Politecnico di Torino

Collegio di Elettronica, Telecomunicazioni e Fisica

Report for the course
Integrated Systems Architecture

Master degree in Electrical Engineering

Authors: Group 35

Pietro Fagnani, Marco Massetti, Pietro Montorsi

February 17, 2023

Contents

CHAPTER 1

Lab 3: RISC-V-lite

The aim of this lab is to design a RISC-V-lite processor capable of running a specific application. The activity is divided in two main parts, in a first part the goal is to develop a regular processor, in the second part the content of the instruction memory will be encrypted and the hardware of the machine will be changed in order to be able to decrypt the instructions before executing them.

1.1 Regular architecture

1.1.1 Architecture definition

The code to run on the RISC-V-lite processor has been obtained compiling the source code. A crt0 file and a linker script have been provided to the compiler to define the startup code and the memory organization of the machine.
The obtained object code is:

```
main:     file format elf32-littleriscv


Disassembly of section .init:

00400000 <_start>:
  400000:       1fc18197                auipc   gp,0x1fc18
  400004:       01c18193                addi    gp,gp,28 # 2001801c <__global_pointer$>
  400008:       7fbff117                auipc   sp,0x7fbff
  40000c:       ff410113                addi    sp,sp,-12 # 7fffeffc <__stack_top>
  400010:       00010433                add     s0,sp,zero
  400014:       008000ef                jal     ra,40001c <main>

00400018 <el>:
  400018:       0000006f                j       400018 <el>

Disassembly of section .text:

0040001c <main>:
  40001c:       10010537                lui     a0,0x10010
  400020:       ff010113                addi    sp,sp,-16
  400024:       00700593                li      a1,7
  400028:       00050513                mv      a0,a0
  40002c:       00112623                sw      ra,12(sp)
  400030:       01c000ef                jal     ra,40004c <minv>
  400034:       00c12083                lw      ra,12(sp)
```

```
  400038:       100107b7                lui     a5,0x10010
  40003c:       00a7ae23                sw      a0,28(a5) # 1001001c <_edata>
  400040:       00000513                li      a0,0
  400044:       01010113                addi    sp,sp,16
  400048:       00008067                ret

0040004c <minv>:
  40004c:       00052603                lw      a2,0(a0) # 10010000 <v>
  400050:       00100713                li      a4,1
  400054:       41f65793                srai    a5,a2,0x1f
  400058:       00c7c633                xor     a2,a5,a2
  40005c:       40f60633                sub     a2,a2,a5
  400060:       02b75863                ble     a1,a4,400090 <minv+0x44>
  400064:       00259593                slli    a1,a1,0x2
  400068:       00450713                addi    a4,a0,4
  40006c:       00b50533                add     a0,a0,a1
  400070:       00072783                lw      a5,0(a4)
  400074:       00470713                addi    a4,a4,4
  400078:       41f7d693                srai    a3,a5,0x1f
  40007c:       00f6c7b3                xor     a5,a3,a5
  400080:       40d787b3                sub     a5,a5,a3
  400084:       00c7d463                ble     a2,a5,40008c <minv+0x40>
  400088:       00078613                mv      a2,a5
  40008c:       fee512e3                bne     a0,a4,400070 <minv+0x24>
  400090:       00060513                mv      a0,a2
  400094:       00008067                ret

Disassembly of section .data:

10010000 <v>:
  10010000:     00000009
  10010004:     ffffffd2
  10010008:     00000015
  1001000c:     fffffffe
  10010010:     0000000e
  10010014:     0000001a
  10010018:     fffffffd
```

As required from the specifications the processor has been realized with the Harvard architecture, and with five pipeline stages, the architecture that has been designed is shown in figure 1.1.

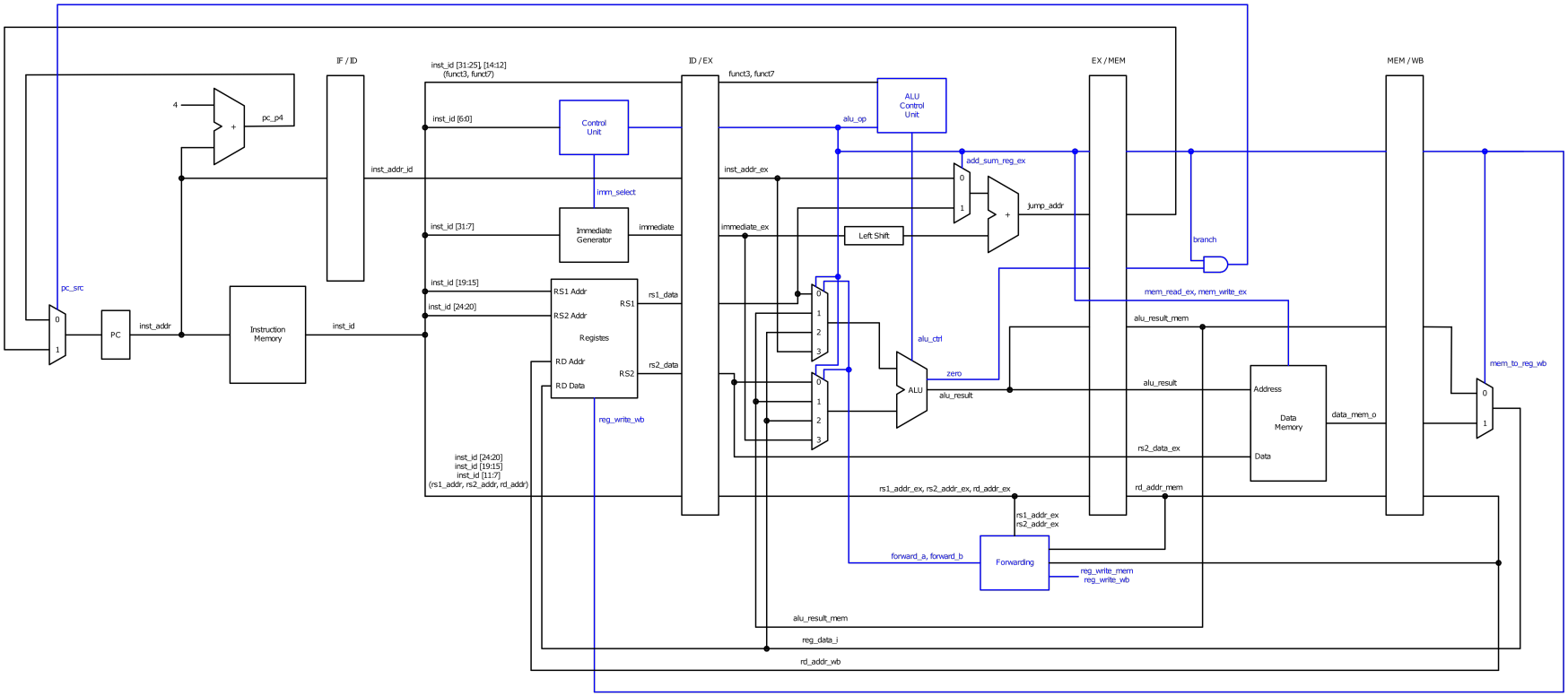

Figure 1.1: Processor architecture

Processor architecture

It is possible to more easily review the architecture in figure 1.1 studying one pipeline stage at a time.

- Instruction fetch (IF): The first stage is responsible for the reading of the instruction to be executed. The address of the instruction to read is stored in the program counter, this address is provided to the instruction memory to retrieve the corresponding instruction.
  At the same time the next address is computed by an adder, this address will be used only if the program is preceding sequentially, in case of jumps the next address is provided by another pipeline stage.
  It is possible to see that the fetched instruction does not pass trough the pipeline register, this is done to compensate the latency introduced by the memory, indeed the instruction address is read on the rising edge of the clock and the instruction is provided on the following falling edge. This does not cause any problems to the system but the maximum working frequency will be drastically reduced (this will be studied more in detail later).

- Instruction decode (ID): In the second stage the instruction is decoded by the control unit to generate the control signals for the components in the processor, and the data needed to perform the operation can be read from the register file. Moreover a component to select the correct bits that form the immediate values is present.

- Execute (EX): In this stage the instruction is executed, this can be the calculation of a mathematical operation or the checking of a logical expression. Since, in this processor, forwarding is used to reduce the number of pipeline stalls the inputs of the arithmetic logic unit are selected by two special multiplexers.

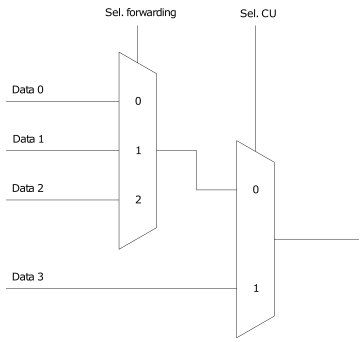

Figure 1.2: ALU multiplexers

The selection of the input operands depends from the control unit and from the forwarding unit, the CU controls if the data should be from the register file or should be an immediate (or the value of the program counter for the other operand), while the forwarding unit evaluates if the value can be read from the registers or it is necessary to forward it from the ”MEM” or ”WB” pipelining stages.
A second adder is used to calculate the new instruction address in case of jumps.

- Memory (MEM): In this stage the data memory can be accessed to read o store values. Also in this case the memory has a latency that needs to be compensated, this time the pipelining register before the RAM has been bypassed by all the signals in input to the memory.
  In this stage it is also detected if a jump operation should be performed, if the result is positive the new instruction address is loaded in the program counter and all the instructions loaded in the pipelining registers are discarded, the penalty introduced by every jump operation is equal to three clock cycle and it will be better commented in the following section.

- Write back (WB): In the last stage the result of the instruction is written in the register file, the control unit can chose if the data should arrive from the data memory or from the operation just executed.

Forwarding unit

The forwarding unit is a component in the processor pipeline that, when possible, resolves data hazards, i.e., situations where an instruction depends on the results of a previous instruction that have not yet been written to the register file. The forwarding unit monitors the instruction pipeline and bypasses the results of an instruction directly to dependent instructions instead of waiting for the results to be written to the register file. This helps to reduce the number of pipeline stalls and increase the processor's performance.
Bypass conditions are as follows:

- Execution hazard

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
    Forward_A_o = 10 //forwarding from EX/MEM register
```

- Memory hazard

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs1))
    Forward_A_o = 01 //forwarding from MEM/WB register
```

In this case are reported conditions concerning only one input of the ALU, the same applies to the second input.
In figure 1.2 is shown the input mux controlled by the forwarding unit.
With the default control ($Forward\_A\_o = 00$) the ALU is connected to the output from the register file.

Instruction memory's structure

Before analysing the whole structure it is possible to better observe how the memories have been realized. It has been requested to build the memories using only the module "sram_32_1024_freepdk45" (a SRAM with synchronous read and write with 32 bit width and 1024 cell length), since in the linker script is defined that the memories should have length 64 Kib (for the instruction memory) and 4000 Mib (for the data memory), more than one module has been used to build each memory.
For the instruction memory the following structure has been used:

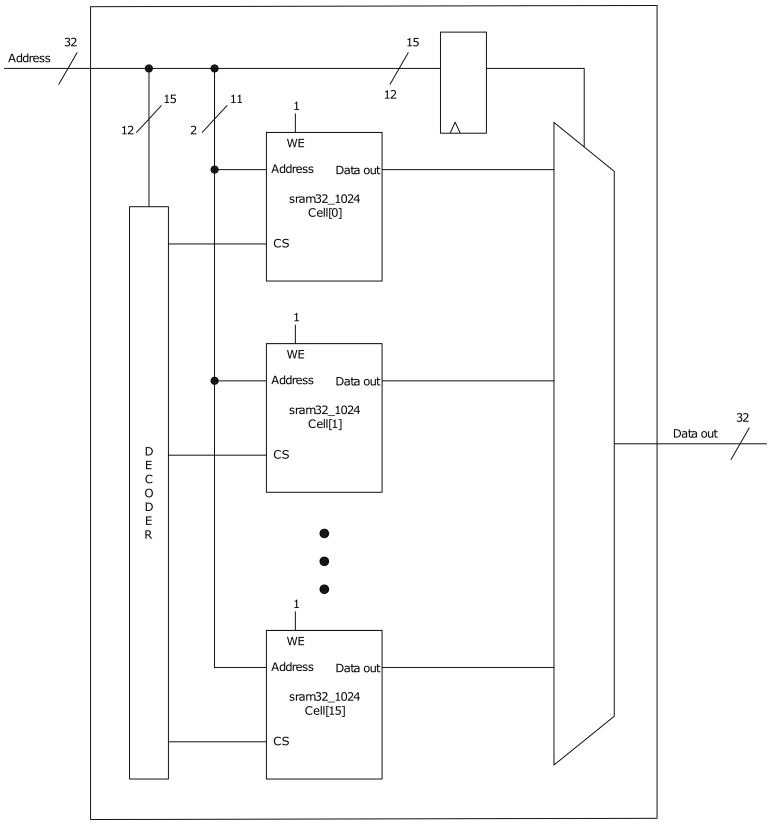

Figure 1.3: Instruction memory structure

Since every SRAM module has length 4 Kib ($length = 1024 \cdot 4\ B = 4096\ B$) 16 modules are required to match the required length.

The input address of each module has been connected together to the ten bits [11:2] of the instruction address (the two least significant its have been skipped since the address increases by 4 at a time, the two bits can be useful only of the SRAM is accessible byte by byte).

To select which module should be accessed a decoder on the four address bits [15:12] has been placed to select only one SRAM module at a time, the output of each module is connected to a multiplexer and also in this case the same four address bits are used to select the correct data line to connect to the output. It is important to notice that a register has been placed on the address bits that determines the SRAM module, this is required because there is a latency from when the read address is valid to when the output data is provided.

Finally the write enable signal of each module is forced to one since it is not required to write in the instruction memory.

Data memory's structure

The data memory structure is similar to the instruction memory one but in this case the only the two SRAM modules that the software needs to run have been used. This is because to realize the memory in its full size more than one million SRAM modules would be required and this would slow down significantly the simulation and synthesis processes. Therefore only two modules have been used and some logic checks which module should be selected comparing the address bits [31:12] and if a read or write operation is required (also in this case the two LSBs have been discarded for the same reason). Also for the data memory the register to store the previous accessed cell is required, its value is updated only if a read/write operation is required on an existing SRAM cell.

Reset signals

The different registers in the architecture do not use the same reset signal, this is due to the necessity of flushing the pipeline in the case of a jump instruction. Three different reset signal are used inside the circuit, all the registers have synchronous reset:

- For the program counter, register file, pipeline registers IF/ID and MEM/WB only the reset signal coming from outside the circuit is used. These registers do not need to be reset in case of jumps.
- The pipelining register EX/MEM can be reset both via the external reset or by the "pc_src" signal generated inside the processor, this signal is asserted every time that a jump is executed. In case of jumps this register needs to be reset since his content is derived from instructions that should be executed proceeding sequentially in the software.
- The ID/EX register is reset in the same conditions as the EX/MEM register but both in case of external reset or in case of reset due to jump the reset signal must be held low for one more clock cycle. This extended reset is needed since before having the correct values at it inputs the program counter should sample the new instruction address and the new instruction needs to be read from the memory, these two operations need two clock cycles.

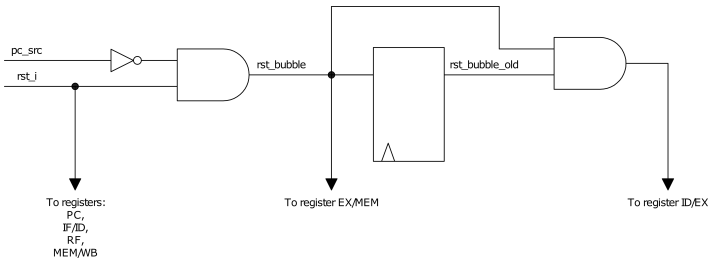

Figure 1.4: Distribution of reset signals

Registers

Thanks to the forwarding unit it is possible to resolve data hazards for operations that require the updated value of a register an operation or two earlier.
However, when the operation requires the updated value from an instruction executed three cycles earlier, the data is loaded into the register on the clock edge when it is requested. This results in an apparent extra pipe level.
Various implementations can be made to solve this problem, such as adding this case to the forwarding unit to perform the bypass. In this instance the choice that has been made is to make the registers sample the incoming data on the falling edge of the clock as the only component present on this path is a mux and therefore the total latency is well under half a period.

1.1.2 Architecture verification

The architecture has been verified via simulation. The instruction and data memory have been loaded in the testbench exploiting the SystemVerilog instruction ”readmemh”. The correctness of the execution has been verified via manual inspection of the waves generated by the simulator and via a textual snapshot of the content of the register file and the data memory at the end of the simulation; the reference data has been obtained using the ”RARS” simulator.

To begin the execution of the application the reset signal is forced low to initialize the register file content to the values defined in the linker script (stack pointer=7fffeffc, global pointer=10008000, all other registers=0) and to initialize the program counter (00400000 as defined in the linker script).

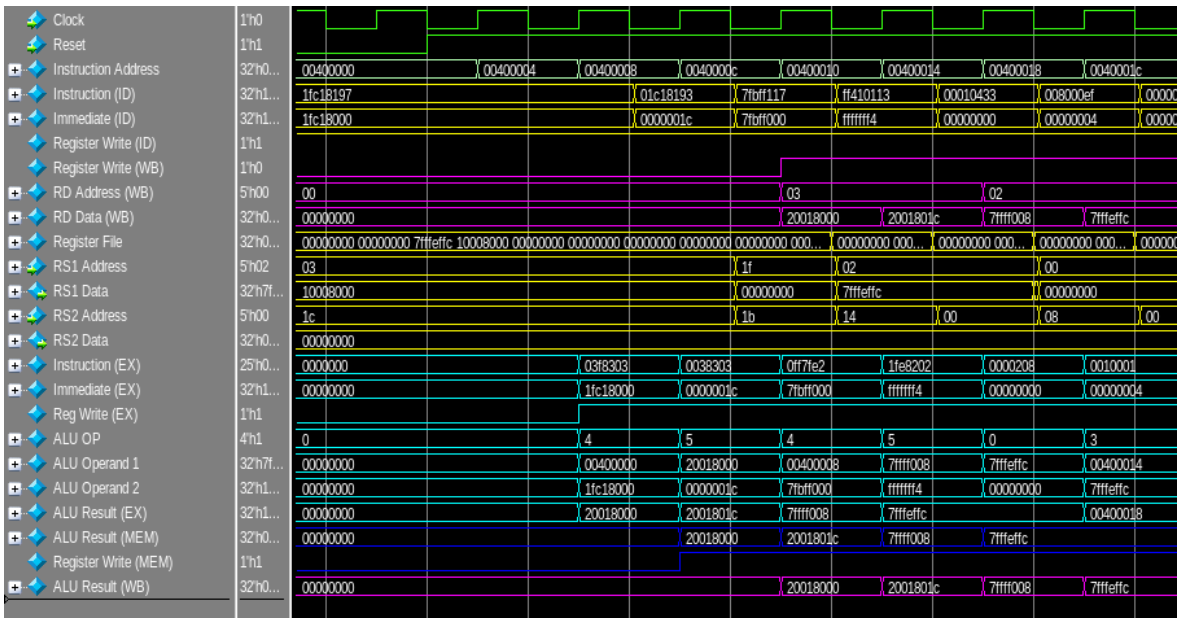

Figure 1.5: Beginning of the execution

If an instruction can cause a jump (instruction at address 00400014 in figure 1.6), the outcome of the jump condition will be known only in the memory pipeline stage, in the meantime the successive instructions will be fetched sequentially as in the normal execution. If the jump needs to be executed the fetched instructions will be discarded and the pipeline needs to be loaded again, this causes a penalty of 3 clock cycle at each jump.
In the image 1.6 it is possible to see how the branch target instruction (address 0040001c) is fetched after a delay of three cycles from the ”JAL” instruction, and how the return address is correctly saved in the register file.

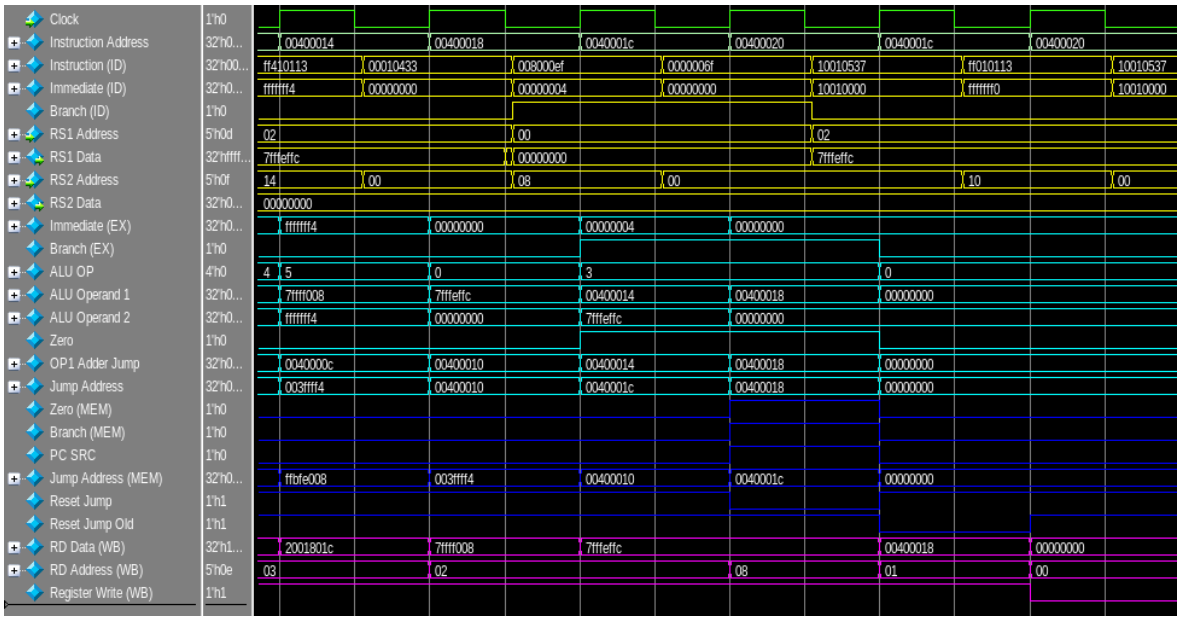

Figure 1.6: Execution of a jump

In figure 1.7 are shown a write and a read operation on the data memory. the instruction at address ”40002c” is a ”store word” instruction, therefore a write on the memory is required. The operation starts when the signal ”memory write” is asserted in the execution pipeline level, on the following rising edge of the clock the input data end the address are sampled by the SRAM module, on the following falling edge of the clock it is possible to see that the writing has been executed (event on the memory).

The following read operation (center of the figure) will not be concluded since a jump is executed (as it is possible to see from the ”Reset Jump” signal).

After the latency caused by the jump the operation at address ”40004c” is a ”load word”, in this case the read signal is asserted and in the first falling clock edge, after the address has been sampled, the output value is available on the data lines.

Since pipeline register ”EX/MEM” has been bypassed by the signal in input to the memory the latency introduced buy the SRAM has been compensated, this way the operations can be concluded without errors and the maximum frequency is not compromised since, even if the output of the memory is provided on the falling edge of the clock, the combinational path after the memory is very short.

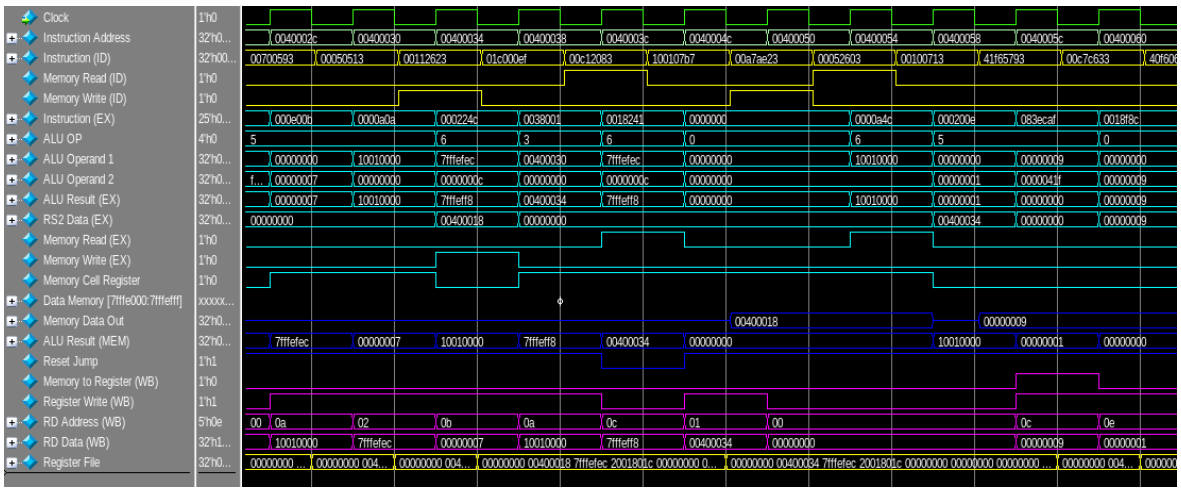

Figure 1.7: Read/Write operations on data memory

As shown in figure 1.8 the total time required to execute the *min-v* application is 2.36 ns that, with a clock period of 20 ps, is equal to 118 clock cycles. The execution is composed by a total of 73 operations, and during the execution 15 jumps are performed; knowing these values it is possible to verify that the number of cycles required to execute the applicaion is congruent with what has been measured in the simulation ($73 + (15 \cdot 3) = 118$).

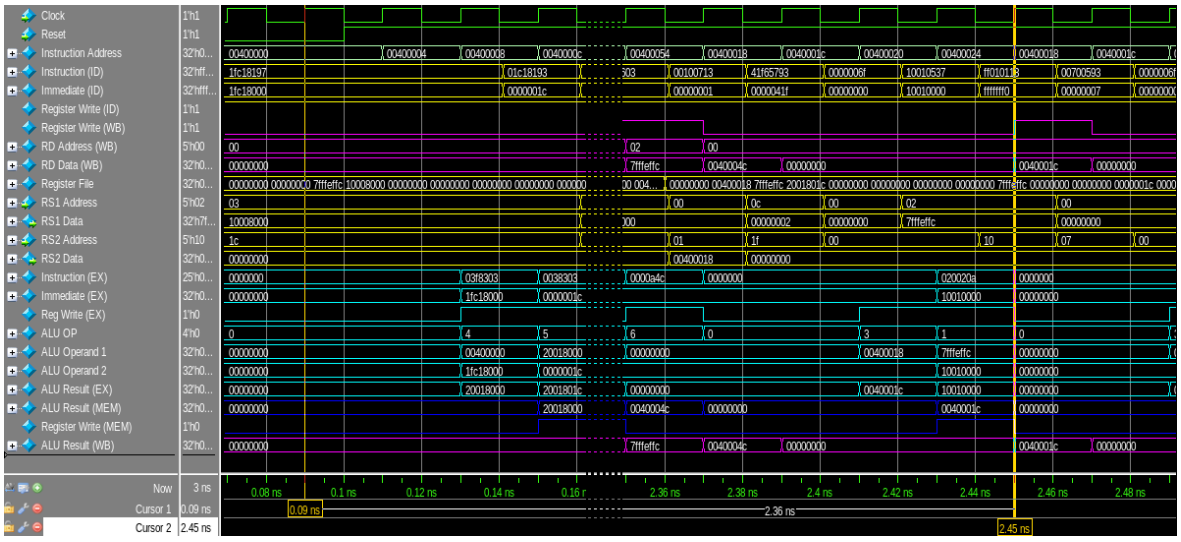

Figure 1.8: End of the execution

1.1.3 Architecture - logic synthesis results

The maximum frequency achieved by the architecture is 184.5 MHz with a total area of about 0.91 mm^2

Frequency

The frequency is limited by the path that starts from the output of the instruction memory, selects the address of the register file and ends in the register ”ID/EX”, this path is short compared to others since the time required to complete it is 2.71 ns, but since the output of the SRAM is given on the falling edge of the clock while the following register samples on the rising edge only half clock period is available to propagate the data.

```
clock MY_CLK (rise edge)                                 5.42       5.42
clock network delay (ideal)                              0.00       5.42
clock uncertainty                                       -0.01       5.41
ID_EX/rs1_data_o_reg[30]/CK (DFF_X1)                     0.00       5.41 r
library setup time                                      -0.03       5.38
data required time                                                  5.38
--------------------------------------------------------------------------
data required time                                                  5.38
data arrival time                                                  -5.38
--------------------------------------------------------------------------
slack (MET)                                                         0.00
```

Figure 1.9: Timing report

A possibility to reduce the critical path is to apply pipelining at the output the instruction memory, this increases the latency to execute each operation and the penalty when executing a jump, however with this modification the theoretical maximum frequency is 273.2 MHz since the output delay of the SRAM is 1.83 ns.

If a memory with a lower delay was used the critical path would become the one starting from the ”EX/MEM” register, passes trough the forwarding unit, the multiplexer, the ALU and ends in the data memory; in that case the maximum frequency would be equal to 637 MHz.

Area

Almost all the area of the circuit is occupied by the instruction memory, every ”sram_32_1024_freepdk45” module occupies 49970 μm^2 and to form the two memories 18 modules are used.

If all the SRAM is not considered the area occupied by the processor is 14659 μm^2, in this case the majority if the space is occupied by the register file (62.3% of the area), then 6.7% by the ALU and 6.4% by the register ”ID/EX” (165 bit register). The area overhead caused by the pipelining (sum of the area of the four pipelining registers) is 1945.5 μm^2, that is 13.3% of the processor’s area.

```
Number of ports:                        2535
Number of nets:                        12320
Number of cells:                        9161
Number of combinational cells:          7702
Number of sequential cells:             1403
Number of macros/black boxes:             18
Number of buf/inv:                      1879
Number of references:                     25

Combinational area:              8309.840032
Buf/Inv area:                    1375.485976
Noncombinational area:           6349.153770
Macro/Black Box area:          899476.312500
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:               914135.306303
Total area:                 undefined
```

Figure 1.10: Area of the device

Hierarchical cell	Global cell area Absolute Total	Percent Total	Local cell area Combi-national	Noncombi-national	Black-boxes	Design
CPU	914135.3063	100.0	8.2460	4.5220	0.0000	CPU
ALU	979.6780	0.1	589.7220	0.0000	0.0000	ALU
ALU/add_36	134.5960	0.0	134.5960	0.0000	0.0000	ALU_DW01_add_0
ALU/r375	103.4740	0.0	103.4740	0.0000	0.0000	ALU_DW01_cmp6_0
ALU/sub_36	151.8860	0.0	151.8860	0.0000	0.0000	ALU_DW01_sub_0
ALU_CU	24.4720	0.0	24.4720	0.0000	0.0000	ALU_CU
Adder_PC	134.5960	0.0	0.0000	0.0000	0.0000	Adder_0
Adder_PC/add_8	134.5960	0.0	134.5960	0.0000	0.0000	Adder_0_DW01_add_0_DW01_add_2
Adder_sum	134.5960	0.0	0.0000	0.0000	0.0000	Adder_1
Adder_sum/add_8	134.5960	0.0	134.5960	0.0000	0.0000	Adder_1_DW01_add_0_DW01_add_1
CU	45.7520	0.0	45.7520	0.0000	0.0000	CU
DATA_MEM	100087.0485	10.9	140.7140	4.5220	99941.8125	RAM
EX_MEM	415.7580	0.0	85.6520	330.1060	0.0000	EX_MEM
Forwarding_Unit	51.6040	0.0	51.6040	0.0000	0.0000	Forwarding_Unit
ID_EX	944.0340	0.1	193.1160	750.9180	0.0000	ID_EX
IF_ID	181.9440	0.0	37.2400	144.7040	0.0000	IF_ID
INST_MEM	800851.4660	87.6	1298.8780	18.0880	799534.5000	ROM
Imm_Gen	98.1540	0.0	98.1540	0.0000	0.0000	Imm_Gen
MEM_WB	403.7880	0.0	82.7260	321.0620	0.0000	MEM_WB
MUX32_PC	64.6380	0.0	64.6380	0.0000	0.0000	MUX_32_0
MUX32_alu_imm	127.6800	0.0	127.6800	0.0000	0.0000	MUX_32_4to1_1
MUX32_alu_pc	127.6800	0.0	127.6800	0.0000	0.0000	MUX_32_4to1_0
MUX32_mem	65.4360	0.0	65.4360	0.0000	0.0000	MUX_32_1
MUX32_sum	65.4360	0.0	65.4360	0.0000	0.0000	MUX_32_2
PC	181.4120	0.0	36.7080	144.7040	0.0000	PC
Registers	9137.3659	1.0	4506.8380	4630.5278	0.0000	Registers
Total			8309.8400	6349.1538	899476.3125	

Figure 1.11: Area of each module

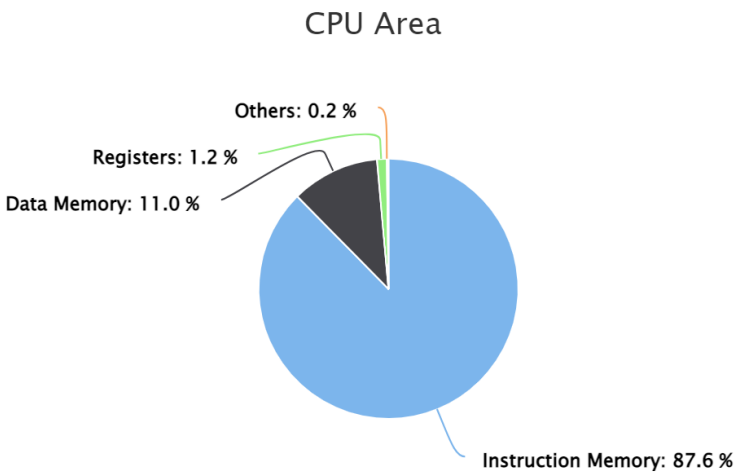

Figure 1.12: Pie Chart of the area

1.2 Advanced architecture

1.2.1 Advanced architecture definition

The encryption of the binary code is accomplished through the use of the source file $crypt.c$. This process involves the generation of a random 32-bit key for every four sequential instructions, followed by a bitwise XOR operation to produce the encrypted code.
The encrypted instructions to run on the RISC-V-lite processor and the sequence of keys are:

Address	Instruction	Key
400000	5ce3fb1f	43227a88
400004	42e3fb1b	
400008	3c9d8b9f	
40000c	bc637b9b	
400010	380c1bbc	380d1f8f
400014	388d1f60	
400018	380d1fe0	
40001c	280c1ab8	
400020	f7feb5ea	08ffb4f9
400024	088fb16a	
400028	08fab1ea	
40002c	08ee92da	
400030	16f294e3	1732940c
400034	17f3b48f	
400038	073393bb	
40003c	17953a2f	
400040	1571db90	1571de83
400044	1470df90	
400048	15715ee4	
40004c	1574f880	
400050	59f2c362	59e2c471
400054	181493e2	
400058	59250242	
40005c	1914c242	
400060	0f5e53cd	0de90bae
400064	0dcc9e3d	
400068	0dac0cbd	
40006c	0d5c0e9d	
400070	6625c18c	6622e60f
400074	6665e11c	
400078	27d5309c	
40007c	66d421bc	
400080	144636ea	5491b159
400084	5456653a	
400088	5496374a	
40008c	aa74a3ba	
400090	3729dbbb	372fdea8
400094	372f5ecf	

The decryption management architecture, depicted in figure 1.13, is the sole modification from the prior version. The Instruction Memory still stores the instruction code, however, an additional SRAM has been incorporated for the Key Memory as only a limited number of keys are required.
The instruction address serves as the address for both memories, with the difference that four LSBs are ignored for the key memory (since the instruction address increases by four at a time and for four consecutive instruction the same key is used). Finally, a bit-wise XOR is utilized to obtain the decrypted instruction.

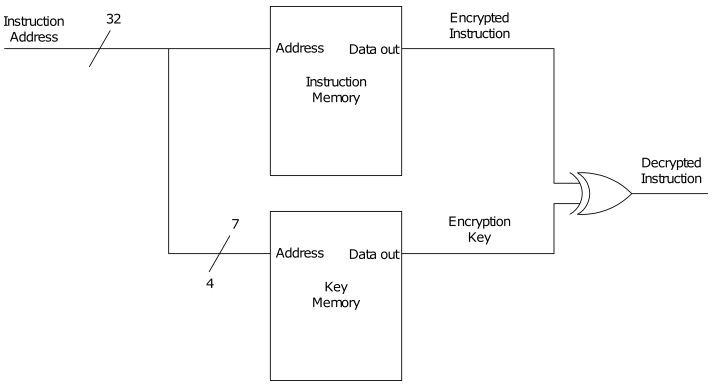

Figure 1.13: Decrypting of the instructions

1.2.2 Advanced architecture verification

The architecture has, again, been verified via simulation, as shown by the following figures. In particular, figure 1.15 shows that the total number of clock cycles to execute the encrypted code is 118. The number of cycles is the same as the original architecture since the introduction of the decryption of the instructions does not introduce additional delays.

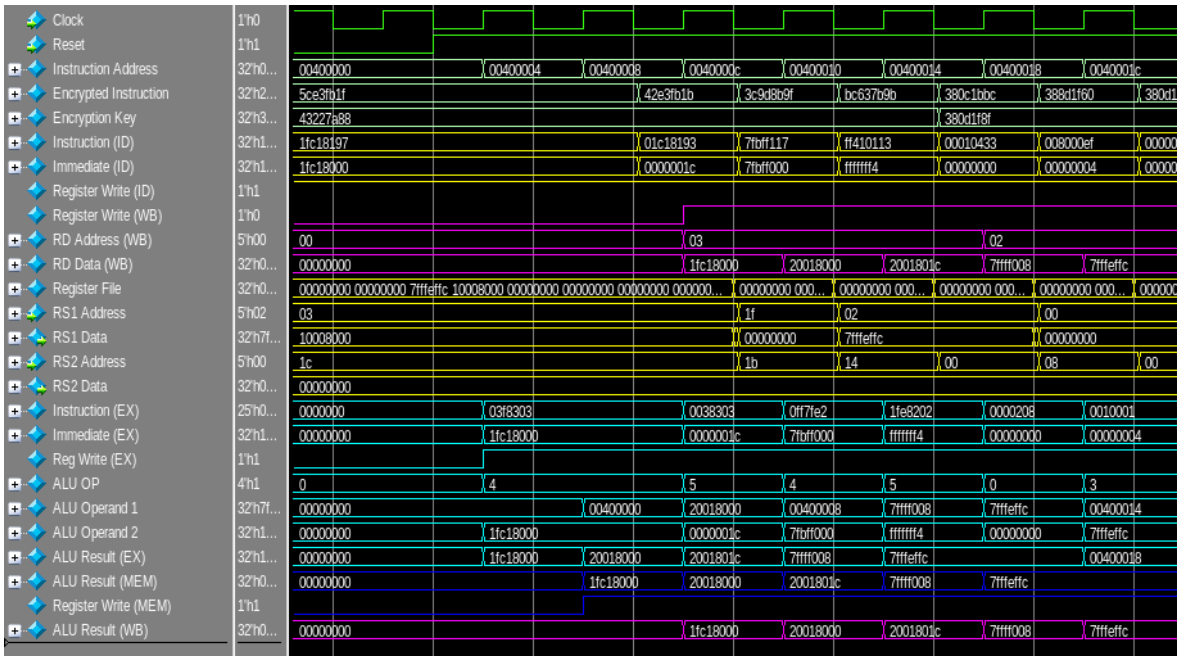

Figure 1.14: Beginning of the execution

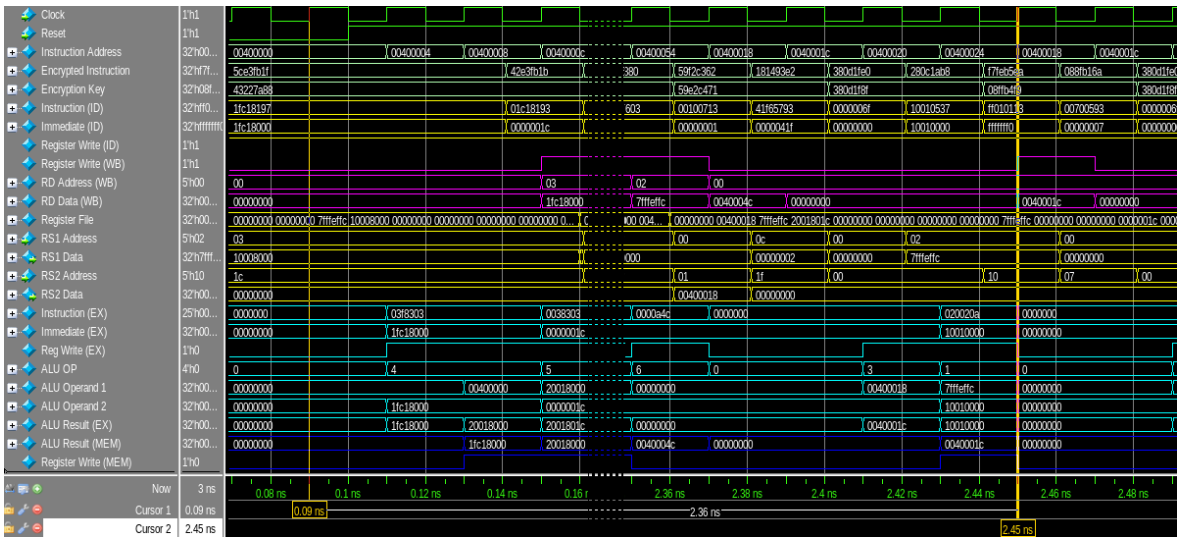

Figure 1.15: End of the execution

1.2.3 Advanced architecture - logic synthesis results

The maximum frequency achieved by the advanced architecture is 172.9 MHz with a total area equal to 0.96 mm^2. Due to the fact that an XOR logic operation is added in the critical path from Instruction Memory to ID/EX register a worst frequency is achieved. The area instead is increased by a value equal to one more SRAM used to store the keys for the encryption. Indeed, the following figures show slack met and area breakdown.

```
clock MY_CLK (rise edge)                                 5.58       5.58
clock network delay (ideal)                              0.00       5.58
clock uncertainty                                       -0.01       5.57
ID_EX/rs2_data_o_reg[20]/CK (DFF_X1)                     0.00       5.57 r
library setup time                                      -0.04       5.53
data required time                                                  5.53
--------------------------------------------------------------------------
data required time                                                  5.53
data arrival time                                                  -5.53
--------------------------------------------------------------------------
slack (MET)                                                         0.00
```

Figure 1.16: Timing report

```
Number of ports:                         2572
Number of nets:                         12201
Number of cells:                         8975
Number of combinational cells:           7514
Number of sequential cells:              1403
Number of macros/black boxes:              19
Number of buf/inv:                       1929
Number of references:                      28

Combinational area:               8201.578035
Buf/Inv area:                     1445.975974
Noncombinational area:            6346.493770
Macro/Black Box area:           949447.218750
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                963995.290555
Total area:                 undefined
```

Figure 1.17: Area of the device

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combi-national	Local cell area: Noncombi-national	Local cell area: Black-boxes	Design
CPU	963995.2906	100.0	89.9080	4.5220	0.0000	CPU
ALU	979.6780	0.1	589.7220	0.0000	0.0000	ALU
ALU/add_36	134.5960	0.0	134.5960	0.0000	0.0000	ALU_DW01_add_0
ALU/r375	103.4740	0.0	103.4740	0.0000	0.0000	ALU_DW01_cmp6_0
ALU/sub_36	151.8860	0.0	151.8860	0.0000	0.0000	ALU_DW01_sub_0
ALU_CU	24.4720	0.0	24.4720	0.0000	0.0000	ALU_CU
Adder_PC	134.5960	0.0	0.0000	0.0000	0.0000	Adder_0
Adder_PC/add_8	134.5960	0.0	134.5960	0.0000	0.0000	Adder_0_DW01_add_0_DW01_add_2
Adder_sum	134.5960	0.0	0.0000	0.0000	0.0000	Adder_1
Adder_sum/add_8	134.5960	0.0	134.5960	0.0000	0.0000	Adder_1_DW01_add_0_DW01_add_1
CU	48.6780	0.0	48.6780	0.0000	0.0000	CU
DATA_MEM	100087.0485	10.4	140.7140	4.5220	99941.8125	RAM
EX_MEM	415.7580	0.0	85.6520	330.1060	0.0000	EX_MEM
Forwarding_Unit	51.6040	0.0	51.6040	0.0000	0.0000	Forwarding_Unit
ID_EX	939.2460	0.1	193.1160	746.1300	0.0000	ID_EX
IF_ID	181.9440	0.0	37.2400	144.7040	0.0000	IF_ID
INST_MEM	800637.0700	83.1	1084.4820	18.0880	799534.5000	ROM
Imm_Gen	91.2380	0.0	91.2380	0.0000	0.0000	Imm_Gen
K_MEM	49974.8962	5.2	3.9900	0.0000	49970.9062	ROM_key
MEM_WB	403.7880	0.0	82.7260	321.0620	0.0000	MEM_WB
MUX32_PC	64.6380	0.0	64.6380	0.0000	0.0000	MUX_32_0
MUX32_alu_imm	127.6800	0.0	127.6800	0.0000	0.0000	MUX_32_4to1_1
MUX32_alu_pc	127.6800	0.0	127.6800	0.0000	0.0000	MUX_32_4to1_0
MUX32_mem	65.4360	0.0	65.4360	0.0000	0.0000	MUX_32_1
MUX32_sum	65.4360	0.0	65.4360	0.0000	0.0000	MUX_32_2
PC	183.5400	0.0	36.7080	146.8320	0.0000	PC
Registers	9161.8379	1.0	4531.3100	4630.5278	0.0000	Registers
Total			8201.5780	6346.4938	949447.2188	

Figure 1.18: Area of each module

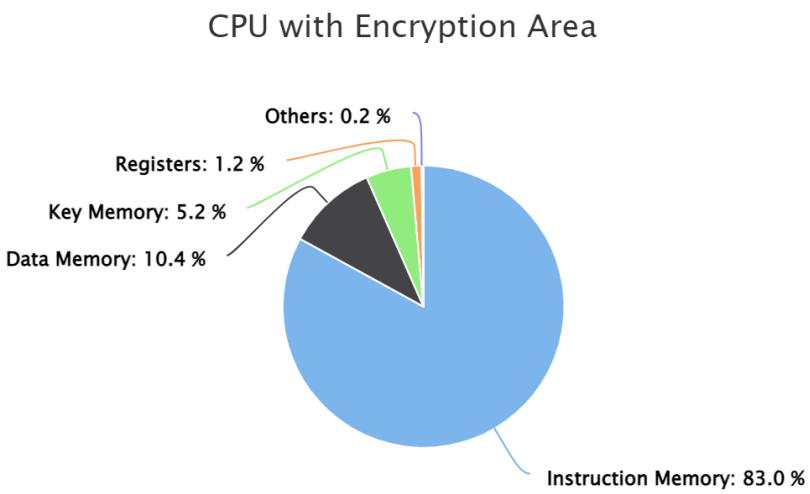

Figure 1.19: Pie Chart of the area

1.3 Script for verification

As explained before, the verification of the architecture was carried out manually. However, for quicker verification of the encrypted circuit and various synthesized circuits, the script $Launch.sh$ can be run, which sequentially executes the various simulations and synthesis.

The testbenches, thanks to the $\$fdisplay()$ command, write in different files the data contained in the Data Memory at the end of the simulation and the value of the program counter at each clock cycle, finally the script verifies that these files are equal for every simulation and for the two versions of the circuit, in the case of differences it prints an error.

For the printing of the content of the register file an additional step is required. In the SystemVerilog description the RF is defined as an array of bits, but, in the netlist generated by Synopsys it is split in 1024 different signals (one for every bit). Since it is not possible to print their values directly with the $\$fdisplay()$ command to resolve this issue the signals are manually rearranged into an array before being printed.