Politecnico di Torino

Integrated Systems Architectures

Report for Lab 3

https://github.com/levnikolaevicmiskyn/ISA/tree/main/lab3

Group 12

Antonio Carlucci 276128
Gabriele Perrone 269089
Alessandro Scisca 276032

Contents

Chapter 1

Introduction

The aim of this laboratory is to design, test and synthesize a RISC-V compliant processor able to execute a selected set of instructions according to the official standard and specifications.

1.1 Instruction Set

The supported instructions are listed in Table 1.1.

Instruction	Format	Description
`add`	R	Addition
`addi`	I	Addition with immediate
`xor`	R	Xor
`slt`	R	Set register to 1 if rs1<rs2
`and`	R	And
`andi`	I	And with immediate
`srai`	I	Shift right arithmetic (with sign extension) with immediate
`auipc`	U	Add upper immediate to pc
`lui`	U	Load upper immediate to a register
`beq`	SB	Branch if register operands are equal
`lw`	I	Load word from memory
`jal`	J	Jump and link
`sw`	S	Store word

Table 1.1: Supported instructions

Chapter 2

Implementation

2.1 RISC-V Processor

The processor is implemented as a pipelined processor where the execution of an instruction is divided in the following stages:

1. Instruction Fetch
2. Instruction Decode
3. Execution
4. Memory
5. Write Back

Each stage is independently described within its own component which implements all of the necessary functionality. The pipeline - specifically, the pipeline registers - is described and managed in the top level entity: this means that every stage is implemented as a combinational component[1]. This allows for clean descriptions of the stages since they do not have to deal with instancing and controlling the pipeline registers.

The processor is also endowed with two specific units that improve the overall performance: the **Forwarding Unit** and the **Branch Prediction Unit**, which minimize the number of pipeline flushes and stalls introduced during the execution of a thread.

To simplify the description of the pipeline, the processor and the top level entities of the various stages use VHDL records, all described in a global package. This allows for simpler signal assignments as well as a more versatile implementation: whenever a stage needs a new signal that was not accounted for, or any modification in general is needed, it is sufficient to update the record and the signal assignments, but no entity or instance has to be modified.

The specific descriptions of each unit, together with their design choices, functionalities and general discussions are listed in the following sections.

[1] Of course, single registers may be needed in various occasions, the clearest instance is the Program Counter. But it is the duty of the single stage to not interfere with the pipeline and internally deal with the components to maintain a correct timing

2.2 Instruction Fetch

The purpose of this unit is to:

- hold the current value of the program counter (`PC`);
- increment PC when executing sequential code;
- fetch the next instruction from the instruction memory;

2.2.1 Structure

A RTL description would contain a `PC` register, treated as a special register within the processor and an incrementer to compute `PC+4`. Normally, the PC is updated with the next sequential address. However, when a jump occurs, a selection logic driven by the control unit inside the ID stage causes the PC to be updated using an address coming from a dedicated adder that computes the target address for jumps or branches. In the case of a stall, the PC is not updated at all and the same instruction is re-fetched from memory.

This stage reads from the instruction memory, requiring a 32-bit output port to deliver the desired address and a 32-bit input port to receive the instruction word.

Figure 2.1 shows an high-level representation of this part of the processor. The `stall` and `jump` signals are provided by the control unit and cross the stage boundary. The loop involving the jump address calculation with the dedicated adder in the ID stage and the selection logic in this stage is potentially a critical path and it crosses the pipeline boundaries. However, we will see that the path including the main adder in the execution stage is way longer.

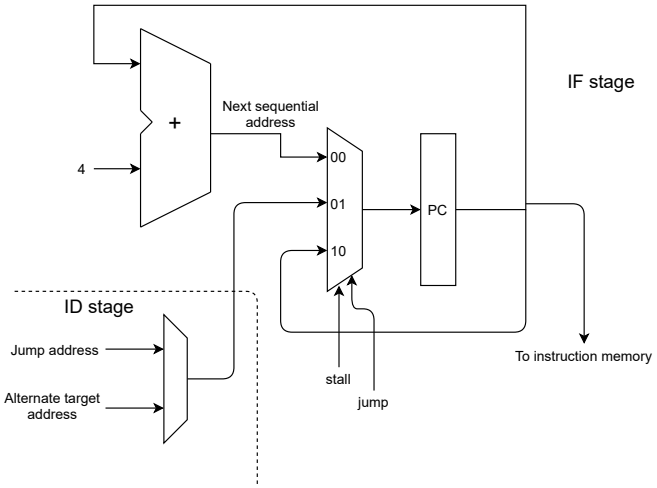

Figure 2.1: RTL schematic of the instruction fetch stage, with part of the ID stage reported which selects the right jump address according to whether it is triggered by a jump instruction or a misprediction.

2.3 Instruction decoding

The instruction decoding step occurs just after the instruction is fetched from memory. Its purpose is to identify the operation to be performed, generating control signals for subsequent stages. Its tasks also include detecting particular execution states where the pipeline has to be stalled or flushed. Given the variety of its purposes, this block is furtherly broken down into several sub-units hierarchically:

- Decoder
- Hazard Detection Unit (HZU)
- Branch Prediction Unit (BPU)
- Target address computation
- Register File

2.3.1 Decoder

The simplicity of RISC instruction sets consists in including few instructions that perform simple tasks. Furthermore, the instruction format is fixed, so as to allow the decoding circuitry to be as simple and fast as possible. Five types of instruction are defined in RISC-V, with the following types of fields that could be present depending on the actual format:

- Opcode
- Specialized opcode fields for arithmetic operations (funct3 and funct7)
- Destination register
- Source registers (one, two or none)
- Immediate field, whose size and format varies across types

Since the position of each of these fields is fixed, they can be extracted into separate signals without detecting the instruction type first. After instruction decoding, the relevant fields are used and invalid signals corresponding to inexistent fields are just ignored.

The main task of this sub-unit is to drive control signals for execution, memory and write-back stages, which includes part of the ID stage itself given that data is written back to the register file.

The VHDL description consists in a combinational process whose sensitivity list includes the opcodes and the source registers. Based on `rs1` and `rs2`, this unit will resort to the hazard detection unit to handle the load-use data hazard occurring when the source operand is to be loaded from memory within the following two clock cycles. Under this circumstance, an operation that depends on such operand cannot proceed to the execution stage because it will be too early for the required data to be available of forwardable. Whenever the HZU runs into a load-use data hazard, the decoding step produces a NOP (no operation), meaning that all control signals are inactive and the instruction fetch stage is prompted to re-fetch the current instruction without updating `PC`. The normal execution flow proceeds as soon as the hazard is removed.

2.3.2 Hazard Detection Unit

This subunit takes the source register indices and the `rd` signals (destination register name) both at the output of ID/EX and EX/MEM pipeline stages. If any of `rs1` or `rs2` is the same as `rd` and the corresponding data memory read enable signal is active, then the condition for a potential load-use data hazard is verified. The instruction decoder will trigger a NOP insertion after validating the hazard signal coming from this unit, which is effective only if the instruction depends on the data contained in the register involved in the hazard.

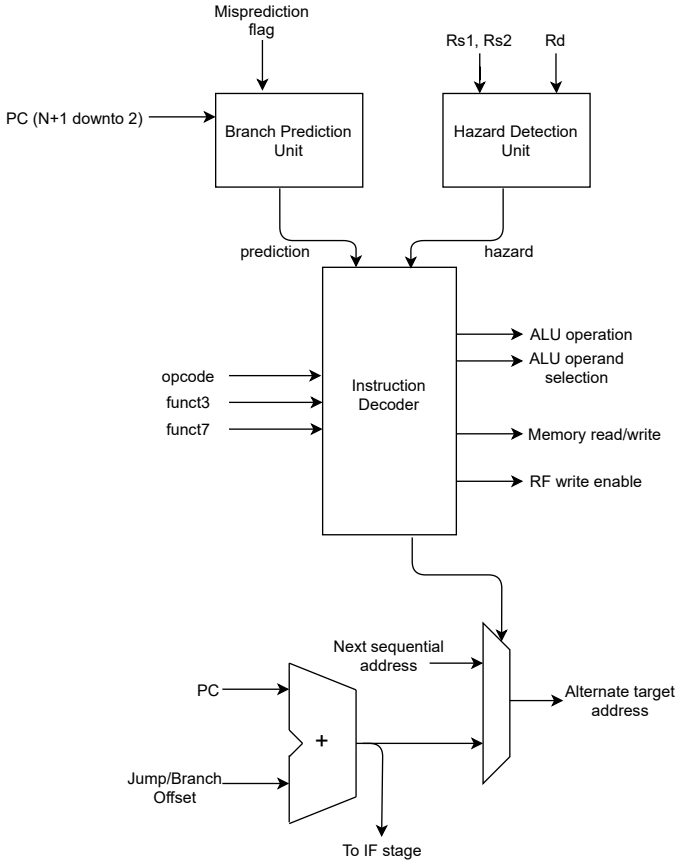

Figure 2.2: Instruction decoding stage with its sub-units

2.3.3 Branch Prediction Unit

The purpose of this block is to estimate which is the most likely path that the execution will take in case of branch instruction. The actual outcome of the branch condition is known as soon as `beq` reaches the MEM stage, which happens with a two clock cycles delay. The BPU uses this outcome to update its internal state according to algorithms specifically designed to refine the prediction accuracy by collecting execution statistics.

In our implementation, branches are statically predicted to be taken, in a similar way as prediction schemes used in early microprocessors. The introduction of a BPU influences the timing of the processor. With a BPU, the pipeline is stalled upon decoding a `beq` for only one clock cycle if the branch is predicted, the minimum required to jump to a non-sequential address. The actual execution path as determined by checking the `Z` flag is compared to the prediction only two clock cycles later by combinational logic contained in the MEM stage. A misprediction causes the pipeline to be flushed (pipeline registers are reset to a `NOP` state) and the IF stage to jump to the alternative target address available in the EX/MEM pipeline register, which contains the alternative address computed in the ID stage.

A basic one-level scheme is the bimodal predictor, consisting in an array of 2-bit saturating counters addressed by part of the instruction address. Whenever a branch is decoded, the counter corresponding to its address is read, thus giving a 'taken' prediction when its value is greater than 1 and 'not-taken' otherwise. As soon as the real outcome of the branch is available the counter is incremented in case of a taken branch and decremented otherwise. This, in our implementation, occurs two cycles after decoding, when the branch is in the MEM stage. In general this delay, which affects the branch penalty to be paid in case of a flush, depends on the number of pipeline stages in between ID and MEM.

2.3.4 Target address computation

This is accomplished simply by summing the offset specified in the instruction's immediate field (properly aligned and extended to 32 bits) to the current program counter. When the branch is predicted to be taken, this address is loaded in the pc and a jump takes place, with the next sequential address loaded in the alternative target address register. Otherwise, the TA is the alternative target address and the execution continues sequentially.

2.3.5 Immediate

The immediate field is encoded differently depending on the instruction format. The ID stage takes care of aligning all the bits from the instruction word correctly and extending the sign bit to deliver a 32-bit immediate operand. This can take place after opcode decoding.

This is described in VHDL using a combinational `when-else` statement to synthesize a selection logic.

2.3.6 Register file

As prescribed by the specifications, RV32I entails grouping 32 internal registers holding 32-bit data in a register file. `x0` denotes a location that always returns zero when read, thus making all write operations targeting it equivalent to not writing any data back.

The VHDL description of this sub-unit is that of a standard memory, with additional care taken to obtain the correct synthesis for the `x0` location, which might not correspond to a physical register.

2.4 Execution Unit

The Execution Unit is the unit responsible for the computations and the general execution of the commands. The core and most significant component is the ALU, which the Execution Unit simply surrounds with multiplexers and their controller to select the operands.

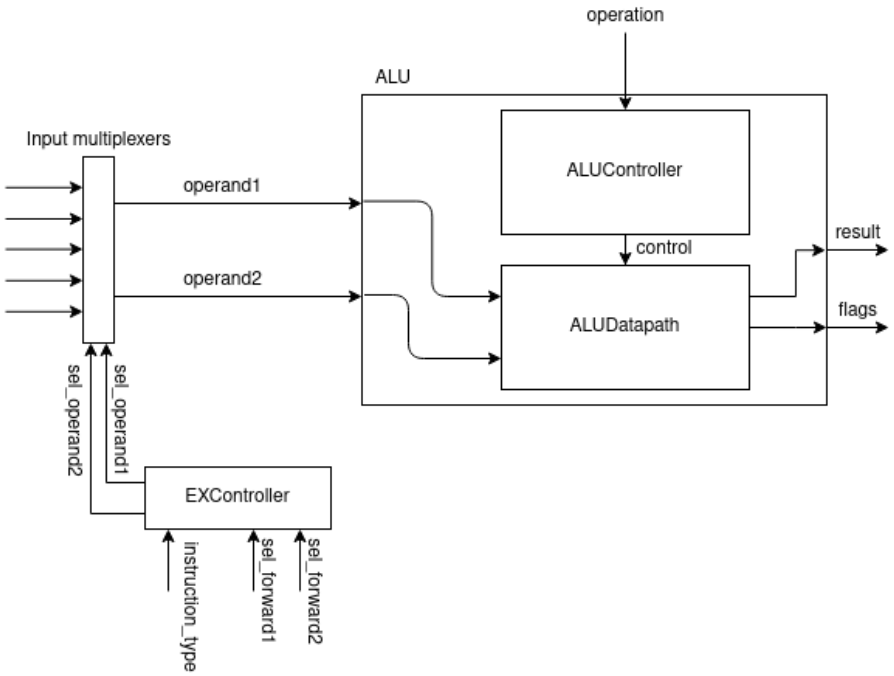

Figure 2.3: Block view of the execution stage

2.4.1 ALU

The ALU is also internally organized with a datapath and its own controller. It is important to underline that to not interfere with the processor's pipeline, this controller (as well as the Execution Stage's) is fully combinational and mostly acts as an input translator.

Datapath

The datapath instantiates the arithmetic and logic operators, generates the status flags and outputs the final result. All the operators are instantiated in parallel and, at every cycle, they all run their own operation and the result is selected through a multiplexer. If power consumption is to be considered, guarded evaluation can be a simple solution to optimize this behavior.

The available operators are:

- Carry Look Ahead adder
- Barrel shifter
- And
- Xor
- Comparator

Most of the functionalities are straightforward: the `and` and `xor` operators are simple logic gates and the barrel shifter is described through the `numeric_std` library's function `shift_right`. The components worth discussing are the adder and the comparator.

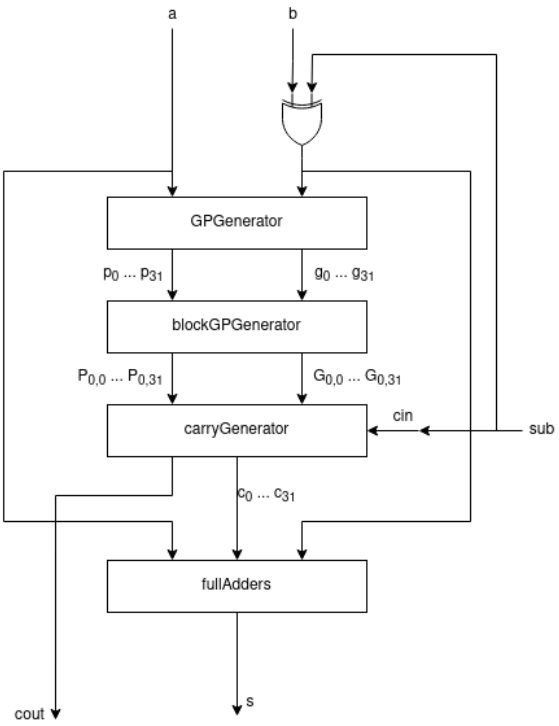

Figure 2.4: Carry Look Ahead adder organization

Adder The adder is designed as a **Ladner-Fischer** Carry Look Ahead. Its structure is organized as a cascade of the following functional blocks:

1. Sign converter
   To accomodate for subtractions, operator `b` goes through a `xor` gate controlled by the `sub` signal which, when asserted, inverts `b` and sets `cin` to 1.

2. Generate and Propagate bits generator
   Given `a` and `b`, output the corresponding generate and propagate signals bit by bit, where

$$g_i = a_i \texttt{ and } b_i$$
$$p_i = a_i \texttt{ xor } b_i$$

3. Block Generate and Propagate bits generator
   Starting from g_i and p_i, compute the block generate $G_{0,i}$ and propagate $P_{0,i}$ as a parallel prefix problem. This is where the Ladner-Fischer structure is instantiated.

   The main component needed to compute the block generate and propagate is called `GPCombiner` which, given $P_{i,j}$ and $P_{j,k}$, combines them into $P_{i,k}$ - the same goes for G. The inputs g_i and p_i (lowercase, meaning they are bit-wise and not block-wise) are assumed as $G_{i,i}$ and $P_{i,i}$, which is equivalent. The operation is simple:

$$G_{i,k} = G_{j,k} \texttt{ or } (G_{i,j} \texttt{ and } P_{j,k})$$
$$P_{i,k} = P_{i,j} \texttt{ and } P_{j,k}$$

   The combiners are interconnected according as in the schematic in Figure 2.5 in a module called `blockGPGenerator`. To keep the generality of the component, the block is described with generics and its structure is instantiated **recursively**. More information on the VHDL description can be found in section A.1.

   This is the slowest stage since paths of different bit weight are interconnected and they form a cascade. The choice of the Ladner-Fischer architecture makes so that the introduced delay is $\log_2 N$.

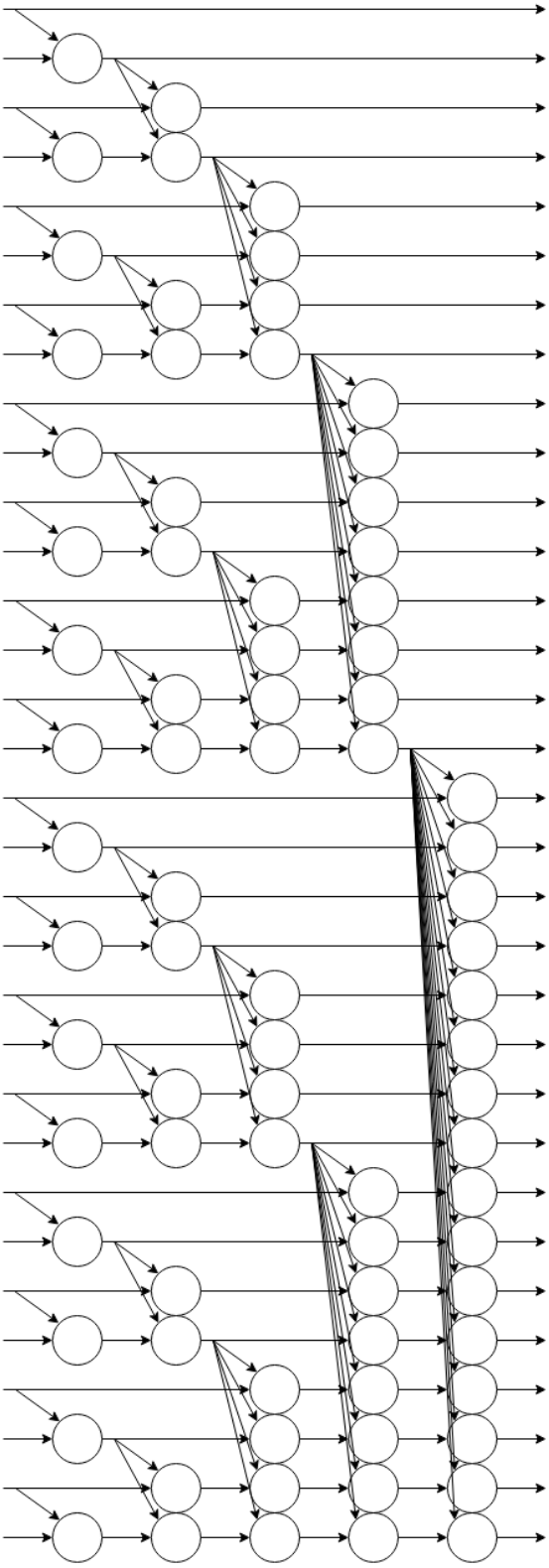

Figure 2.5: Ladner-Fischer GPCombiner. Each circle represents an instance of the component GPCombiner.

4. Carry Generator
   Given the results from the previous stage, it is possible to compute the value of all c_i in parallel, since

$$c_i = G_{0,i} \texttt{ or } (c_0 \texttt{ and } P_{0,i})$$

5. Full adders
   The final stage is a barrier of full adders. At this point, every adder has all of its inputs ready: a_i, b_i, c_i. For this reason there is no need for one adder to wait for the result of another adder, so they can all operate in parallel, so the introduced delay is only 1 ^{t}FA.

6. Flag generators
   The adder also generates two overflow flags to be used to control the flow of operations: one for signed sums, one for unsigned ones. In particular,

$$\texttt{cout} = C = c_{32}$$
$$\texttt{ovf} = V = c_{32} \texttt{ xor } c_{31}$$

The ALU also internally computes all of the status flags: C and V are obtained directly from the adder, Z and N are trivial to obtain by checking the sum result.

Comparator A comparator is usually implemented as a subtractor that then performs some checks on its result, given that the generic comparison operation ? between A and B can be expressed as

$$A \ ? \ B \rightarrow A - B \ ? \ 0 \tag{2.1}$$

Since an adder/subtractor is already available and all of the relevant observations on the result were performed for the status flags, the comparator is implemented as an extension of the adder that only checks the status flags and outputs a result based on the requested comparison, as seen in the schematic in Figure 2.6.

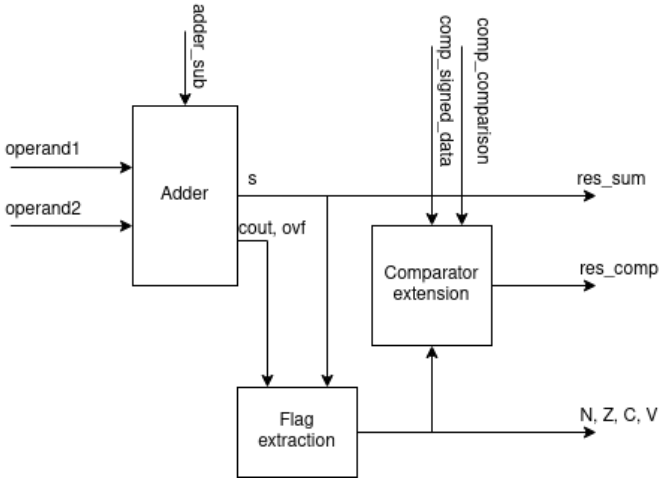

Figure 2.6: Schematic of the comparator implemented as an extension of the adder

It internally computes all of the possible comparisons and a multiplexer selects the result to output. In particular,

$$lt = N \texttt{ xor overflow}$$
$$le = lt \texttt{ or } Z$$
$$eq = Z$$
$$ge = \texttt{not } lt$$
$$gt = \texttt{not } le$$

The `overflow` signal is either `C` or `V`, depending on whether the comparison should consider its operands as respectively unsigned or signed.

It is clear to see how it is the controller's job to correctly deliver all of the necessary commands when a comparison is requested, such as switching the adder to subtracting mode, deciding whether the operands are signed or not and selecting the output.

ALU Controller

The controller for the ALU receives an ALU opcode to choose what operation to perform and then internally manages all the necessary signals. It is described with a simple process and a switch on the opcode that updates the control signals, as in any classical control unit output process.

For example, when the opcode is `alu_op_lt`, the control unit sets `adder_sub` to 1 (to perform a subtraction instead of an addition), switches the ALU multiplexer to select the comparison result, specifies to the comparator that the data is signed and switches its multiplexer to choose the `lt` result.

All of the control signals are packed inside of a VHDL record to simplify possible future changes without the need of modifying any entity in the circuit.

2.4.2 Execution Controller and Input Selection

Depending on the instruction to be executed, different sources for the two operands may be needed. Overall, the execution stage receives the following signals:

- Two entries from the register file
- Immediate
- Program counter
- Data from the Memory stage (used in forwarding)
- Data from the Write-Back stage (used in forwarding)

To these, some constants are inserted as additional options, such as the constant 0 being used in various operations. The two operands which are fed to the ALU are selected from the EXController. In order to pick the operands, it analyses three control signals: `instruction_type` from the Control Unit and the two `sel_forward` from the Forward Unit.

The control logic works as follows: first, `instruction_type` is analysed. This signal specifies what kind of operators the current instruction works with, specifically if each one is a register, a specific constant, the immediate and so on. Whenever an operand is not a register, this request is immediately accepted and the corresponding operand is selected since there are no risks of hazards. Whenever one operand is a register, instead, hazards are a risk and the signals from the forwarding unit should be considered as well.

The candidates for operand 1 and 2 are not necessarily the same, so they are categorized in a generic way that allows all the supported instructions to be executed but still optimizes the selection process. The multiplexers for the two operands may then be connected to the specific signals associated to the generic selector. All of the possible values for both operands are listed in Table 2.1.

Selector	Operand 1 value	Operand 2 value
`SEL_ZERO`	0	0
`SEL_OPERAND`	Register 1	Register 2
`SEL_CONST`	0	4
`SEL_SPECIAL`	PC	Immediate
`SEL_FWD_MEM`	Data from Memory stage	Data from Memory stage
`SEL_FWD_WB`	Data from Write-Back stage	Data from Write-Back stage

Table 2.1: Possible operand values for both operands in the Execution stage

2.5 Forwarding Unit

Forwarding unit is a unit that works across Execution stage, Memory Stage and Write-back stage. Its role is to prevent read after write data hazard that can occur between successive instructions due to the pipelined implementation of the processor. To do so, it compares separately the source register Rs1 and Rs2 of the instruction at Execution Stage first against destination register Rd of the instructions at Memory Stage, then against the one of the instruction at Write-back stage. If there is a match it checks also that this register is not x0, and that the destination register will be effectively wrote-back into the register-file, otherwise the forwarding will be useless or wrong.

Performed all the comparisons, the Forwarding unit tells to the execution if the correct operands are to be taken from register-file, Memory Stage or Write-back stage.

2.6 Adding ABS

Given the highly parametrized implementation, supporting an additional instruction set in a second version of the same processor is straightforward.

2.6.1 Changes in the Control Unit

As far as the decoding is concerned, the addition of the absolute value instruction amounts to adding a dedicated case in the combinational process describing the instruction decoder along with a specific ALU operation. The new ALU operation is just appended to the enumerated type that was already defined in the previous version. The new instruction is R-type and its opcode is 0110011, while the funct3 field is 000.

2.6.2 Changes on the Execution stage

In order to best utilize the already available hardware, the addition of an ad-hoc unit was discarded. The adder path is already used for nop-like operations, where the passed operand is simply summed to 0; being an adder/subtractor, it also offers the possibility to invert the sign of one operand (specifically, operand 2), which is the only operation needed in this case. For this reason, the only real changes happened at the controller level.

Given the generality of the description, accomodating for a new instruction did not require any change since its opcode was simply added in the `t_alu_op` enumeration. In this specific case, the operand of interest is passed to ALU's operand 2, but its inversion still needs to be conditional. This requires two additional signals to be sent to the ALUController, called `operand1_info` and `operand2_info` of a newly described record type called `t_OperandInfo`.

Currently, an OperandInfo signal only keeps track of the sign of the operand, but it could also hold some additional information[2]. The controller, upon receiving the opcode corresponding to the `ABS` operation, can simply check for operand2's sign and issue a subtraction only when the value is negative.

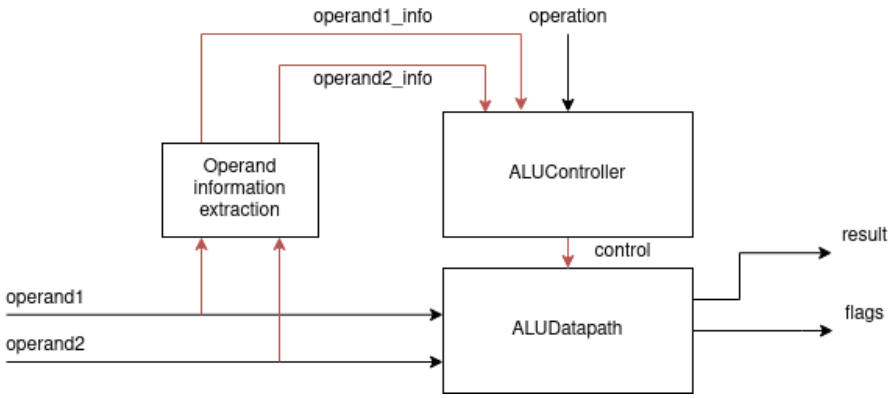

Figure 2.7: ALU with the new control paths needed for the implementation of ABS

The downside of this simple solution is a longer combinational path. Initially, the operand paths and the controls path was separated: the controls were determined from the processor's Control Unit and the operands were directly connected from their source to the datapath.
In this new scenario, operands also go through an information-extraction stage (which, in ABS case, does not add any logic) and then go through ALUController as well, generating the control signals and finally arriving in the datapath.

If the extracted information is kept as simple as possible like in this case, where only the sign bit is sufficient to perform the operation, the high level of controllability and customizability well justifies the introduced timing costs. In the cases where a more complex operation is needed, instead, it is best to implement a custom unit to maintain balance in the logical paths.

[2]As discussed later, this connection between the operands paths and ALUController creates a longer combinational path. It is best to keep the extracted information as simple as possible, only considering for instance the operand's sign, its parity or a simple sticky bit.

Chapter 3

Verification

3.1 Instruction decoding

This sub-unit was tested along with the rest of the pipeline, since several signals coming from other stages are important to determine the control flow. For instance, the misprediction flag is generated in the MEM stage and the data used by the hazard detection unit comes from the pipeline registers. The instruction decoding stage was connected to a behavioral ALU in order to focus the debugging effort on the instruction sequence. Figure 3.1 shows the occurrence of a branch (at $t = 1.2\,\mu$s), the corresponding jump due to the taken prediction and the subsequent recovery two clock cycles later, when the prediction is found to be incorrect.

3.2 Execution Stage

In this stage, the most significant component to verify is the ALU with its internal operators. Initially, two small testbenches were described to quickly test the adder and the barrel shifter (`tb_adder` and `tb_barrelShifter`), mostly to check if they compiled and if they produced some results, meaning no signals were left unassigned nor were they driven by multiple sources. When `tb_ALU`, a more functional and flexible testbench was realized, they were discarded and all the testbenches happened at the ALU datapath level.

The testbench process is composed of the following parts:

- A Python script that generates meaningful inputs for an ALU with a specific file format
- A C++ description of the same ALU that generates the reference results
- A VHDL testbench that simulates the hardware behavior

The reference and the simulation results can then be compared to each other to complete the test.

3.2.1 Input generation

Inputs are generated through a very simple Python script, that generates a certain number of lines (specified through the command line) using the random number generators. Each line of the generated output contains the following fields separated by spaces:

1. 8 hex digits for operand 1
2. 8 hex digits for operand 2
3. 1 hex digit for the ALU opcode

which are all the inputs of an ALU. Since, in VHDL, the ALU opcode is an enumeration that may be changed for optimizations or various design reasons, the numbers that represent the various operations are arbitary and the VHDL testbench will need a specific unit to convert the arbitrary signals into the ones that the circuit actually uses.

For a file format example, the following line from a generated input file will represent an addition between 00000001 and `ffffffff`.

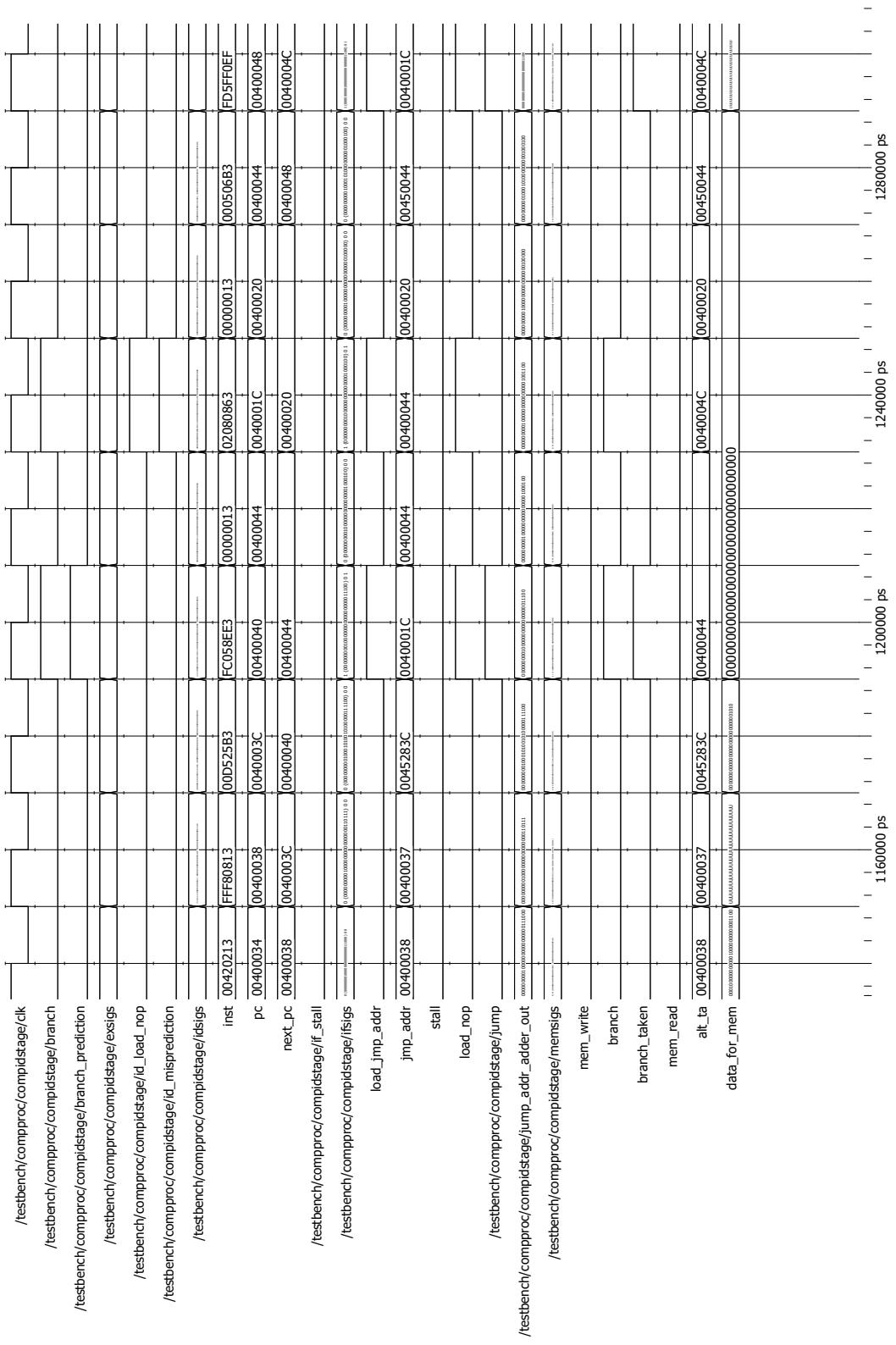

Figure 3.1: Instruction decoding stage signals showing the occurrence of a branch and the following flush due to its misprediction.

```
1    00000001 ffffffff 0
```

3.2.2 VHDL infrastructure

This testbench instantiates a whole ALU, provides data and control inputs and stores the outputs. At the same time, it instantiates a clock generator, an ALUFileReader and an ALUFileWriter.

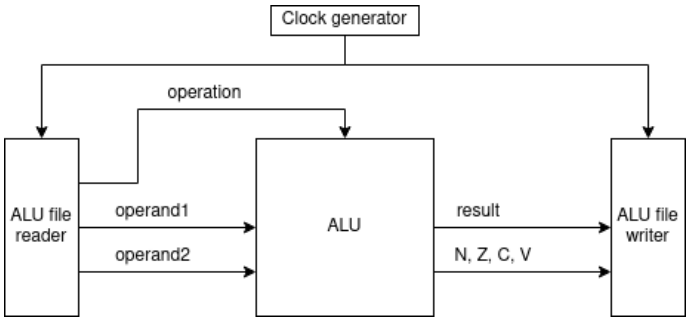

Figure 3.2: Structure of the testbench for ALU

As stated in subsection 3.2.1, ALUFileReader has to map the arbitrary opcodes into the local enumeration. This task is done through a function, described in the global package, that uses a switch to return the correct result.

3.2.3 C++ Reference

The software version of the ALU used to generate the reference results is implemented with a single C++ class called ALU. The declaration of this class, along with a brief description of its functionality, is present in section A.2.

By default, the compiled executable works in streaming mode: it continuously reads inputs from the standard input, performs the operations and prints the outputs to the standard out until the user interrupts the stream. This is useful for quickly testing specific input combinations by hand through the command line. It is possible to redirect the inputs or outputs from/to files to simulate arbitrarily large sequences of inputs.

3.2.4 Running the simulations

Operand specific simulations can be executed manually by generating input sequences to feed to the reference and to the simulation. Complete ALU simulations, instead, were scripted to simplify the whole process. It is possible to launch the script by simply specifying the amount of inputs and the automated process takes care of initializing the environment, generating the inputs, running the simulations and cleaning up. To get a decent coverage of the possible input combinations, plenty of simulations with input files whose size ranged from 50'000 to 100'000 lines each were launched over the span of multiple testing sessions.

Chapter 4

Synthesis

4.1 First version

	`compile`	`compile_ultra`
Area	15500	14800
T_{ck}	1.14 ns	1.10 ns

Table 4.1: Comparison between standard and ultra compile

Maximum clock frequency The basic design was synthesized with the retiming option enabled using both the standard compile followed by `optimize_registers` and `compile_ultra`. Table 4.1 shows that in this case the second option should be preferred. The compiler issues a warning when it applies retiming to registers that have both a preset and a reset signal, a situation that occurs with pipeline registers since they must be initialized at startup and reset synchronously during the execution as well (flush). Therefore, a simulation of the synthesized netlist is mandatory to verify that the retiming has not altered the functionality of the processor. This check was done on the testbench program provided (minimum search) with successful results.

Final synthesis After the zero clock period synthesis, a second compile command was issued with a slightly larger clock period constraint in order to obtain the final netlist to be routed. The results of this process are summarized in Table 4.2. The QOR report indicates that there are as many as 20 logic levels, this suggests that this design could benefit from a deeper pipeline, since it is well known that the ideal number of combinational logic levels for a processor is between 6 and 8.

The result from `report_resources` indicates that several DesignWare library components were used to synthesize comparators, incrementers and decrementers in the IF and ID stages. A barrel shifter was inferred in the ALU and an incrementer/decrementer in the branch prediction unit.

Area	14 644 µm^2
Combinational area	7438 µm^2
Noncombinational area	7205 µm^2
Clock period	1.16 ns
Critical path length	1.05 ns
Levels of logic	20

Table 4.2: Synthesis results

Place and route The gate count report shows the data reported in Table 4.3. The total area is slightly larger than estimated by Synopsys.

Gates	18903
Cells	7942
Area	15 084.9 µm^2

Table 4.3: Gate count from Innovus

The post-route timing report confirms that timing requirements are met with a positive slack equal to 0.017 ns in the worst case. The verify connectivity report confirms the successful completion.

4.2 Second version

The second version of the design reaches a minimum clock period equal to 1.2 ns. It was synthesized with a larger clock period equal to 1.4 ns with the results reported in Table 4.4 The overall area

Area	14 091 µm^2
Combinational area	6971 µm^2
Noncombinational area	7120 µm^2
Clock period	1.4 ns
Critical path length	1.29 ns
Levels of logic	23

Table 4.4: Synthesis results for the processor with extended ISA

and the number of gates are again summarized in Table 4.5. For this design, the looser timing constraint set during synthesis resulted in a smaller circuit as predicted by the area report in Synopsys.

Gates	18123
Cells	7062
Area	14 462 µm^2

Table 4.5: Gate count regarding the second version of the processor

Chapter 5

Conclusion

In conclusion, the result of this development process is a very performant, small and versatile processor. It can work with a clock frequency higher than 850 MHz while still minimizing stalls thanks to the branch prediction unit and forwarding unit.

As discussed in section 2.6, where the ABS instruction was introduced, the generality and elasticity of its components allow for minimal yet effective changes to support additional operations, or slightly change the behavior of the current ones.

Appendix A

Code Highlights

A.1 Recursive Adder Description

The code of this block is in the file `ALU/arithmetic/blockGPGenerator_LadnerFischer.vhd`, which is reported below:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity blockGPGenerator is
5      generic(N: positive := 32);
6      port (
7          g: in std_logic_vector(N-1 downto 0);       -- Generate vector
8          p: in std_logic_vector(N-1 downto 0);       -- Propagate vector
9          bg: out std_logic_vector(N-1 downto 0);     -- Block generate vector
10         bp: out std_logic_vector(N-1 downto 0)      -- Block propagate vector
11     );
12 end entity blockGPGenerator;
13
14 architecture LadnerFischer of blockGPGenerator is
15     component blockGPGenerator is
16         generic(N: positive := 32);
17         port (
18             g: in std_logic_vector(N-1 downto 0);       -- Generate vector
19             p: in std_logic_vector(N-1 downto 0);       -- Propagate vector
20             bg: out std_logic_vector(N-1 downto 0);     -- Block generate vector
21             bp: out std_logic_vector(N-1 downto 0)      -- Block propagate vector
22         );
23     end component;
24
25     component GPCombiner is
26         port (
27             gij: in std_logic;      -- Generate bit [i, j]
28             pij: in std_logic;      -- Propagate bit [i, j]
29             gjk: in std_logic;      -- Generate bit [j, k]
30             pjk: in std_logic;      -- Propagate bit [j, k]
31             gik: out std_logic;     -- Generate bit [i, k]
32             pik: out std_logic      -- Propagate bit [i, k]
33         );
34     end component;
35
36     signal internal_bg: std_logic_vector(N-1 downto 0);
37     signal internal_bp: std_logic_vector(N-1 downto 0);
38 begin
39     gen_base_net: if N = 1 generate
40         bg(0) <= g(0);
41         bp(0) <= p(0);
42     end generate gen_base_net;
43
44     gen_recursive_net: if N > 1 generate
45         comp_recursiveLowerHalf: blockGPGenerator
46             generic map (N/2)
47             port map (
48                 g => g(N/2 - 1 downto 0),
```

```
49                  p => p(N/2 - 1 downto 0),
50                  bg => internal_bg(N/2 - 1 downto 0),
51                  bp => internal_bp(N/2 - 1 downto 0)
52              );
53          comp_recursiveUpperHalf: blockGPGenerator
54              generic map (N/2)
55              port map (
56                  g => g(N-1 downto N/2),
57                  p => p(N-1 downto N/2),
58                  bg => internal_bg(N-1 downto N/2),
59                  bp => internal_bp(N-1 downto N/2)
60              );
61          -- Lower half goes through
62          bg(N/2 - 1 downto 0) <= internal_bg(N/2 - 1 downto 0);
63          bp(N/2 - 1 downto 0) <= internal_bp(N/2 - 1 downto 0);
64          -- Upper half is combined with the highest lower half
65          gen_combine_upperHalf: for i in N/2 to N-1 generate
66              comp_upperHalfCombiner_i: GPCombiner
67              port map (
68                  internal_bg(N/2 - 1), internal_bp(N/2 - 1),
69                  internal_bg(i), internal_bp(i),
70                  bg(i), bp(i)
71              );
72          end generate gen_combine_upperHalf;
73      end generate gen_recursive_net;
74  end architecture LadnerFischer;
```

It is possible to see how the recursion was acheived through a conditional generate statement: in the general case, considered at line 44, the component instantiates itself with half the bit-width. After some number of recursive instantiations, the bit-width finally reaches 1, triggering the other generate statement in line 39. This is the degenerate case where the Ladner-Fischer network is receiving a single bit and consists of a single input-to-output line.

A.2 Simulated ALU

The following is the header for the ALU simulation that contains the ALU class declaration.

```
#ifndef ALU_ALU_H
#define ALU_ALU_H

#include <iostream>
#include <cinttypes>
#include <exception>
#include <map>
#include <functional>

class ALU {
public:
    // Types
    // Must use unsigned integers because the sign bit is protected and overflow is
        not allowed
    using dtype = uint32_t;
    struct Input {
        dtype operand1;
        dtype operand2;
        int control;
    };
    struct Output {
        dtype result;
        int N;
        int Z;
        int C;
        int V;
    };
    using ALUFunction = std::function<dtype(const Input &)>;
private:
    // Static variables
    static const int N_INPUT_FIELDS = 3;
    // Attributes
    Input m_input;
    Output m_output;
public:
    ALU();
    // ALU state
    const Input &input() const;
    const Output &output() const;
    // Execution
    const Output &executeStep(const Input &input);
    const Output &executeStep(const std::string &instruction);
    void executeStream(std::istream &istream, std::ostream &ostream);
    // I/O operations
    static Input parseInputString(const std::string &instruction);
    static std::string formatOutputString(const Output &output);
    // Errors
    class InputFormatError : public std::runtime_error {
    public:
        InputFormatError(const std::string &msg);
    };
private:
    // ALU internal operations
    static dtype _add(const Input &input);
    static dtype _sub(const Input &input);
    static dtype _srai(const Input &input);
    static dtype _and(const Input &input);
    static dtype _xor(const Input &input);
    static dtype _lt(const Input &input);
    // Opcode to function mapper
    static const std::map<int, ALUFunction> functions;
};

#endif //ALU_ALU_H
```

To separate the specific instance of the class from the operation execution, all the supported operation are implemented as static function, which are automatically called when the `executeStream` function is used.

This function takes an input stream and an output stream as parameters. Input data is acquired, checked and converted from the input stream, then the requested function to execute is retrieved through the use of a static map, the function is collected and its generated result is written to the output stream.

It is also possible to use the same class to work on a single input through the `executeStep` whenever the user prefers to run a certain hardcoded pattern for testing.