010c		lw x13 0 x15
☐	0x10110	WB	sw x13 0 x14
☐	0x10114	MEM	addi x15 x15 4
☐	0x10118	EX	addi x14 x14 4
☐	0x1011c	ID	bne x15 x12 -16
☐	0x10120	IF	addi x9 x0 0

Processor: 5-Stage processor w/o forwarding unit ISA: RV32IM

Figure 9: Pipeline diagram for 100 cycles performed - no forwarding unit

Unfortunately forward is not always possible and this can be seen in the pipeline diagram generated for our processor with forward unit. We still have a stall cycle, when `sw` is about to go to the EX state, `x13` is not yet updated by the `lw` instruction and it is necessary to wait for the next clock cycle to execute the step.

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

According to the Ripes documentation, there are different environment calls that `ecall` can perform (for example, printing to console) and some of them are supported by the simulator. Table 1 shows the different environment calls supported by Ripes. The documentation also says that the table is valid for version 2.0.0 of the simulator.

a7	a0	Name	Description
1	(integer to print)	print_int	Prints the value located in a0 as a signed integer
2	(float to print)	print_float	Prints the value located in a0 as a floating point number
4	(pointer to string)	print_string	Prints the null-terminated string located at address in a0
10	-	exit	Halts the simulator
11	(char to print)	print_char	Prints the value located in a0 as an ASCII character
34	(integer to print)	print_hex	Prints the value located in a0 as a hex number
35	(integer to print)	print_bin	Prints the value located in a0 as a binary number
36	(integer to print)	print_unsigned	Prints the value located in a0 as an unsigned integer
93	(status code)	exit	Halts the simulator and exits with status code in a0

Table 1: Ripes supports environment calls

The documentation also says that to stop the Ripes simulator, either use the `ecall` convention, or we must branch to a label that is located at the end of the `.text` segment in the source code (an ending label). If we look at the instructions executed at each clock cycle in the Editor tab, we can notice that the last instruction before program termination is an `ecall`, like represented in Figure 10. So for our case, we are dealing with an `ecall` convention as illustrated by Table 1. Basically the instruction `addi x17 x0 10` writes 10 into register `x17` (`a7`) and then the simulator is stopped by the `ecall`, by calling the system and then the system terminates the program.

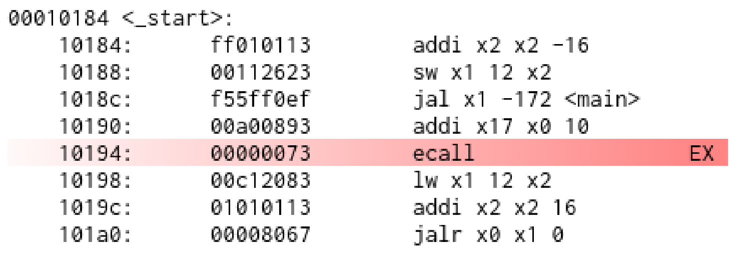

```
00010184 <_start>:
    10184:      ff010113        addi x2 x2 -16
    10188:      00112623        sw x1 12 x2
    1018c:      f55ff0ef        jal x1 -172 <main>
    10190:      00a00893        addi x17 x0 10
    10194:      00000073        ecall               EX
    10198:      00c12083        lw x1 12 x2
    1019c:      01010113        addi x2 x2 16
    101a0:      00008067        jalr x0 x1 0
```

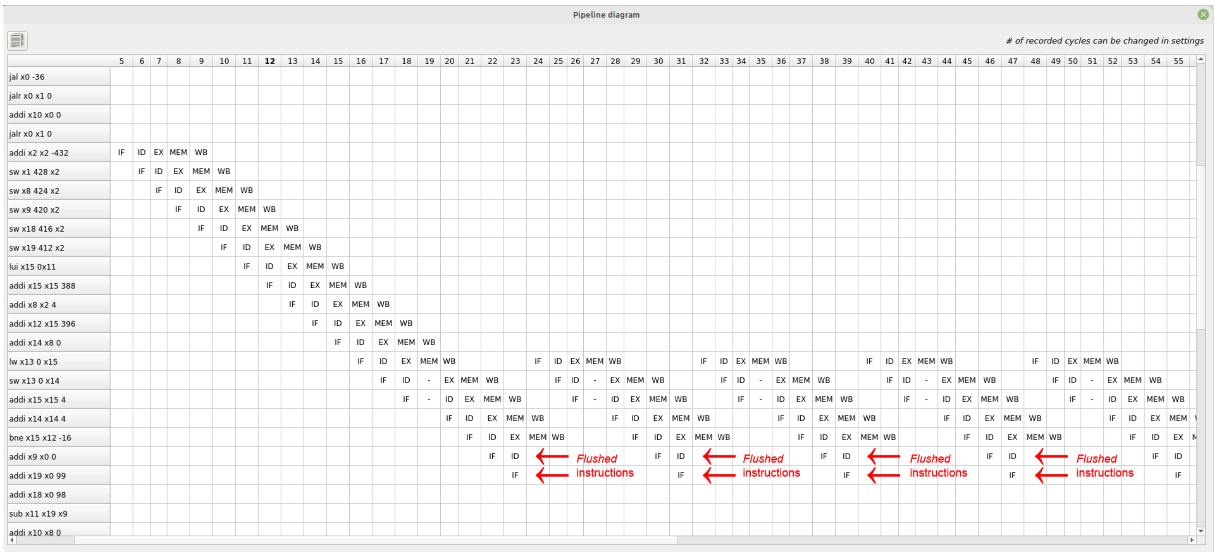
Figure 10: Stopping the simulator

4 Branches and Multiple-Issue

Aims: Understand the impact of branches and more complex pipelines.

In the following, we discuss the observed behaviour of the 5-stage pipelined processor when it executes the "insertion sort" program.

By looking at the pipeline diagram, we can observe that the two *consecutive* instructions `addi x9 x0 0` (located at the address `0x10100`) and `addi x19 x0 99` (located at the address `0x10104`) do not complete their execution. This happens due to the fact that the instruction that directly precedes them in the memory (at address `0x100fc`) is a conditional branch instruction (`bne x15 x12 -16`) which provokes the interruption (*flushing*) of the two following instructions when the condition it tests is true (i. e., when `x15` and `x12` are different). The flushing occurs because the two instructions following the `bne` shouldn't be executed, due to the branching; instead, it is the instruction located at the target address of the branch (in this case, `0x100ec`) that needs to be executed, after `bne` (when the branch is taken). Hence, when the tested condition is true, the instruction `addi x9 x0 0` (the first instruction after the `bne`) is flushed at its `ID` state; The instruction `addi x19 x0 99` (which is the second one after the `bne`) is flushed at its `IF` state. In Figure 11, the instruction's executions that are flushed are indicated by red arrows, and the stalls (due to the load instructions) are indicated by hyphens ("-") inside the corresponding cells.


Pipeline diagram

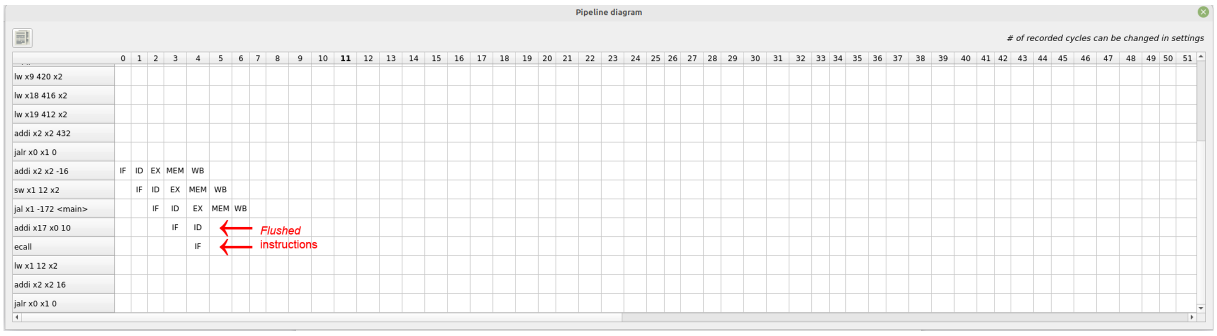
# of recorded cycles can be changed in settings

	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
jal x0 -36																																																			
jalr x0 x1 0																																																			
addi x10 x0 0																																																			
jalr x0 x1 0																																																			
addi x2 x2 -432	IF	ID	EX	MEM	WB																																														
sw x1 428 x2		IF	ID	EX	MEM	WB																																													
sw x8 424 x2			IF	ID	EX	MEM	WB																																												
sw x9 420 x2				IF	ID	EX	MEM	WB																																											
sw x18 416 x2					IF	ID	EX	MEM	WB																																										
sw x19 412 x2						IF	ID	EX	MEM	WB																																									
lui x15 0x11							IF	ID	EX	MEM	WB																																								
addi x15 x15 388								IF	ID	EX	MEM	WB																																							
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	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB			
sw x13 0 x14													IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB			IF	ID	-	EX	MEM	WB	
addi x15 x15 4														IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB			IF	-	ID	EX	MEM	WB
addi x14 x14 4																IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM
bne x15 x12 -16																	IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX	MEM	WB				IF	ID	EX
addi x9 x0 0																		IF	ID							IF	ID							IF	ID							IF	ID							IF	ID
addi x19 x0 99																			IF								IF								IF								IF								IF
addi x18 x0 98																																																			
sub x11 x19 x9																																																			
addi x10 x8 0																																																			

Figure 11: Indication of the instructions that are flushed due to taken branches

A similar situation can be noticed at the beginning of the execution of the program. The instructions `addi x17 x0 10` (at address `0x10170`) and `ecall` (at `0x10174`) are flushed at their `ID` and `IF` stages, respectively, because the instruction that precedes them (`jal x1 -172 <main>`, at address `0x1016c`) provokes a "*jump*" to the beginning of the "`main`" function. For this reason, the execution of the `addi` and `ecall` are interrupted.

Figure 12 indicates by red arrows the instructions that are flushed at the beginning of the program, due to a “jump”.


Pipeline diagram

# of recorded cycles can be changed in settings

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
lw x9 420 x2																																																				
lw x18 416 x2																																																				
lw x19 412 x2																																																				
addi x2 x2 432																																																				
jalr x0 x1 0																																																				
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																																															
lw x1 12 x2																																																				
addi x2 x2 16																																																				
jalr x0 x1 0																																																				

Figure 12: Indication of the instructions that are flushed due to the jump

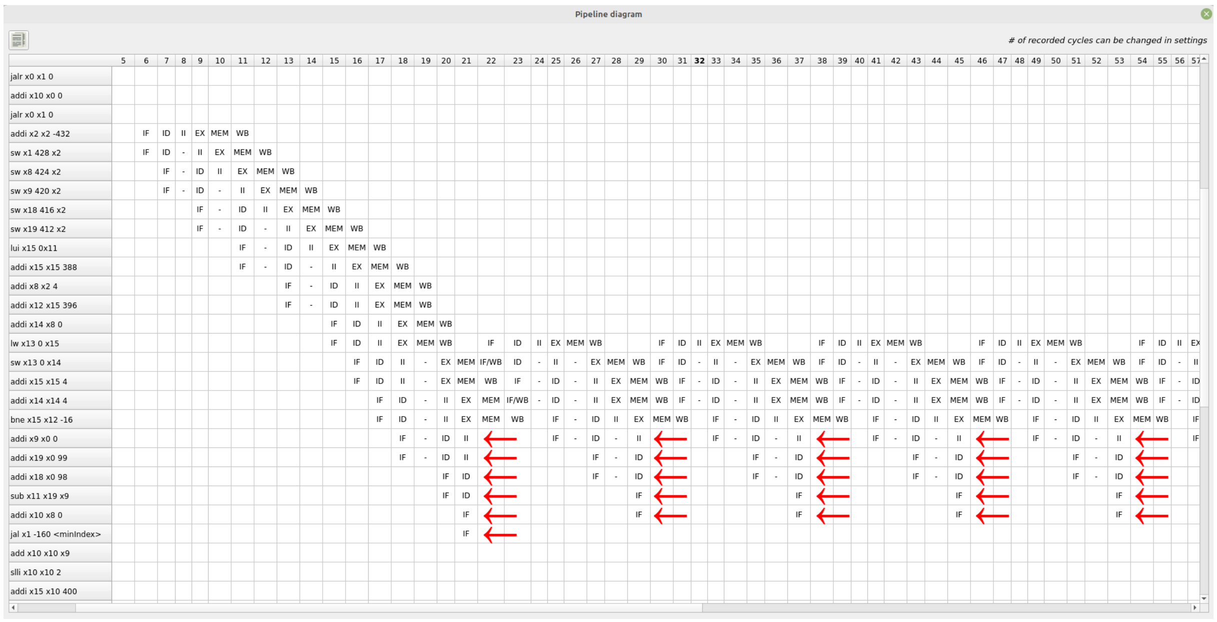
By examining the pipeline diagram and looking at the way the branches are handled, we may conclude that the processor does not have a *branch predictor*. This follows from the fact that the branch corresponding to the instruction `bne x15 x12 -16` (at `0x100fc`) is *always* predicted as *untaken* (which causes the two instructions after `bne` to be always flushed), even if in reality the branch is always being *taken* during the execution of the program, as shown in Figure 11. This shows that there isn’t a (*dynamic*) predictor that “learns” based on the “history” of the program (i.e., based on its past behavior on runtime). Even if the processor disposed of a *static* branch predictor, it might be set to always predict that this specific branch is untaken, which is not much appropriate for the program, since it’s always taken.

The optimal value of the processor’s CPI is equal to 1. The processor does not achieve the optimal CPI value due to the misprediction of branches and to the “stallings” associated with the execution of the load instructions, which are factors that contribute to a higher CPI value (to each one of the flushed instructions after a taken branch and to each stall corresponds a penalty). In fact, in the analysis of the processor discussed in the lectures (also without branch predictor), the value obtained for the CPI was 1,47 – which also follows from the penalties caused by the branch mispredictions and by the stalls occurred during the execution of the load instructions.

We tried to write a modified version of the program, counting in a rough way the number of branches performed during execution, to understand how many cycles do we really lose. This version of the code can be found in AppendixD and, to explain briefly, it increments a `cycle_counter` every time we do a cycle, meaning one conditional branch, and a `function_call` counting the number of branch to function or returns we perform. The nature of different counters has to be seen in the fact that most of function calls and returns could be statically predicted, while cycles are dynamic. We obtained a sum of 5146 cycles and 196 function calls; considering that the test cpu committed 42769 instructions in 58996, by removing or reducing to one the cycle penalty for branches, we would have the program executed in 48312 cycles in the first case and 53654 in the second. This can be done by adding a branch target predictor in the ID stage and possibly exploiting a one clock branch delay slot, effectively removing flush. Such a processor could achieve a CPI of 1,13 to 1,25 which translates to a total speedup of 10,4% to 22%. We must anyway point out that our evaluation was rough by considering a branch delay slot always used and a predictor that predicts always correctly, which is a border case, but still close to the reality in such a program, where would miss-predict just 1% of times.

Now, based on the results of the simulation, we discuss the behaviour of the 6-stage dual-issue processor when it executes the "insertion sort" program.

As shown in Figure 13, we can see that, at each cycle, two instructions advance in their pipeline stages simultaneously. We notice that the branches associated to the instruction `bne x15 x12 -16` are generally mispredicted, which causes the flushing of the execution of the instructions that follow the branch instruction and that had already started their pipeline sequence of stages. In Figure 13, the flushings are indicated by red arrows, and the stalls (due to the data hazards) are indicated by hyphens inside the cells. Those factors correspond to penalties and therefore provoke the augmentation of the processor's overall CPI.



Pipeline diagram

# of recorded cycles can be changed in settings

	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57
jalr x0 x1 0																																																					
addi x10 x0 0																																																					
jalr x0 x1 0																																																					
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																																								
addi x15 x15 388							IF	-	ID	-	II	EX	MEM	WB																																							
addi x8 x2 4									IF	-	ID	II	EX	MEM	WB																																						
addi x12 x15 396									IF	-	ID	II	EX	MEM	WB																																						
addi x14 x8 0											IF	ID	II	EX	MEM	WB																																					
lw x13 0 x15											IF	ID	II	EX	MEM	WB		IF	ID	II	EX	MEM	WB			IF	ID	II	EX	MEM	WB			IF	ID	II	EX	MEM	WB			IF	ID	II	EX	MEM	WB			IF	ID	II	EX
sw x13 0 x14												IF	ID	II	-	EX	MEM	IF/WB	ID	-	II	-	EX	MEM	WB	IF	ID	-	II	-	EX	MEM	WB	IF	ID	-	II	-	EX	MEM	WB	IF	ID	-	II	-	EX	MEM	WB	IF	ID	-	II
addi x15 x15 4												IF	ID	II	-	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID
addi x14 x14 4													IF	ID	-	II	EX	MEM	IF/WB	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID	-	II	EX	MEM	WB	IF	-	ID
bne x15 x12 -16													IF	ID	-	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB		IF	-	ID	II	EX	MEM	WB		IF
addi x9 x0 0														IF	-	ID	II	←			IF	-	ID	-	II	←			IF	-	ID	-	II	←			IF	-	ID	-	II	←			IF	-	ID	-	II	←			IF
addi x19 x0 99														IF	-	ID	II	←					IF	-	ID	←					IF	-	ID	←					IF	-	ID	←					IF	-	ID	←			
addi x18 x0 98																IF	ID	←					IF	-	ID	←					IF	-	ID	←					IF	-	ID	←					IF	-	ID	←			
sub x11 x19 x9																IF	ID	←							IF	←							IF	←							IF	←							IF	←			
addi x10 x8 0																	IF	←							IF	←							IF	←							IF	←							IF	←			
jal x1 -160 <minIndex>																	IF	←																																			
add x10 x10 x9																																																					
slli x10 x10 2																																																					
addi x15 x10 400																																																					

Figure 13: Indication of the instructions that are flushed due to taken branches

Since it consists of a dual-issue processor, the optimal CPI would be equal to 0,5 (that is, half of the optimal CPI of the processor discussed above). Again, in this case, the penalties associated to the flushings due to the taken branches and associated to the stalls provoke the augmentation of the CPI value. Also the programs suffers from lots of memory copies that are inherently sequential. If we would have had an even number of elements, loop unfolding could have put two `lw`s one after the other and the corresponding `sw`s after, performing double the speed in the copy cycle.

5 Caches

Aims: Understand the performance of caches.

In this section we are going to focus on data/instruction caching. However, the pipeline simulator is not really accurate for cache simulation (some results may be distorted). We start by analyzing the assembly code of the program, in parallel with its source C code, using the `objdump` tool in order to determine the number of distinct memory words accessed by the program.

Initially, it is determined the number of instructions fetched from the disassembled code available in Appendix B. Firstly in the `_start` function we have 5 instructions being executed (including an unconditional branch to main) and stopping at the `ecall` instruction (which halts the program).

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

Subsequently, `main` starts initialising the stack and the addresses in registers `a4` and `a5`. After, we have the loop used to copy the `input` data into the `buf` array. For that part we have 11 new instructions.

```
000100e0 <main>:
   100e0:       e5010113                addi    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                addi    a5,a5,420 # 111a4 <input>
   10100:       00410413                addi    s0,sp,4
   10104:       18c78613                addi    a2,a5,396
   10108:       00040713                mv      a4,s0
```

Next, we have 5 instruction that are executed 99 times (one for each input data).

```
   1010c:       0007a683                lw      a3,0(a5)
   10110:       00d72023                sw      a3,0(a4)
```

```
10114:        00478793                addi    a5,a5,4
10118:        00470713                addi    a4,a4,4
1011c:        fec798e3                bne     a5,a2,1010c <main+0x2c>
```

Then more 5 instructions are executed.

```
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
```

It then performs the sorting loop with 12 instructions, 98 times.

```
10134:        f61ff0ef                jal     ra,10094 <minIndex>
10138:        00950533                add     a0,a0,s1
1013c:        00251513                slli    a0,a0,0x2
10140:        19050793                addi    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                addi    s1,s1,1
1015c:        00440413                addi    s0,s0,4
10160:        fd2496e3                bne     s1,s2,1012c <main+0x4c>
```

Finally, 8 instructions for remove the data from stack and returns.

```
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                addi    sp,sp,432
10180:        00008067                ret
```

In the sorting loop, the `minIndex` function is called, which contains 19 instructions where 6 of them are executed Ω times (because of the loop). We have,

$$\Omega = \sum_{n=1}^{99} i = 4950$$

```
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            addi    a4,a4,1
100b0:      00468693            addi    a3,a3,4
100b4:      02e58063            beq     a1,a4,100d4 <minIndex+0x40>
100b8:      00251793            slli    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
```

So, for fetch instruction, we obtain:

$$\text{Instruction Fetches } = 5 + 11 + 99 \cdot (5 + 2) + 5 + 98 \cdot (12 + 13) + \Omega \cdot (6 + 2) + 8 = 42772$$

And for the data accesses, in the main function we have 2 per iteration of the copy loop at the beginning of `main`, 4 per iteration of the sort loop and 10 because of stacking/unstacking; 1 because of the `_start` function, 2 for each instruction in `minIndex` loop. So in total, we obtain:

$$\text{Data Fetches } = 99 \cdot 2 + 98 \cdot 4 + 10 + 1 + 2 \cdot \Omega = 10501$$

Then, we have selected the single-cycle processor in order to perform the simulation of the insertion sort ELF binary. Figure 14 shows the assumed processor.

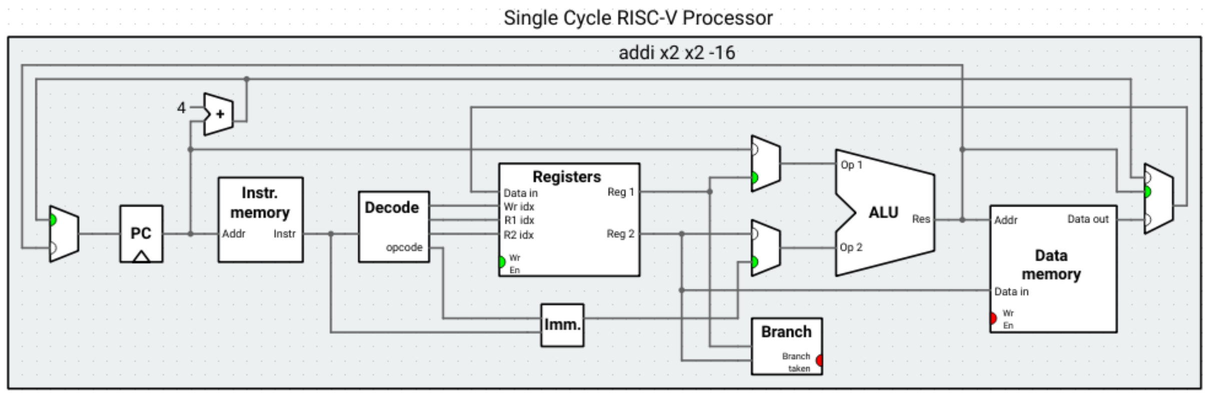
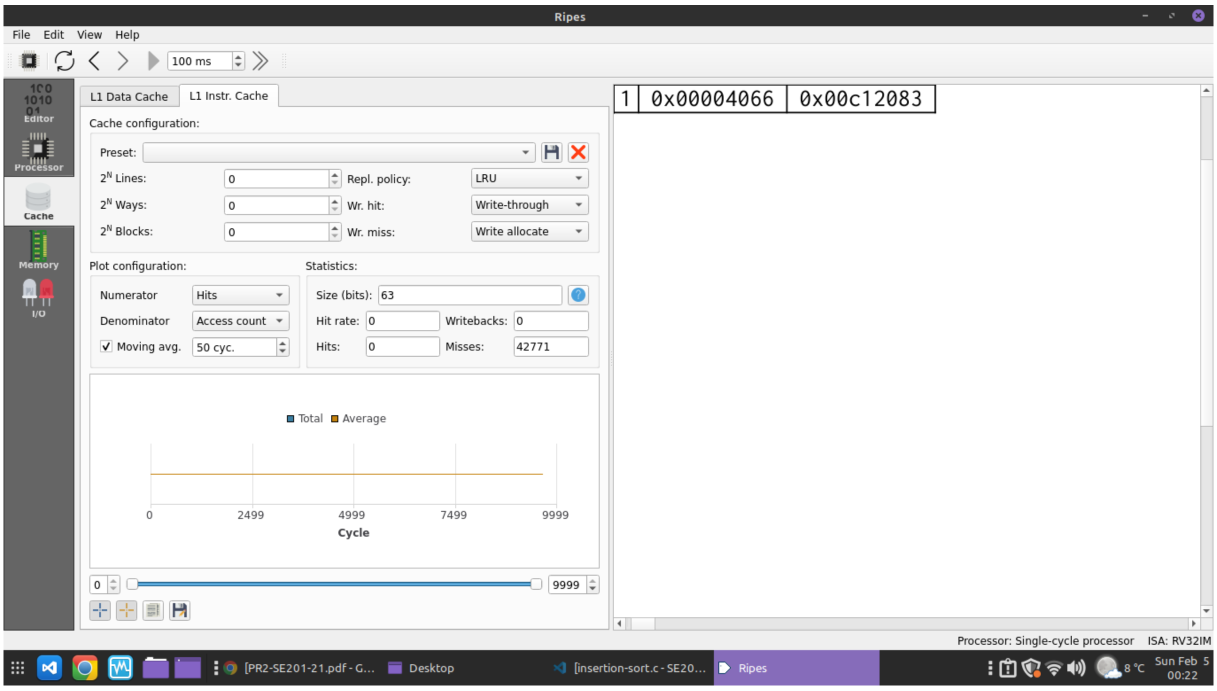


Figure 14: Single-cycle processor

Figures 15 and 16 show a way we have chosen to validate our previous results. Starting by configuring the cache as a write-through cache with write-allocate, we have set up caches as a single row and a single block, so the number of fetches will be the number of cache misses.



Cache configuration:

Preset:

2^N Lines:	0	Repl. policy:	LRU
2^N Ways:	0	Wr. hit:	Write-through
2^N Blocks:	0	Wr. miss:	Write allocate

Plot configuration:

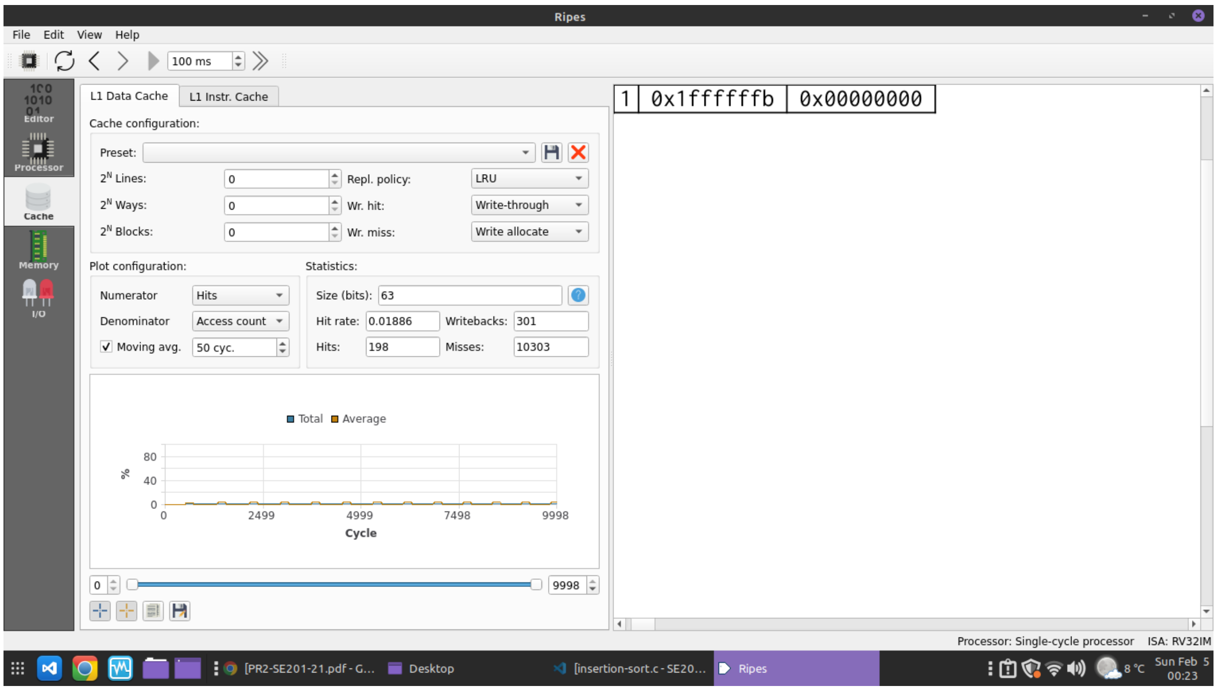
Numerator	Hits
Denominator	Access count
✓ Moving avg.	50 cyc.

Statistics:

Size (bits):	63		
Hit rate:	0	Writebacks:	0
Hits:	0	Misses:	42771

1	0x00004066	0x00c12083

Figure 15: Instruction Fetch



Cache configuration:

Preset:

2^N Lines:	0	Repl. policy:	LRU
2^N Ways:	0	Wr. hit:	Write-through
2^N Blocks:	0	Wr. miss:	Write allocate

Plot configuration:

Numerator	Hits
Denominator	Access count
✓ Moving avg.	50 cyc.

Statistics:

Size (bits):	63		
Hit rate:	0.01886	Writebacks:	301
Hits:	198	Misses:	10303

1	0x1ffffffb	0x00000000

Processor: Single-cycle processor ISA: RV32IM

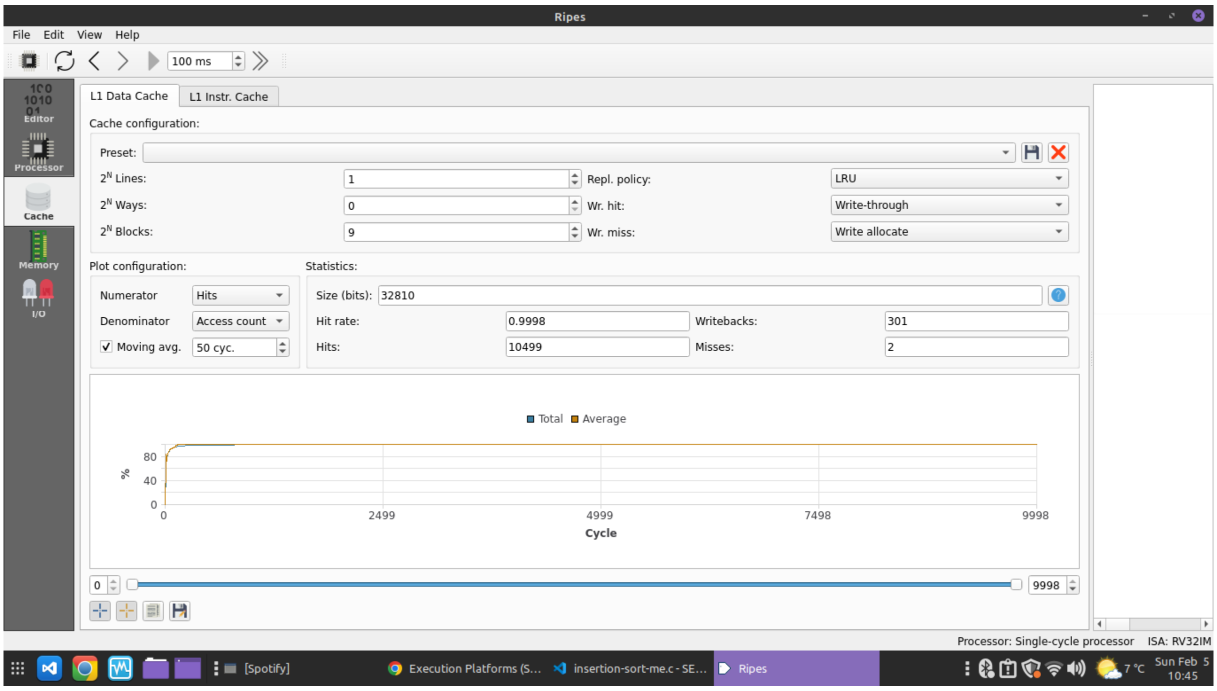
Figure 16: Data Fetch

From the simulator values, we supposed to have 10303 Data Fetches Misses, to which we should add the Hits (198), so 10501 and 42771 Instruction Fetches. That means that all the evaluations were good.

5.1 Cache optimization and experiments

Here we could start by setting up the biggest possible value of cache in order to minimize the misses, but it is not a good idea since we want to also minimize the size. Therefore, the idea is to set the maximum size of the data section, in order to have enough space to fit all the data inside the cache. By running the command `size insertion-sort.elf` we can see that the `data` section of our executable has 400 bytes, so we need to keep at least this amount of data in our cache.

Since we need to increase the cache in order to have less misses, we started by setting a single line and a block size of 2^9 (512) bytes for the Data Cache and also a single line, but a block size of 2^7 (128) bytes, for the Instruction Cache. Figure 17 and 18 show that, by doing so, we have a low number of misses because practically we loaded almost the whole data section in one set. It doesn't seem like a good idea, because actually we are still waiting a lot of time, but we just don't see it because we lost it all on one miss. In our program, we have a lot of misses that are compulsory because we are copying the 2 arrays. So all those misses can be avoided by a pre-fetching cache. The best scenario should be to have almost zero misses on the sorting algorithm, so to be sure that after copy finishes we have all the rest of the stack array in cache.



Cache configuration:

Preset:

2^N Lines: 1 Repl. policy: LRU

2^N Ways: 0 Wr. hit: Write-through

2^N Blocks: 9 Wr. miss: Write allocate

Plot configuration:

Numerator: Hits

Denominator: Access count

✓ Moving avg. 50 cyc.

Statistics:

Size (bits): 32810

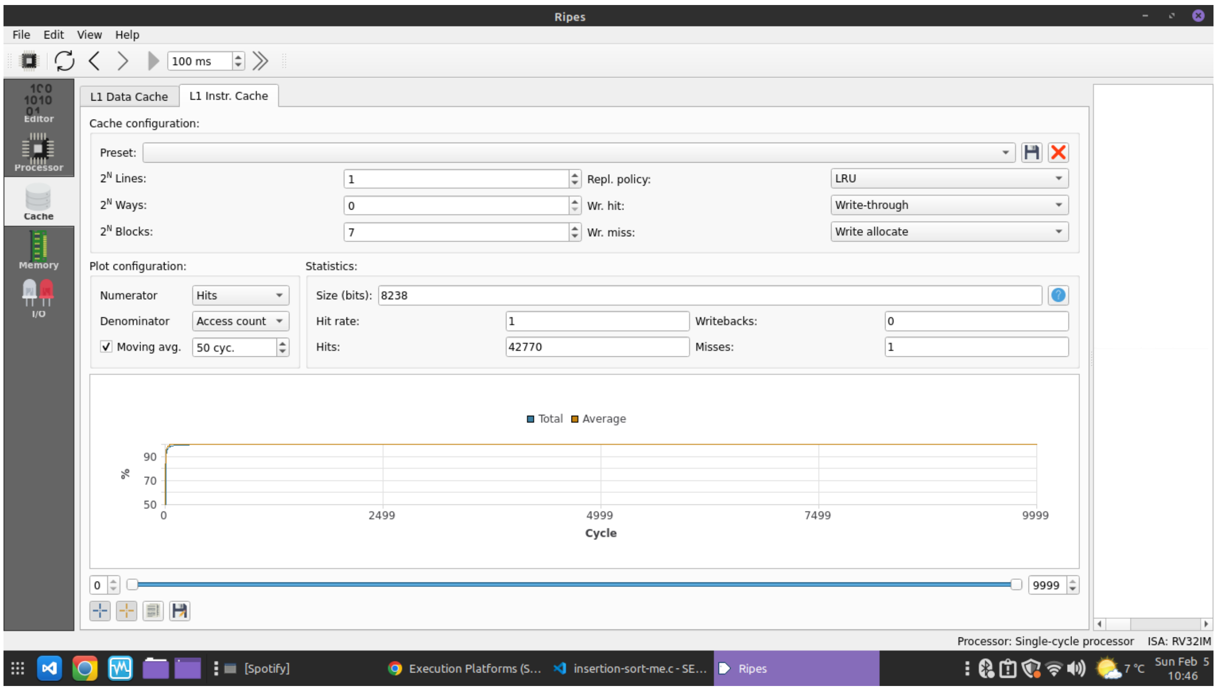
Hit rate: 0.9998 Writebacks: 301

Hits: 10499 Misses: 2

0 9998

Processor: Single-cycle processor ISA: RV32IM

Figure 17: Data Cache - First Optimisation



Cache configuration:

Preset:

2^N Lines: 1 Repl. policy: LRU

2^N Ways: 0 Wr. hit: Write-through

2^N Blocks: 7 Wr. miss: Write allocate

Plot configuration:

Numerator: Hits

Denominator: Access count

✓ Moving avg. 50 cyc.

Statistics:

Size (bits): 8238

Hit rate: 1 Writebacks: 0

Hits: 42770 Misses: 1

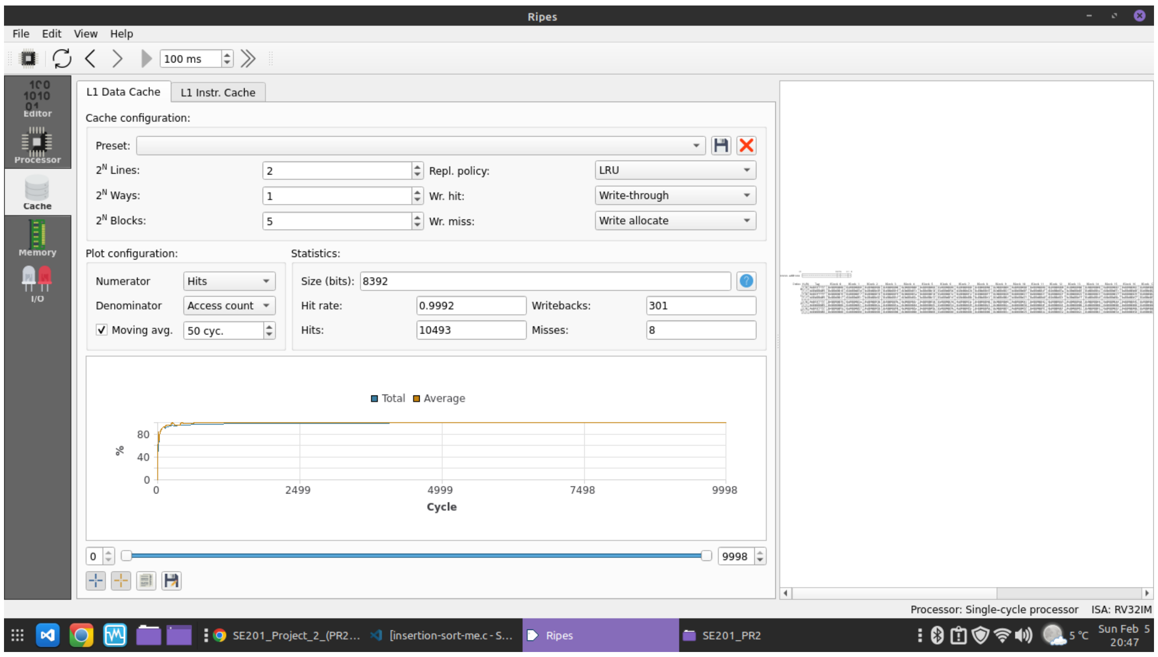
0 9999

Processor: Single-cycle processor ISA: RV32IM

Figure 18: Instruction Cache - First Optimisation

For instance we have arrays, which means that if we could make profit of the spatiality, it would be interesting. Since we have a lot of data that is consecutive in memory, if we don't get it come back. There is a trade-off between size and misses in that kind of project.

For the Data Cache, we have found out that it would be possible to put all data into a cache after only one access to the main memory if we have a single line and a block size of 128 bytes at least. It could result in only one cache miss, but the problem is that the cache wouldn't be explored as it was made for (fast and small). So we have defined a Data Cache with a similar capacity, but with an architecture divided in order to have the data distributed. Our block has only 32 bytes for the data burst and 1 way to avoid conflicts. Figure 19 shows the result for that optimization. It is possible to see that we have a number of cache misses down to single digit and also the total size of the caches in bits, it is not larger then a single-digit factor of the size of the accessed data/instructions, since we got a size of 8392 bits and 8 cache misses.



Cache configuration:

Preset:			
2^N Lines:	2	Repl. policy:	LRU
2^N Ways:	1	Wr. hit:	Write-through
2^N Blocks:	5	Wr. miss:	Write allocate

Plot configuration:

Numerator	Hits
Denominator	Access count
✓ Moving avg.	50 cyc.

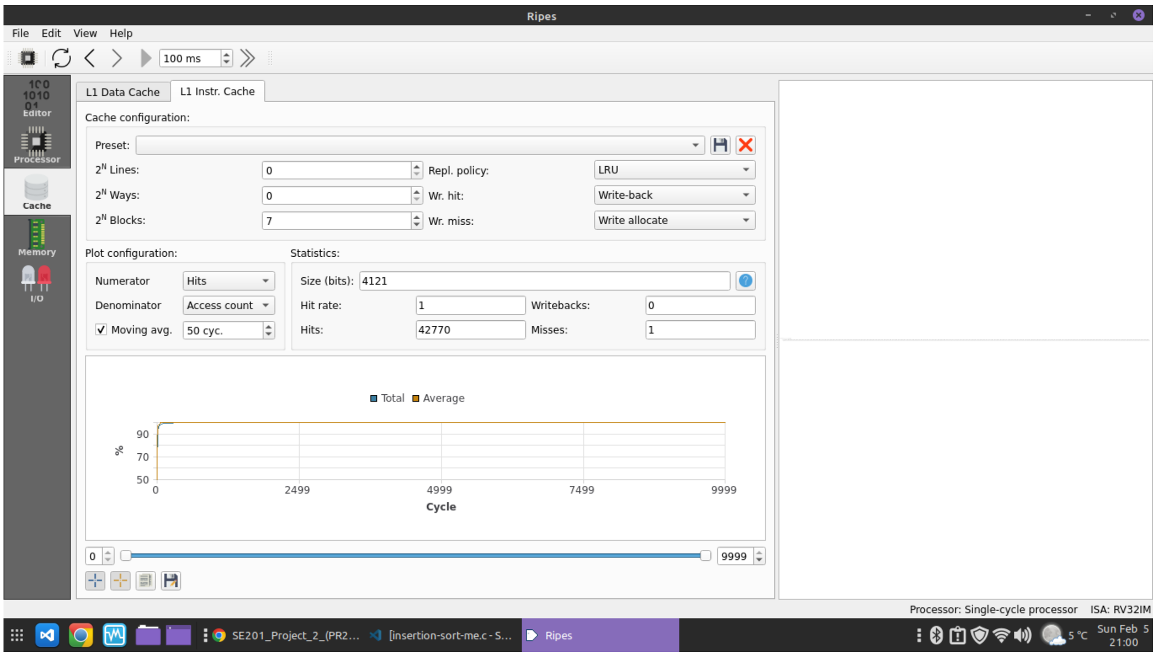
Statistics:

Size (bits):	8392		
Hit rate:	0.9992	Writebacks:	301
Hits:	10493	Misses:	8

Processor: Single-cycle processor ISA: RV32IM

Figure 19: Data Cache - Final Optimisation

Figure 20 shows the result for the Instruction Cache. In our previous optimization we got already a relative good outcome. Before we had 2 lines and a block size of 128 bytes. Hence, by decreasing only the number of lines to just a single line we could achieve the same result as before in terms of cache miss, but with half of the size. Now we have only one cache miss and Instruction Cache with a size of 4121 bits.



Cache configuration:

Preset:

2^N Lines: 0 | Repl. policy: LRU

2^N Ways: 0 | Wr. hit: Write-back

2^N Blocks: 7 | Wr. miss: Write allocate

Plot configuration:

Numerator: Hits

Denominator: Access count

✓ Moving avg.: 50 cyc.

Statistics:

Size (bits): 4121

Hit rate: 1 | Writebacks: 0

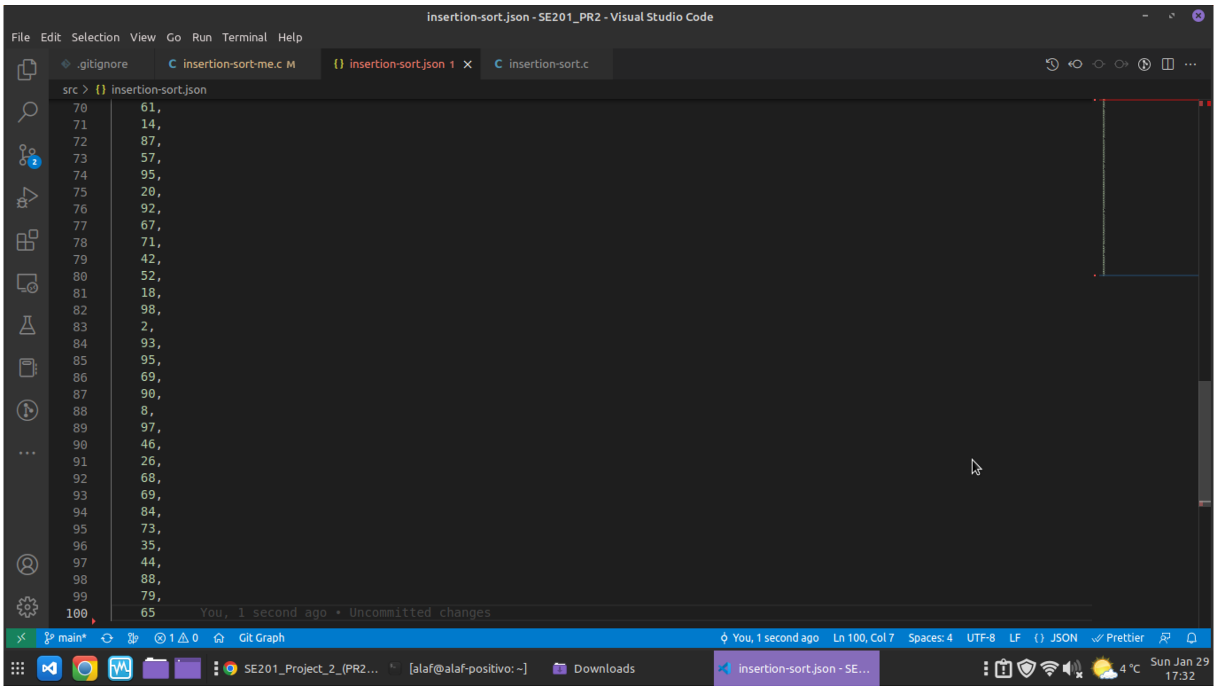
Hits: 42770 | Misses: 1

Processor: Single-cycle processor ISA: RV32IM

Figure 20: Instruction Cache - Final Optimisation

6 Additional comment about the provided C code

We found an unexpected result while using the provided C code in some experiments. When running the provided functions on our machines, it can be seen that the amount of input data was greater than the sorted data at the end of the execution. Basically we have realized by the following steps that the constant SIZE value should not be 99, but 100. Since we have 100 input values and the `buf` array is 99-width we are losing a value, more precisely, we are not including the value one 65 from our `input` array. Before coding anything to prove our suspicions, we decided to verify the `input` array. So by copying and pasting its values into a `JSON` file (the extension assumed was defined in order to use tome keyboard shortcuts to organize the data automatically) using Visual Studio Code, it was easy to see that we actually have 100 positions in this array, as showed in Figure 21.



```
70      61,
71      14,
72      87,
73      57,
74      95,
75      20,
76      92,
77      67,
78      71,
79      42,
80      52,
81      18,
82      98,
83      2,
84      93,
85      95,
86      69,
87      90,
88      8,
89      97,
90      46,
91      26,
92      68,
93      69,
94      84,
95      73,
96      35,
97      44,
98      88,
99      79,
100     65     You, 1 second ago • Uncommitted changes
```

Figure 21: Values of the input array

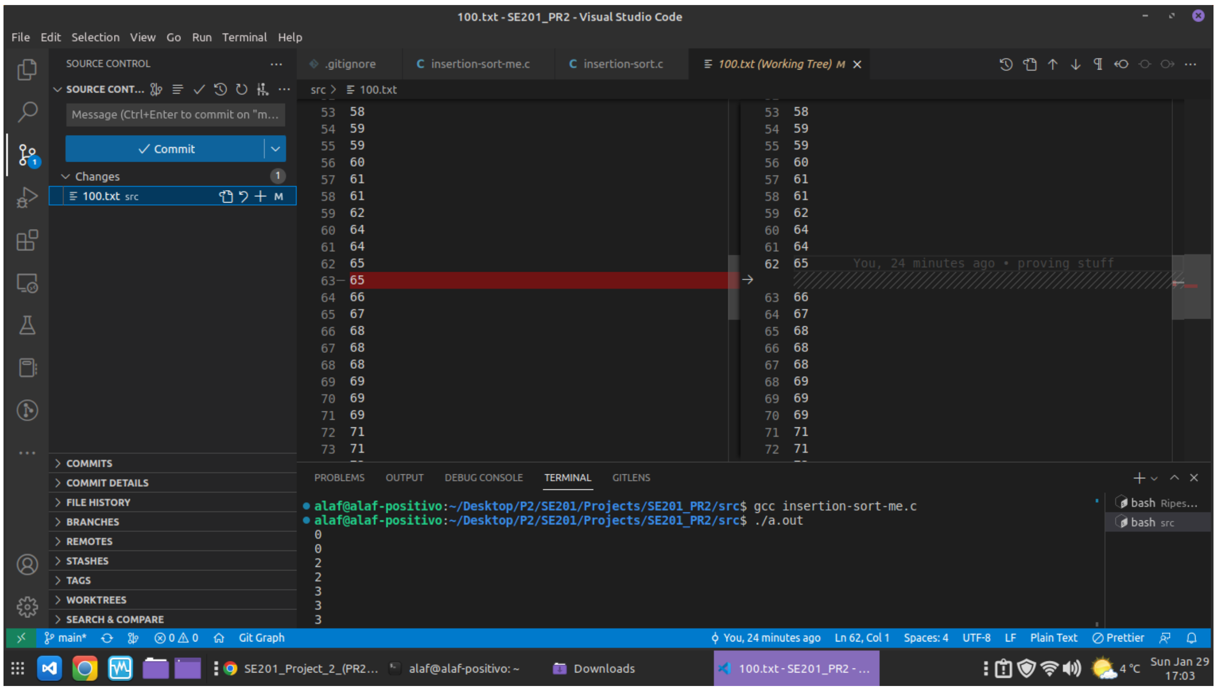
We started by adapting the provided code to our architecture and compiler on a Linux Mint 20.1 machine with gcc version 9.4.0 installed. We made a copy of the given C code file into a new file called `insertion-sort-me.c` and then we needed to include the standard input and output library by writing in the begin of the file `#include<stdio.h>`. After, we have deleted the `_start` because for the current purpose it didn't make sense to keep it in our code. We changed the name of our `main` function to `insertion` and we added to it a loop before the end of the function in order to print the values stored on the `buf` array. It is possible to see the obtained code in the following lines:

```
int insertion()
{
        int buf[SIZE];
        int i;

        for (i = 0; i < SIZE; i++)
        {
                buf[i] = input[i];
        }

        for (i = 0; i < SIZE - 1; i++)
        {
                int minIdx = i + minIndex(&buf[i], SIZE - i);
                int tmp = buf[minIdx];
                buf[minIdx] = buf[i];
                buf[i] = tmp;
        }

        for (i = 0; i < SIZE; i++)
        {
                printf("%d\n", buf[i]);
        }

    return buf[0];
}
```

Finally we wrote a new main function that only calls the previous one and then return 0 to the system as a way to sign an execution without problems.

```
int main()
{
  insertion();
  return 0;
}
```

With the current code which is available in Appendix E, we could run the command line `gcc insertion-sort-me.c` and then `./a.out` in order to compile and run the program. We did this for SIZE equal to 100 and then we saved the obtained result in a text file called `100.txt`, we committed the work up to the current step using git and then we did the same process for SIZE equal to 99 and we overwrote the result in the file same text file. Thus the git extension itself from Visual Studio Code automatically provided us the difference between the results obtained. Figure 22 shows the difference between the output obtained when `SIZE` is equal to 99 (there is only one 65 value) and when it is equal to 100 (65 appears twice, just like expected due to the `input` array).



```
58
59
59
60
61
61
62
64
64
65
65
66
67
68
68
68
69
69
69
71
71
```

```
58
59
59
60
61
61
62
64
64
65     You, 24 minutes ago • proving stuff
66
67
68
68
68
69
69
69
71
71
```

```
alaf@alaf-positivo:~/Desktop/P2/SE201/Projects/SE201_PR2/src$ gcc insertion-sort-me.c
alaf@alaf-positivo:~/Desktop/P2/SE201/Projects/SE201_PR2/src$ ./a.out
0
0
2
2
3
3
3
```

Figure 22: Output comparison for SIZE equal to 99 and SIZE equal to 100

References

[1] 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
OBJDUMP := $(PREFIX)objdump

CFLAGS:=-O -g -fno-pic -fno-inline -mcmodel=medlow -mabi=ilp32 -march=rv32im
-Wall -static -nostdinc -nostartfiles -nodefaultlibs -nostdlib

.PHONY: all clean

all: insertion-sort.elf

insertion-sort.elf: insertion-sort.c
$(CC) $(CFLAGS) -o $@ $<

clean:
rm -f insertion-sort.elf
```

B Assembly Code

```
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                addi    a4,a4,1
   100b0:       00468693                addi    a3,a3,4
   100b4:       02e58063                beq     a1,a4,100d4 <minIndex+0x40>
   100b8:       00251793                slli    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                addi    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                addi    a5,a5,420 # 111a4 <input>
   10100:       00410413                addi    s0,sp,4
   10104:       18c78613                addi    a2,a5,396
   10108:       00040713                mv      a4,s0
   1010c:       0007a683                lw      a3,0(a5)
   10110:       00d72023                sw      a3,0(a4)
   10114:       00478793                addi    a5,a5,4
   10118:       00470713                addi    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     ra,10094 <minIndex>
   10138:       00950533                add     a0,a0,s1
   1013c:       00251513                slli    a0,a0,0x2
   10140:       19050793                addi    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                addi    s1,s1,1
   1015c:       00440413                addi    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                addi    sp,sp,432
   10180:       00008067                ret

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

C Insertion Sort

```c
#define SIZE 99

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};

int minIndex(int *array, int n)
{
        int i,minIdx = 0;
        for(i = 0; i < n; i++)
        {
                if (array[i] < array[minIdx])
                {
                        minIdx = i;
                }
        }

        return minIdx;
}

int main()
{
        int buf[SIZE];
        int i;

        for(i = 0; i < SIZE; i++)
        {
                buf[i] = input[i];
        }

        for(i = 0; i < SIZE-1; i++)
        {
                int minIdx = i + minIndex(&buf[i], SIZE - i);
                int tmp = buf[minIdx];
                buf[minIdx] = buf[i];
                buf[i] = tmp;
        }

        return buf[0];
}

int _start()
{
  int x = main();
  asm volatile ("li a7, 10; ecall"); // exit system call
  return x;
}
```

D Insertion Sort With Counters

```
#include ”stdio.h”
#define SIZE 99

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};

int cycle_counter = 0;
int function_branch = 0;

int minIndex(int *array, int n)
{
        int i,minIdx = 0;
        for(i = 0; i < n; i++)
        {
                cycle_counter++;
                if (array[i] < array[minIdx])
                {
                        minIdx = i;
                }

        }
        function_branch++;
        return minIdx;
}

int main()
{
        int buf[SIZE];
        int i;

        for(i = 0; i < SIZE; i++)
        {
                cycle_counter++;
                buf[i] = input[i];
        }

        for(i = 0; i < SIZE-1; i++)
        {
                cycle_counter++;
                function_branch++;
                int minIdx = i + minIndex(&buf[i], SIZE - i);
                int tmp = buf[minIdx];
                buf[minIdx] = buf[i];
                buf[i] = tmp;
        }

        printf("Cycles: %d, function calls: %d\n", cycle_counter,function_b
        return buf[0];
}

/*int _start()
{
  int x = main();
  asm volatile ("li a7, 10; ecall"); // exit system call
  return x;
}*/
```

E Insertion Sort With Correct Size

```c
#include <stdio.h>
#define SIZE 100

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};

int minIndex(int *array, int n)
{
        int i, minIdx = 0;
        for (i = 0; i < n; i++)
        {
                if (array[i] < array[minIdx])
                {
                        minIdx = i;
                }
        }

        return minIdx;
}

int insertion()
{
        int buf[SIZE];
        int i;

        for (i = 0; i < SIZE; i++)
        {
                buf[i] = input[i];
        }

        for (i = 0; i < SIZE - 1; i++)
        {
                int minIdx = i + minIndex(&buf[i], SIZE - i);
```

```
                        int tmp = buf[minIdx];
                        buf[minIdx] = buf[i];
                        buf[i] = tmp;
        }

        for (i = 0; i < SIZE; i++)
        {
                printf("%d\n", buf[i]);
        }

        return buf[0];
}

int main()
{
        insertion();
        return 0;
}
```