Politecnico di Torino

02GQCOQ - INTEGRATED SYSTEMS ARCHITECTURE

A.A. 2020/2021

PROF. MAURIZIO MARTINA
PROF. GUIDO MASERA

LAB3

Design of a RISC-V-lite processor

Group 18

---

https://github.com/wackozz/isa/tree/main/lab3

Federica BONGO	s292395
Stefano RIZZELLO	s288013
Francesco VACCA	s279895

Contents

1 Introduction and specification

The goal of the laboratory is to design a RISC-V lite processor, with 5 pipeline stages and 32-bit fixed width instruction set. The processor supports a subset of the RV32I Base Integer instruction set v.2.1, which is reported in Table 1. A detailed explanation of the various instructions at user level is provided in the RISC-V Instruction Set Manual, Volume I.

Binary format						Name	Type
imm[31:12]				rd	0110111	`lui`	[U-type]
imm[31:12]				rd	0010111	`auipc`	[U-type]
imm[20|10 : 1|11|19 : 12]				rd	1101111	`jal`	[J-type]
imm[11:0]		rs1	010	rd	0000011	`jw`	[I-type]
imm[11:0]		rs1	000	rd	0010011	`addi`	[I-type]
imm[11:0]		rs1	111	rd	0010011	`andi`	[I-type]
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	`sw`	[S-type]
imm[12|10 : 5]	rs2	rs1	000	imm[4:1|11]	1100011	`beq`	[B-type]
0100000	shamt	rs1	101	rd	0010011	`srai`	[R-type]
0000000	rs2	rs1	000	rd	0110011	`add`	[R-type]
0000000	rs2	rs1	010	rd	0110011	`slt`	[R-type]
0000000	rs2	rs1	100	rd	0110011	`xor`	[R-type]

Table 1: Reduced RV32I instruction set adopted for the RISC-V lite

The machine is based on the Harvard architecture, so two separate memories are needed for instructions and data in order for the processor to operate. From a systemic standpoint, the developed hardware is able to:

- Perform basic arithmetic and logic operations among the content of its registers;
- Load and store information from/to the registers to/from the data memory;
- Perform conditional and unconditional jumps;
- Detect and handle data dependencies (hazards) in the instruction flow.

Where possible, the design optimization effort was focused to reduce at minimum the delay, at the expenses of higher area allocation and power consumption.

2 RTL Design

A diagram for the complete CPU datapath, detailed with controls and forwarding paths is reported in Figure 1.

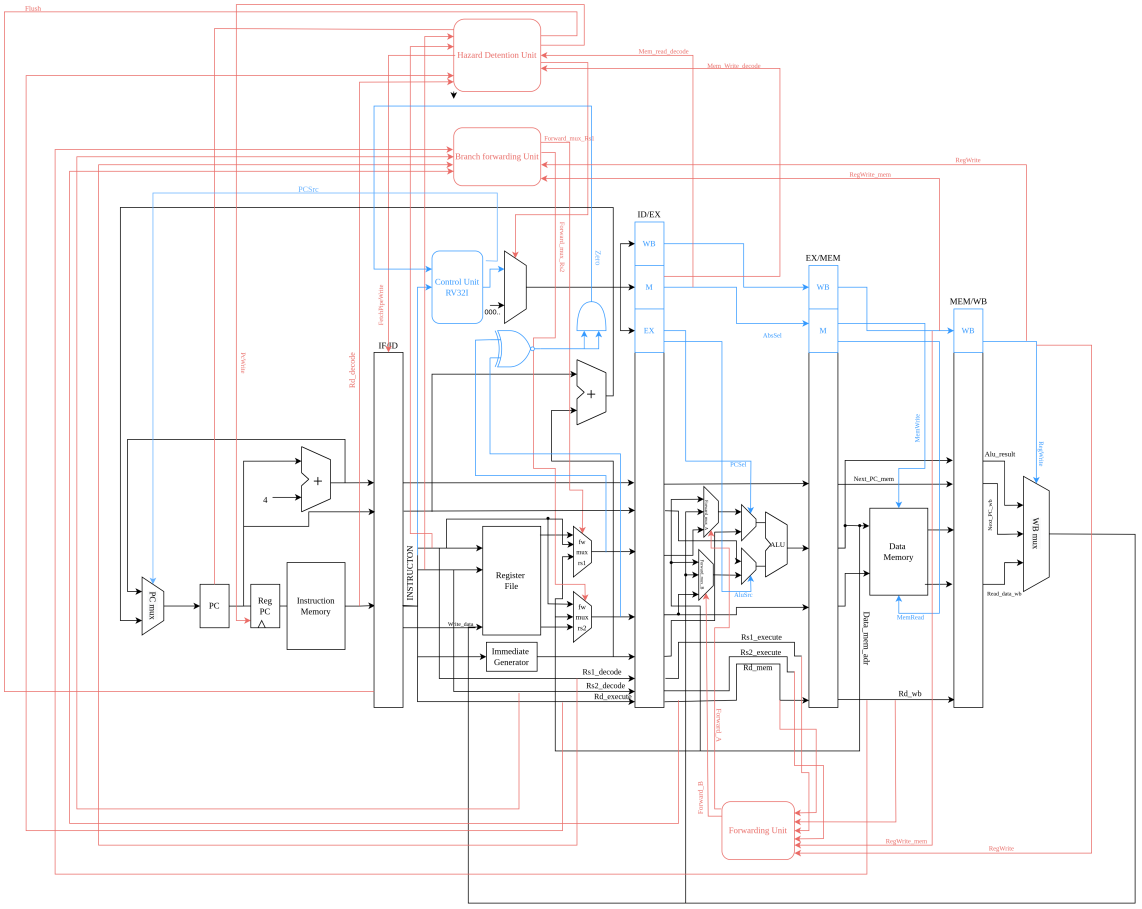

Figure 1: Datapath of CPU with control signals (blue) and forwarding (red)

2.1 Design choices

Given the complexity of the system and the high number of sub-blocks and interconnections, some design rules have been set in order to develop the RTL design and the vhdl descriptions.

Conventions, datapath and control unit Data and controls have been strictly divided at top-entity level, which is composed of blocks for each computational stage and a dedicated control unit. Each stages's output is pipelined and its names reports the destination stage in case it is propagated through multiple stages (e.g. `RegWrite_mem`). Keeping a regular naming convention was crucial to automate port mapping and wiring of the components using Emacs VHDL mode[1], which resulted in quicker bug fixing and more effective design time.

To simplify the control propagation and better handle connections, the CU itself is also internally divided in stage blocks, which are as well pipelined in output. A spreadsheet containing the controls to be generated for every instruction, was realized in advance in

[1] https://www.gnu.org/software/emacs/manual/html_mono/vhdl-mode.html

order to have a control-wise guideline for the realization of the various data blocks.

2.2 Fetch stage

The fetch stage houses the program counter register (PC), which stores the current instruction address. A dedicated adder increment its value by 4 at every clock cycle, to generate the next address in sequence[2]. A multiplexer is instantiated in input to the PC to select whether the processor should go ahead sequentially or jump to a target address. A block diagram of the first stage is reported in Figure 2.

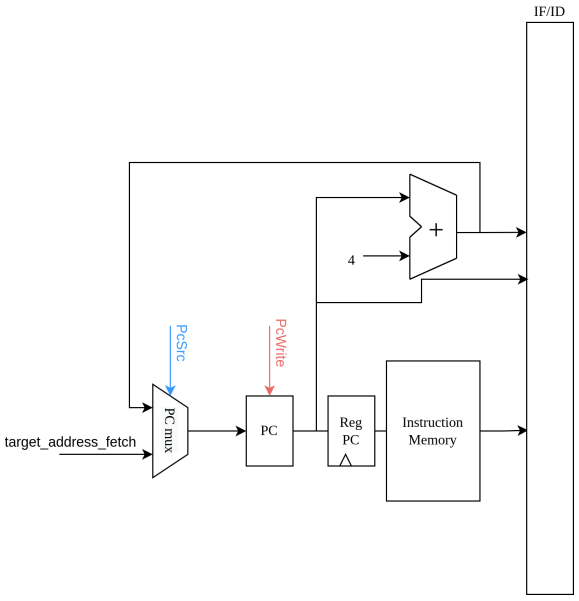

Figure 2: RISCV-lite datapath, fetch stage detail

2.3 Decode Stage

Figure 3 reports a block diagram of the decode stage. In this step, the instruction is split into several sub-fields, which are feed into various blocks. The registers addresses `rs1`, `rs2` and `rd` are provide in input to the register file, while the opcode is interpreted by the main control unit to generate proper control signals for the rest of the machine. Eventually, the immediate fields are decoded by an ad-hoc unit accordingly to the instruction layout.

Conditional and unconditional jumps In order to limit the need of pipeline stalling, BEQ and JAL instruction handling is also performed during the decode phase. The managing hardware consist of a dedicated adder, an equality checker and two multiplexer to employ forwarding from other stages.

[2]The memory is supposed to be byte addressable, so adding '+4' advances the memory address by 32-bit

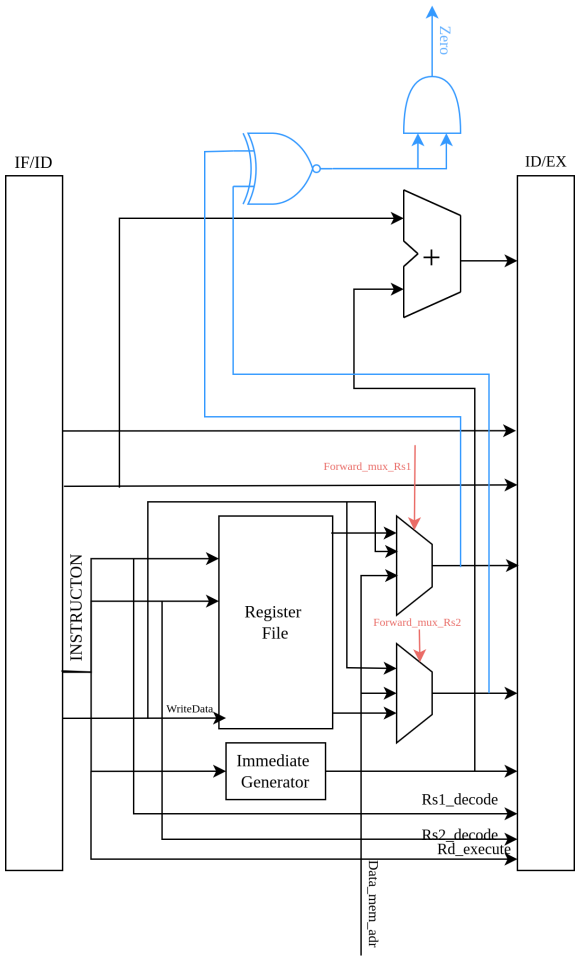

Figure 3: RISCV-lite datapath, decode stage detail

2.3.1 Register file

The core element of the stage is the register file (RF), which accounts for storing information from the ALU and the main memory. Register file is feed in input with three addresses: the first two are used to perform two simultaneous readings (rs1,rs2), while the third one is the destination register address for the write operation (rd).

In order to overcome additional forwarding insertion, the RF perform a write in the first clock half cycle and a read in the latter half. To decouple the operations, the 5-bits selection signals for the output multiplexers are sampled at the falling edge and the pipeline read registers are temporized with an inverted clock signal.

To be enabled for writing, a register must receive the general `WriteReg` and its exclusive enable signal obtained from the decoder. `WriteReg` is always evaluated at the rising clock edge.

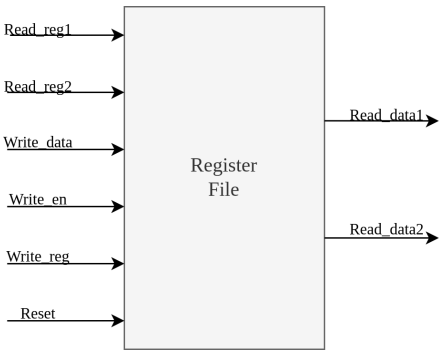

Figure 4: Register File block

— inst[31] —			inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —			inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —		inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —			U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Table 2: Immediate Encoding Variants

2.3.2 Immediate generator

The implemented instructions reported in Table 1 are encoded with 6 possible formats. For each instruction type, the immediate generator organizes properly the immediate fields to produce a 32-bits data. The ordering procedure is pictured in Table 2. Being the instructions aligned as blocks of 32-bits, the immediate target address fetch for B,U and J is represented divided by two, so a zero padding is needed to reconstruct the wanted target address.

2.3.3 Equality checker

The equality checker takes as input the value of the two source registers and outputs a `zero` signal, which is equal to '1' if the registers have the same value, '0' if they have not. The logical implementation of the block is showed in Figure 5.

The output flag is or-ed with the Branch signal from the control unit, in order to decide for the branch-equal-to jump.

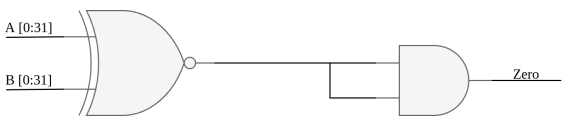

Figure 5: Equality checker block

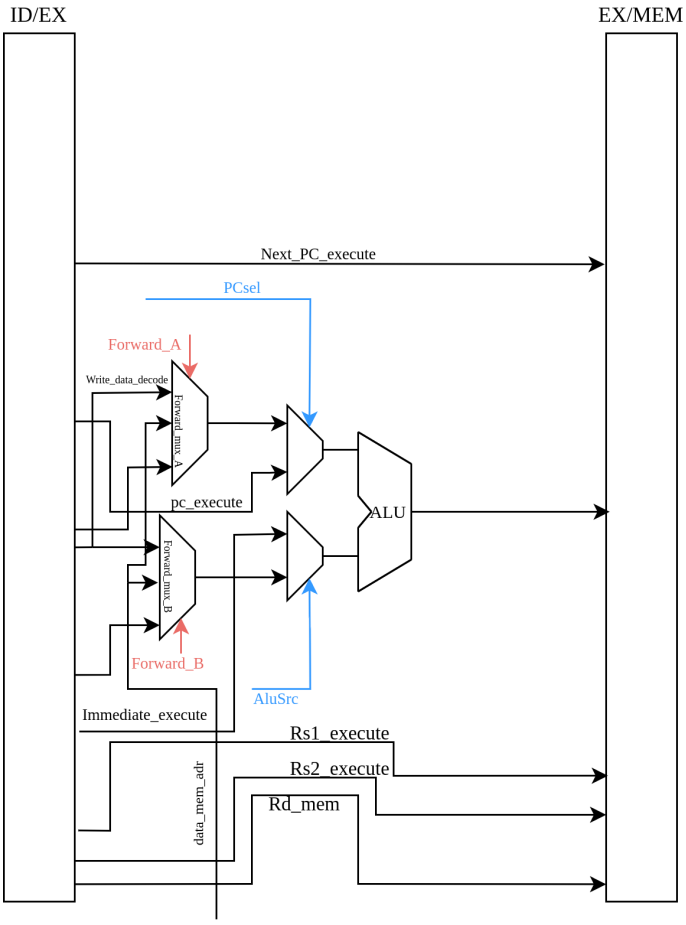

Figure 6: Execute stage

2.4 Execute Stage

The execute stage, detailed in Figure 6, is where the computation is performed. The stage's datapath mainly consists of an ALU block, with two multiplexers connected to its inputs.

For the implemented instructions, the ALU must be able to perform operations on the registers, PC and immediate values.

2.4.1 ALU

The ALU (Arithemic and Logic Unit) supports arithmetic and logic operations needed by the instructions. As showed in Figure 7, it consists of:

- An Adder/Subtracter;
- Two logic blocks for 32-bits bitwise AND and XOR operations;
- A Right Barrel Shifter.

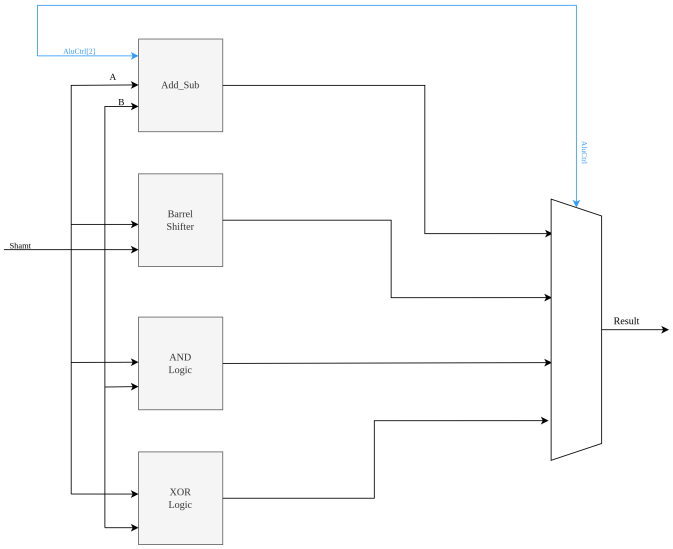

Figure 7: ALU block scheme

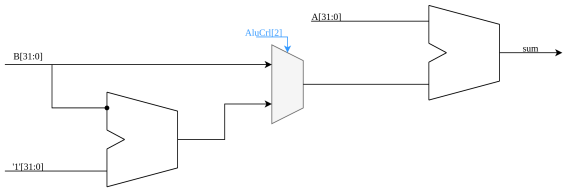

Figure 8: Adder/Subtracter implementation

Add/sub block The implemented adder is reported in Figure 8 To get more flexibility in the synthesis phase, it has been choose to implement the adder with numeric.std's + operator. To avoid the creation of two separate adders for addition and subtraction, a not+incrementer chain has been instantiated to handle the B input CA2 conversion. The selection between the correct B to use as input is handled by a multiplexer.

A `sub` instruction was not requested by specifications, but the subtractor needs to be instantiated to execute the `slt` instruction. In this case, the MSB of the Add/Sub result is used as LSB of the general Alu result, with the remaining 31 bits padded with 0.

Barrel Shifter The Barrel Shifter block is used to perform the `srai` instruction. It performs an arithmetic right shift which amount depends on the value of immediate (`shamt` field) . The block is implemented as a cascade of 160 parallel multiplexers, needed to shift a 32 bits data by 1 to 31 positions right.

2.5 Memory and WriteBack stages

The last operation stages are the Memory and WriteBack. A detailed datapath is pictured in Figure 9.

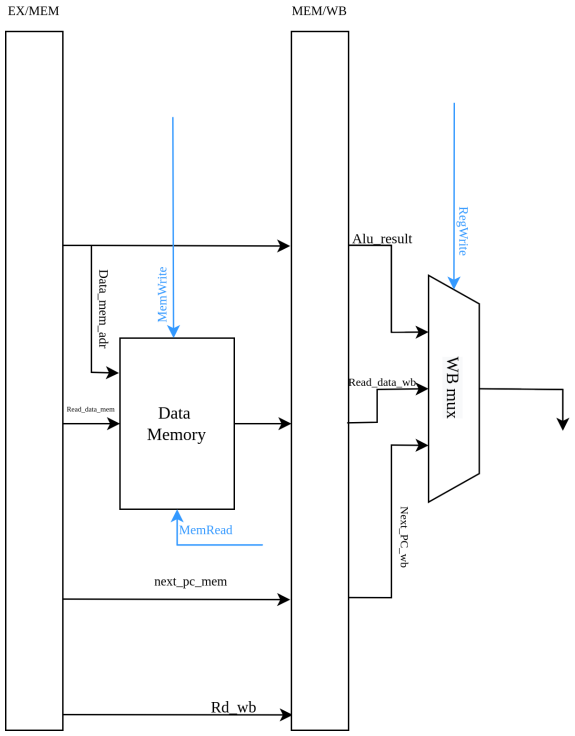

Figure 9: Memory and WriteBack stages

2.5.1 Memory Stage

The memory stages handles load and store operation from the data memory. To perform operation, `MemRead` or `MemWrite` must be active in the proper clock cycle. In the implementation, read and write operations are mutually exclusive.

2.5.2 WriteBack Stage

The last stage consists in a multiplexer that selects which signal to bring to the Write port of the RF. The three possibilities are the Alu Result, the Data Memory output (`lw`) or the current PC+4 (`jal`).

3 Controls

A part from the ALU, which has a dedicated CU, the control dispatch for the entire machine is handled by the main Control Unit. Data dependencies are solved using dedicated units to implement forwarding and pipeline stalling.

3.1 Control Unit

The control unit is feed with the instructions opcodes, formed by the 7 LSB. It manages the control signals according to the decoded instruction, as reported in Table 3. Controls are

Signals	R-type	SRAI/ I-type	LW	SW	BEQ	LUI	AUIPC	JAL
ALUSrc	0	1	1	1	0	1	1	1
PCSel	0	0	0	0	0	0	1	0
RegWrite	1	1	1	0	0	1	1	1
MemRead	0	0	1	0	0	0	0	0
MemWrite	0	0	0	1	0	0	0	0
Branch	0	0	0	0	1	0	0	0
Jump	0	0	0	0	0	0	0	1
MemToReg	00	00	01	00	00	00	00	10
ALUOp	10	10	00	00	01	00	00	11

Table 3: Signals generates by the Control Unit

ALUOp	funct7	funct3	ALUCtrl	Operation
00	-	-	0010	ADD
01	-	-	0110	-
10	-	000	0010	ADD
10	0	010	0100	SLT
10	0	100	0111	XOR
10	1	101	0101	SLT
10	-	111	0011	ANDI
11	-	-	0000	-

Table 4: Signals generates by the ALU Control Unit

used to route the multiplexers, enable the RF and the Data Memory for R/W operations, generate Branch and Jump flags and codes for ALU operations (`ALUOp`).

3.2 ALU Control Unit

The ALU CU combines `ALUOp`, funct3 and bit-30 of the instruction code to generate proper ALU controls. When ALUOp is equal to ”10”, the ALU operation is decided by the fields funct3 and bit-30. If ALUOp is equal to ”00”, it forces the unit to execute an ADD operation. Addition is needed to perform address calculation for `lw` and `sw`; `LUI` requires to sum the immediate with zero, and `AUIPC` requires to sum the immediate with the current value of PC+4.

For other ALUOp values[3], zero is forced as ALU result.

Controls and associated ALU operations are reported in Table 4

[3]The output code 0101 is generated because the ALU was initially in charge of performing the BEQ handling operations.

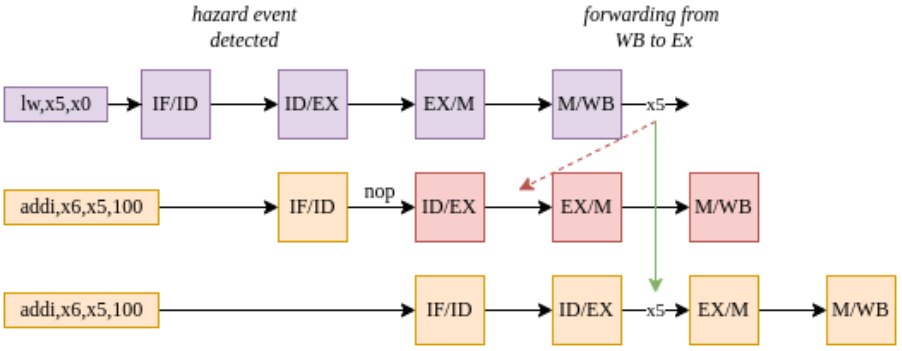

Figure 10: Data dependency detenction and handling. The register x5 data is available for forwarding only in the WB stage.

3.3 Hazard Unit

The Hazard Unit is a FSM with the role of monitoring instruction flow to detect and handle events where the pipeline must be stalled or flushed. A control flowgraph of the machine is reported in Figure 11, while Table 5 provides controls values for each state.

Control Hazard As mentioned before, the processor is able to provide the decision to jump or proceed sequentially in the decode stage. If the fetched instruction is a JAL or the condition for the BEQ is satisfied, the processor should jump to the target address and the following decoded instruction must be substituted with a nop.

Every time this condition occurs, the Hazard Unit commands a multiplexer to override every control generated by the control unit with '0', effectively making the decoded instruction a nop.

Data Hazard If the data is not available when the BEQ is being decoded, the processor must stall the pipeline and wait until the equality checker is able to generate the proper decision. In order to do so, the pipeline and the program counter registers are blocked, while the BEQ is overridden with a nop and the processor proceeds with the rest of the operations.

With the proper forwarding, the processor must stall for one clock cycle if the data is calculated by ALU; two clock cycles are instead required if the target data has to be recovered by memory.

The same stall mechanism applies for other instructions, where data needed in execute stage has yet to be retrieved from memory. An simple instruction flow example is reported in Figure 10.

3.4 Forwarding Units

Forwarding mechanisms have been implemented in order to limit the need to stall the pipeline to solve data dependencies. Two Forwarding Units are in charge of properly

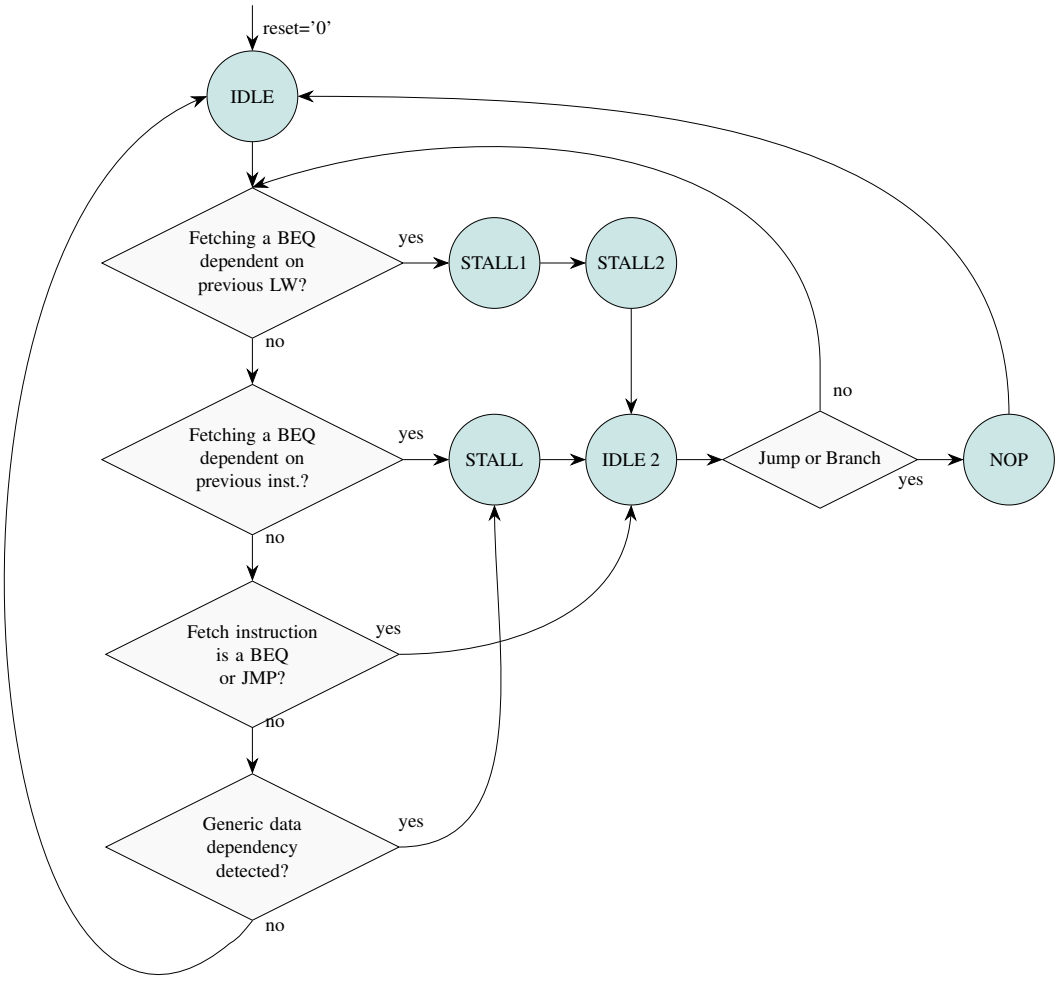

Figure 11: Hazard Unit Control Flow Graph

State	PCWrite	StallSrc	PipeWrite_fetch
Idle,Idle2	1	1	1
Stall, Stall1, Stall2	0	0	0
Nop	1	0	1

Table 5: Hazard Unit FSM truth table

31 25	24 20	19 15	14 12	11 7	6 0
X	X	rs1	X	rd	0001011

Table 6: Custom instruction for the absolute operation with the special unit

routing the data in input at the ALU and at the equality checker. The units checks source and destination registers in order to decide which stage has the most up-to-date version of the target data.

4 Absolute value Special Unit

The original design was modified in order to implement an arithmetic module in the execute stage to perform the absolute value operation.

Instruction A custom instruction was implemented as reported in Table 6. The chosen opcode is the *custom_0* proposed by the RISC-V instruction set manual, with layout conforming to R-type instruction. The special unit is given as input the value of the rs1 register and saves its modulus in the rd register.

Implementation The special unit was implemented in the execute stage and it works in parallel with the ALU. A multiplexer selects among the output of the two units, with a new signal from the control unit called AbsSel.

The mux switches to the '1' position only when the opcode coincides with the absolute instruction one, otherwise remains always to '0'. No changes are introduced in the behavior of the processor during other instructions execution.

5 Verification

Verification of the design unit was carried out using ModelSim to perform the simulations and RARS as validation and test tool.

Testbench and RAM Instruction and data memories, being part of the testbench, were described in vhdl using non synthesizable code. This makes possible to realize a virtual "bootstrap" procedure, where the content of a text file is loaded row-by-row in memory at the beginning of each simulation. Both RAM are 32x256 bit, so only the first 10 bits of address are needed to cover the memories space, the remaining 22 are discarded.

$$\text{\#address bits}=log_2(256 \times 4) = 10$$

To emulate the byte addressing, the output address is divided by four in the testbench to make the memories point at the proper location. As in the RARS environment, if the generated address is not divisible by four, the testbench stops running and warns the user.

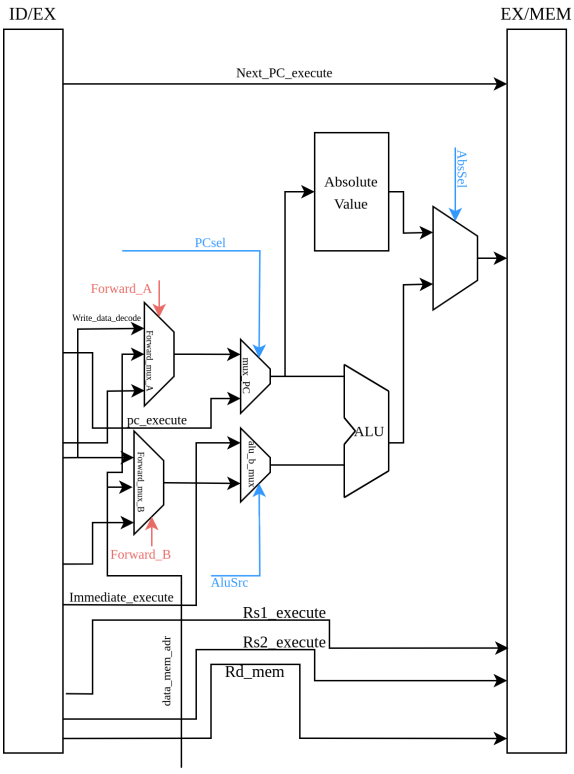

Figure 12: Execute stage with Absolute value block

To speed-up the debug and verification process a custom enumerate type was defined in the testbench to help labelling instruction opcode in fetch, decode and execute stages.

ASM Basic assembly test instructions were coded with the aid of RARS, which was also used to convert instructions to binary text file in order to be loaded by the memory. To validate the adopted instruction set, the processor was also tested with simple C code compiled with `riscv32-unknown-elf-gcc`.

The compiler was built from the RISC-V toolchain with RV32I as target architecture. The command was issued with the option -S to produce the assembly code. Not implemented instructions, such as BLE, were manually substituted with suitable options.

A final test was performed running the minimum absolute value search code provided, with and without dedicated unit availability.

5.1 Arithmetic/logic tests

5.1.1 ALU example test

A simple asm program was coded in order to test all the implemented arithmetic and logic instructions. A C model of the implemented code, used to verify the outcome, is reported below, while Figure 13 illustrates the instruction execution stage in the processor.

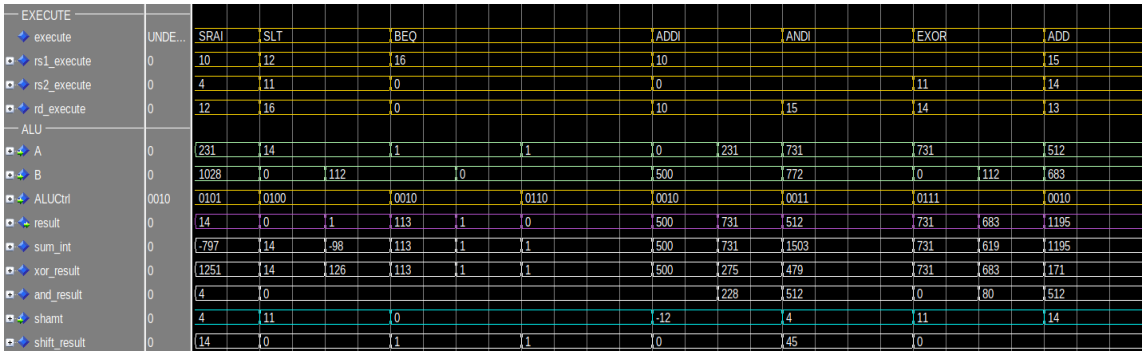

Figure 13: Simulation snapshot of ALU operations (ALU test)

```c
int main() {
  int a = 231;
  int b = 112;
  int srai = 0;
  int add = 0;
  int exor = 0;
  int andi = 0;
  srai = a >> 4;
  if (srai<b){
    a+=500;
  }
  andi = a & 0x304;
  exor = a ^ b;
  add = andi + exor;
  return 0;
}
```

5.1.2 Fibonacci sequence

To test the a gcc-generated asm, a simple C program was coded to produce the first 40 numbers of the Fibonacci sequence. The only needed edit to the assembly was the exchange of the BLE (not implemented) with a SLT+BEQ combination. The execution of the program with the RISCV-lite is reported in Figure 14, while the C code is reported below.

```c
int main() {
  int a = 0;
  int a_1 = 1, a_2 = 1; //initialize Fibonacci sequence
  a = a_1 + a_2;
  for (int i = 0; i < 40; i++) {
```

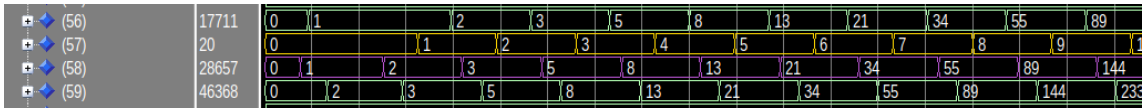

Figure 14: Variable storing in Data Memory, Fibonacci test

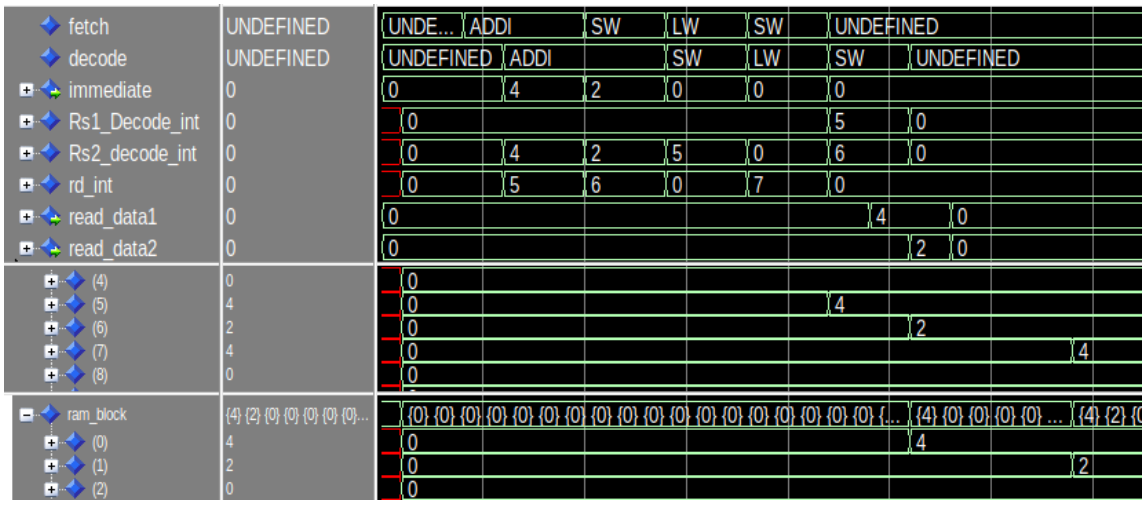

Figure 15: Simulation snapshot of LW and SW instructions

```
    a_2 = a_1; //(57) in the picture
    a_1 = a; //(58) in the picture
    a = a_1 + a_2; //(59) in the picture
  }
  return 0;
}
```

5.2 Load and Store test

5.3 Hello World

The test was carried out to observe the generation and storing of the famous ”Hello World!” string from the C program below. Result is reported in Figure 16.

```
int main() { char text[12] = "lleHoW o!dlr"; }
```

Every char is 8 bit wide (UTF-8), so three memory locations are needed to store the entire string (Figure 16).

Figure 16: Memory load of "Hello World!" string

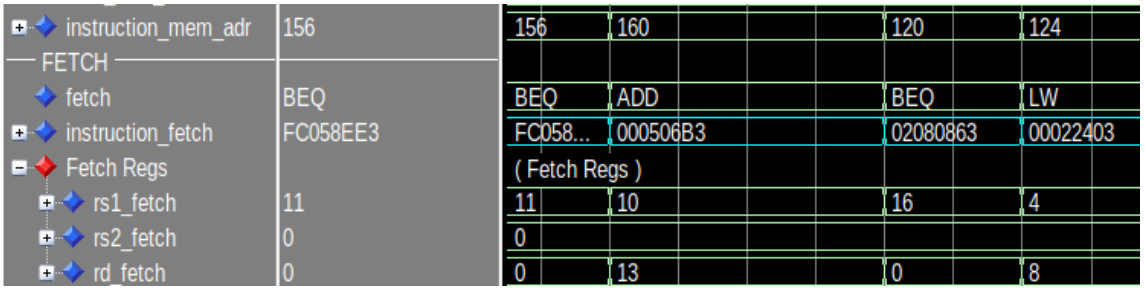

Figure 17: Simulation snapshot of BEQ instruction

5.4 BEQ and JAL test

JAL operation is illustrated in Figure 18. The instruction memory address jumps to the target jump address two clock cycles after the JAL instruction fetch. Analogously, Figure 17 reports the a conditional jump perfomed by the BEQ, with

Data dependency When the hazard unit detects a stall event, it behaves like in Figure 19. As described in subsection 3.3, the BEQ is dependent on the previous SLT. This implies that pipeline must stall for a clock cycle in order for the equality checker to make the proper branch decision. FSM performs according to the CFG in Figure 11.

5.5 AUIPC and LUI test

AUIPC and LUI execution tests are reported in Figure 20 and Figure 21 respectively.

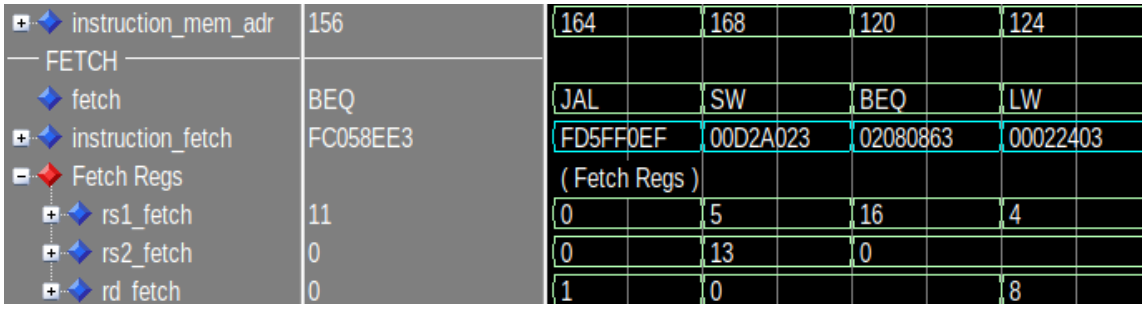

Figure 18: Simulation snapshot of JAL instruction

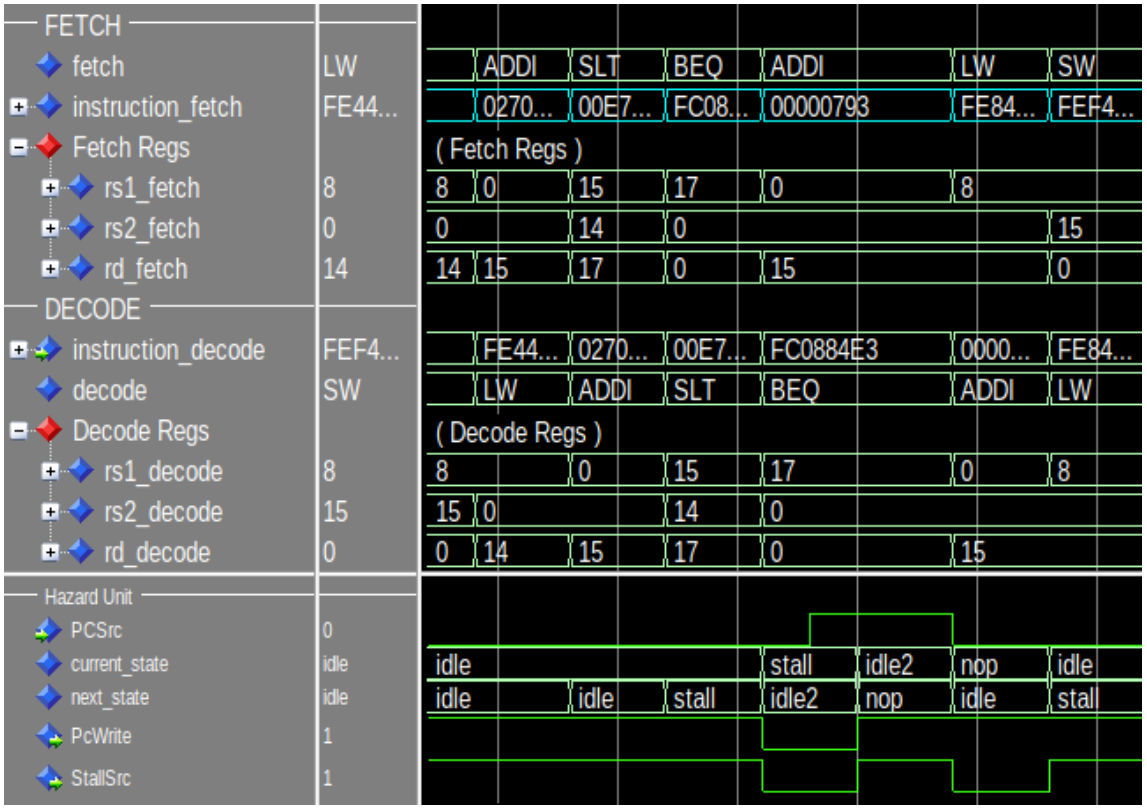

Figure 19: Simulation snapshot of data dependency for the BEQ instruction

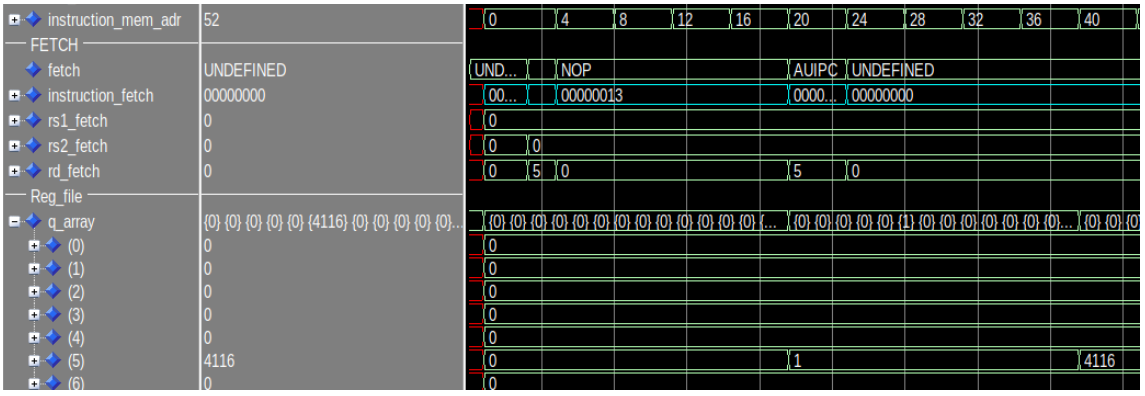

Figure 20: Simulation snapshot of AUIPC instruction

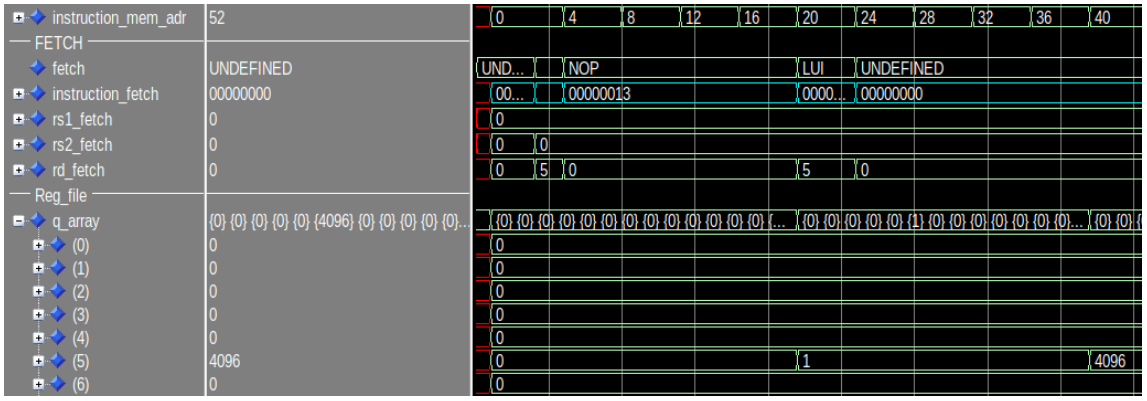

Figure 21: Simulation snapshot of LUI instruction

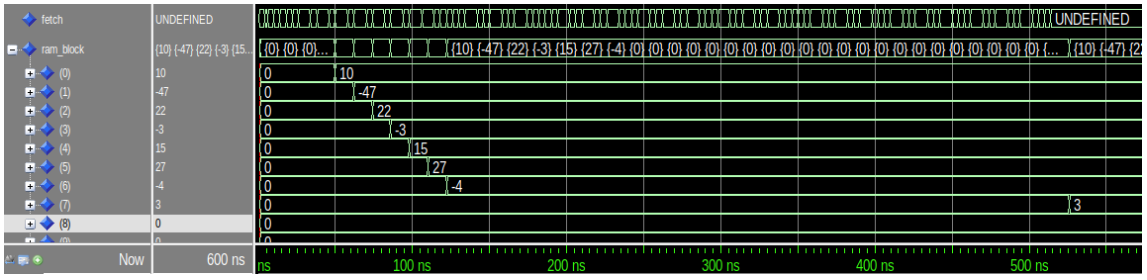

Figure 22: Minimum absolute value search without special unit

5.6 Absolute value program test

In order to test the advantages of having a custom absolute value unit, the provided asm program for absolute minimum value as been tested by both RISCV-lite versions. Code is reported in appendix, subsection 9.1.

Assembly code was modified in order to fill the memory with the vector values. The code used for special unit implementation was then further edited, substituting the four instruction for the absolute value computation with the custom instruction ABSOLUTE.

As expected, the implementation with the dedicated unit is faster than the regular one. Assuming the array length equal to 7, as in the example, a 10% reduction of total computation time is observed. Considering that the amount of instructions per cycle is reduced from 12 to 9, ignoring the overhead for storing the vector, a $\approx 25\%$ improvement in speed can be seen in the CPU with the special unit.

6 Synthesis

The synthesis process was carried on using Synopsys' Design Compiler. The minimum clock period the design without special unit can achieve is equal to 2.22 ns, while the other design reach a minimum of 2.20 ns.

The discovered critical path starts from one output of the register file, proceeds to the ALU

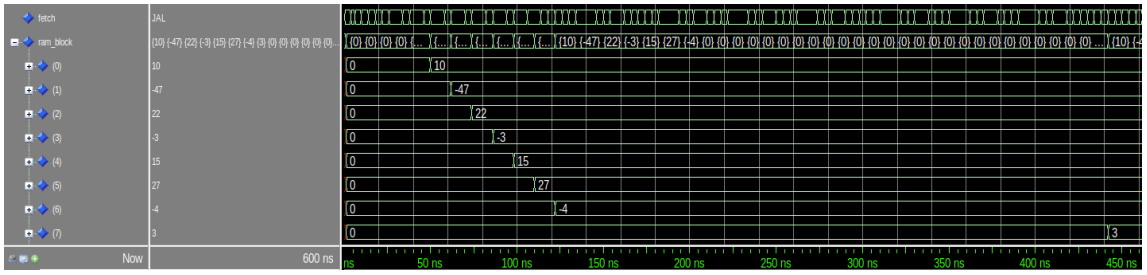

Figure 23: Minimum absolute value search with special unit

[t]

Number of	T_{ck} =2.2 ns
ports:	228
nets:	6971
cells:	6764
combinational cells:	5227
sequential cells:	1536
buf/inv:	613
references:	43

Area [μm^2]	T_{ck} =2.2 ns
Combinational area:	6497
Buf/Inv area:	413
Noncombinational area:	8172
Total cell area:	14670

Table 7: Design compiler area report for CPU

forwarding and selection multiplexers and arrives to the ALU result pipeline register. The design with the special unit have to pass through an additional multiplexer, so a slightly worse critical path is observed ($\approx -1\%$). The results are in accord with the preliminary analysis conducted observing the diagram.

To automate the task of getting the minimum clock period, a custom python/tcl script was coded. Starting from $T_{ck} = 0$, the script iteratively re-synthesized the RTL, increasing each time the clock value until a zero slack(MET) was found.

The summary of the `report_area` command output is reported in Table 7 for the special unit design and in Table 8 for the other one. After the synthesis, the produced netlist has been validated through modelsim.

Number of	T_{ck} =2.22 ns
ports	228
nets	6689
cells	6484
combinational cells	4942
sequential cells	1542
buf/inv	562
references	44

Area [μm^2]	T_{ck} =2.22 ns
Combinational area	6287
Buf/Inv area	378
Noncombinational area	8204
Total cell area	14491

Table 8: Design compiler area report for CPU+special unit

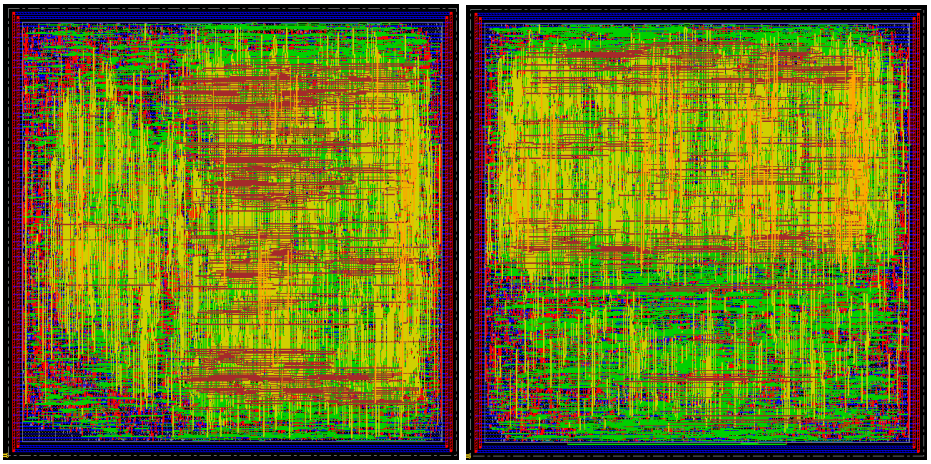

Figure 24: (a) Snapshot of RV32I without special unit from Innovus (b) Snapshot of RV32I with special unit from Innovus

7 Placing and routing

Placing and routing of the filter was conducted with Cadence Innovus.

To complete the design flow, the following steps have been taken, following the supplied guideline:

1. Importing the design
2. Floor planning
3. Power planning and routing
4. Cell placing
5. Signal routing
6. Timing and design analysis

Snapshot of the generated layouts, taken after the last step, are reported in Figure 24. The total area, accounting for the interconnections, has been found to be 14 616 μm^2 (CPU) 15 090 μm^2 (CPU+special unit). The latter is $\approx 3\%$ bigger than the former.

Power estimation After the placing, it is possible estimate the power consumption of the circuit through back-annotation. To do so, the Innovus generated verilog netlist is supplied to ModelSim to run a new simulation.

A *.vcd* file is obtained as result, which contains the nodes activity; the file is then fed back to Innovus in order to generate a power estimation from the netlist. The results of the power estimation are summarized in Table 10 and Table 9. The special unit adds to the CPU $\approx 6\%$ in power consumption.

Power	Internal	Switching	Leakage	Total	Percentage
Sequential	3.864	0.4467	0.1325	4.443	67.22
Combinational	0.8132	1.146	0.1555	2.114	31.99
Clock (Combinational)	7.1e-04	0.052	1.435e-05	0.053	0.7985
Total	4.678 mW	1.644 mW	0.2881 mW	6.61 mW	

Table 9: Innovus power report for CPU without special unit

Power	Internal	Switching	Leakage	Total	Percentage
Sequential	4.0	0.47	0.13	4.6	65
Combinational	0.98	1.253	0.17	2.4	34
Clock (Combinational)	7e-04	0.053	1.435e-05	0.05426	0.7691
Total	4.969 mW	1.781 mW	0.304 mW	7.054 mW	

Table 10: Innovus power report for CPU with special unit

8 Considerations and final remarks

The implemented CPU is just a proof of concept, the processor misses lot of important instruction from the RV32I set, there is no support for interrupt and exception handling and the RISCV-lite has no way to communicate with the external world.

However, as demonstrated, it is able to execute simple C programs using rv32i-gcc with just little modification of the asm instructions. More importantly, the work showed the dramatic speed enhancement implied by having dedicated hardware in the specific proposed scenario. Little extra cost in terms of area and power dissipation was observed with regard of computational time improvement. This leads to understand the commercial trend of adding more and more co-processors to the main CPU seen in the recent years.

9 Appendix

9.1 Minimum absolute value program test

```
##################
# Basic VERSION
# This program takes an array v and computes
# min{|v[i]|}, the minimum of the absolute value,
# where v[i] is the i-th element in the array
        .data
        .align        2
v:
        .word 0
```

```
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
        .word 0
m:
        .word 0

        .text
        .align        2
        .globl        __start
__start:
        li x16,7          # put 7 in x16
        la x4,v           # put in x4 the address of v
        la x5,m           # put in x5 the address of m
        li x13,0x3fffffff # init x13 with max pos

        # Store data 1
        li x20, 10
        sw x20, 0(x4)
        addi x4, x4, 0x4
        # Store data 2
        li x20, -47
        sw x20, 0(x4)
        addi x4, x4, 0x4
        # Store data 3
        li x20, 22
        sw x20, 0(x4)
        addi x4, x4, 0x4
        # Store data 4
        li x20, -3
        sw x20, 0(x4)
        addi x4, x4, 0x4
        # Store data 5
        li x20, 15
        sw x20, 0(x4)
        addi x4, x4, 0x4
        # Store data 6
        li x20, 27
        sw x20, 0(x4)
        addi x4, x4, 0x4
        # Store data 7
        li x20, -4
```

```
        sw x20, 0(x4)
        # Store 0 in m
        sw x0, 0(x5)

        la x4, v
loop:
        beq x16,x0,done    # check all elements have been tested
        lw x8,0(x4)        # load new element in x8
        srai x9,x8,31      # apply shift to get sign mask in x9
        xor x10,x8,x9      # x10 = sign(x8)^x8
        andi x9,x9,0x1     # x9 &= 0x1 (carry in)
        add x10,x10,x9     # x10 += x9 (add the carry in)
        addi x4,x4,0x4           # point to next element
        addi x16,x16,-1    # decrease x16 by 1
        slt x11,x10,x13    # x11 = (x10 < x13) ? 1 : 0
        beq x11,x0,loop    # next element
        add x13,x10,x0     # update min
        jal loop           # next element
done:
        sw x13,0(x5)       # store the result
#endc:
#        jal endc             # infinite loop
#        addi x0,x0,0
```