Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering
Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano

Authors: group_8
Matteo Isoldi, Filippo Marostica, Elena Roncolino

September 19, 2023

Contents

CHAPTER 1

Introduction

The DLX is a fully pipelined processor with a RISC architecture derived from notorious predecessors (AMD 29K, DECstation 3100, HP 850, ...) which is based on the Harvard Architecture: two different memories are used for instructions and data, allowing simultaneous instruction-fetching and data transactions. It is also based on a Load/Store architecture, since all values must be loaded into registers before they can be used inside any computation. The objective of this project is to realize a functioning, DLX-based, microprocessor, from the knowledge acquired during the course. Three main phases have to be considered during its implementation: design, synthesis and physical layout design.

CHAPTER 2

CPU design

In this chapter, an introduction to the implemented CPU is given, analyzing its features and top level structure. Afterwards, the list of the supported instructions is provided, with simple syntax examples, for easily verifying the behaviour of the proposed hardware.

2.1 Architecture overview

The core was designed around the basic properties of the DLX microprocessor. It supports the usage of two distinct memories for data (DRAM) and instruction fetching (IRAM) while also maintaining a Load/Store architecture. Thus, all operations are performed on register operands or immediate values, hardcoded inside the instruction itself. Reading and writing to the memory requires the usage of special instructions: load (`lw`) and store (`sw`). This is to ensure more regularity between different operations and ease of decoding in order to increase the performances.

The top level structure is represented in Figure 2.1. As can be seen, two main blocks compose the CPU: the control unit and the datapath. The control unit is responsible for the correct handling of the instructions, by means of a set of FSMs that analyze and evaluate them, with the aim of producing the control signals required to accurately determine the flow inside the blocks which make up the Datapath. It is described thoroughly in Section 3.1. Thus, the datapath is made of components which acts on the input signals, manipulating them, to obtain a specific result, determined by the control unit. To increase the throughput of the processor, it is divided in five stages, to constitute a pipeline, so as to achieve parallel instructions handling. These stages are:

- Fetch (IF): this stage is tasked with retrieving a new instruction to execute from the IRAM while computing the address which will be read during the following clock cycle. This address is then stored inside the program counter register (PC);
- Decode (ID): during this phase, the instruction received from the IF stage is decoded, extracting the indexes of the registers to read from the register file (RF) and the immediate value used by some operations, like I-type and J-type instructions;
- Execute (EXE): all the operations on the requested operands are performed by this stage, inside the ALU block. This block contains every piece needed for computing arithmetical and logical results, like the Pentium 4 adder or the UltraSPARC T2 logic unit. A more accurate description of this block can be found in Section 3.4.
- Memory (MEM): in this stage, memory accesses, related to the load and store operations, are performed while the computed addresses tied to the branch operations are sent back to the fetch stage to set up the new flow of operations;

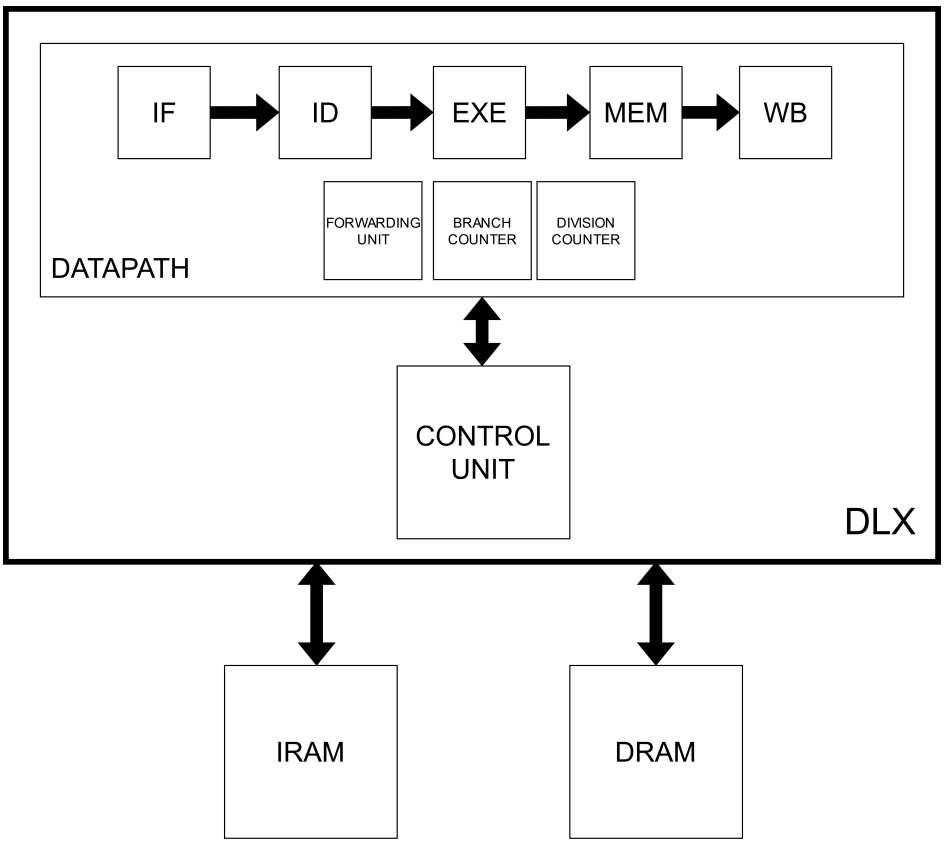

Figure 2.1: Top level CPU structure

- Write Back (WB): last stage of the pipeline. It sends to the register file the address and data to write before concluding the instruction's cycle.

The scheme of the Datapath can be found in Figure 2.2.

2.2 Optimization features

To further increase the processor's throughput, some features were implemented:

- a **forwarding unit**, which allows instructions to avoid waiting for their operands to be committed in the register file before starting their execution, has been realized. It makes use of multiplexers, located both in the ID and EXE stages, to redirect the outputs produced in the later stages as inputs for the stage's manipulations. The unit harnesses the information contained in the instruction registers to produce the control signals to correctly select the inputs of each multiplexer;

- an **untaken branch policy**, which allows to avoid wasting clock cycles waiting for the condition of the branch to be evaluated. The IRAM keeps retrieving instructions to be sent to the next stages and, if the branch has been predicted correctly, the flow of instructions won't be interrupted while, if the branch has been incorrectly forecasted, the instruction registers have their content resetted to a **NOP**, to avoid committing wrong instructions;

- a **multiplier unit** has been added, based on the Booth's algorithm, to perform signed multiplication on 64-bits;

- a **divider unit** has been added, based on a non-restoring algorithm, to perform signed division;
- the **instruction set** has been expanded, to support more instructions, like unsigned operations, additional logic instructions, division and multiplication. The full list is located in the next section (Section 2.3);
- an almost fully **structural datapath**, to create more efficient designs, in terms of delay and area usages. This impacts most of all the ALU block, since it contains very complex blocks.

2.3 Supported instructions

A list of all the supported instructions of the DLX microprocessor can be found in the next sections, to easily allow realizing test programs for checking the CPU behaviour. All of them work on 32-bits values and produce 32-bits results. `RS1`, `RS2` refer to the source registers while `RD` refers to the destination register. Their index can vary between 0 and 31 (i.e. `r0`, `r1`, ..., `r31`). `#Imm` indicates a 16-bits or 26-bits value, depending on the instruction, that can be inserted by the programmer inside the instruction, using a decimal or hexadecimal notation (other notations may be supported, based on the compiler used). For some instructions, this immediate value is sign-extended, while for others it is zero-extended. This is the case of unsigned instructions, denoted with a **u** letter, and the logic functions.

To properly compile the test programs, together with the report, it is delivered a compiler written in Perl (can be found in `./Compiler/dlxasm.pl`). **Note**: this is the **ONLY** one that can generate the proper instruction configurations, since the OPCODE and FUNCTION fields of the general instructions have been modified, while others have been created from scratch, to allow custom instructions (i.e. multlo, multhi, ...).

2.3.1 R-type instructions

In Table 2.1 all of R-type instructions are listed, together with a small description and a syntax example.

Operation	Description	Instruction
`add`	Signed addition	`add RD, RS1, RS2`
`addu`	Unsigned addition	`addu RD, RS1, RS2`
`sub`	Signed subtraction	`sub RD, RS1, RS2`
`subu`	Unsigned subtraction	`subu RD, RS1, RS2`
`and`	Bitwise AND	`and RD, RS1, RS2`
`nand`	Bitwise NAND	`nand RD, RS1, RS2`
`or`	Bitwise OR	`or RD, RS1, RS2`
`nor`	Bitwise NOR	`nor RD, RS1, RS2`
`xor`	Bitwise XOR	`xor RD, RS1, RS2`
`xnor`	Bitwise XNOR	`xnor RD, RS1, RS2`
`sll`	Logical left shift of A register by B shifts	`sll RD, RS1, RS2`
`srl`	Logical right shift of A register by B shifts	`srl RD, RS1, RS2`
`sra`	Arithmetic right shift of A register by B shifts	`sra RD, RS1, RS2`
`sgt`	Set DEST to one if A is greater than B	`sgt RD, RS1, RS2`
`sgtu`	Set DEST to one if A is greater than B (both A and B are unsigned)	`sgtu RD, RS1, RS2`
`sge`	Set DEST to one if A is greater or equal than B	`sge RD, RS1, RS2`

Continued on next page

`sgeu`	Set DEST to one if A is greater or equal than B (both A and B are unsigned)	`sgeu RD, RS1, RS2`
`seq`	Set DEST to one if A is equal to B	`seq RD, RS1, RS2`
`sne`	Set DEST to one if A is not equal to B	`sne RD, RS1, RS2`
`slt`	Set DEST to one if A is lower than B	`slt RD, RS1, RS2`
`sltu`	Set DEST to one if A is lower than than B (both A and B are unsigned)	`sltu RD, RS1, RS2`
`sle`	Set DEST to one if A is lower or equal to B	`sle RD, RS1, RS2`
`sleu`	Set DEST to one if A is lower or equal than than B (both A and B are unsigned)	`sleu RD, RS1, RS2`
`multlo`	Signed multiplication (outputs the last 32-bits of the result)	`multlo RD, RS1, RS2`
`multhi`	Signed multiplication (outputs the first 32-bits of the result)	`multhi RD, RS1, RS2`
`div`	Signed division	`div RD, RS1, RS2`

Table 2.1: R-type instructions

2.3.2 I-type instructions

In Table 2.2 all of I-type instructions are listed, together with a small description and a syntax example. Every unsigned instruction, denoted with a **u**, and all of the logic functions extend the 16-bits immediate value with all zeros, while all the other ones extend the value to 32-bits using the MSB.

Operation	**Description**	**Instruction**
`addi`	Signed addition	`addi RD, RS1, #Imm`
`addui`	Unsigned addition	`addui RD, RS1, #Imm`
`subi`	Signed subtraction	`subi RD, RS1, #Imm`
`subui`	Unsigned subtraction	`subui RD, RS1, #Imm`
`andi`	Bitwise AND	`andi RD, RS1, #Imm`
`nandi`	Bitwise NAND	`nandi RD, RS1, #Imm`
`ori`	Bitwise OR	`ori RD, RS1, #Imm`
`nori`	Bitwise NOR	`nori RD, RS1, #Imm`
`xori`	Bitwise XOR	`xori RD, RS1, #Imm`
`xnori`	Bitwise XNOR	`xnori RD, RS1, #Imm`
`slli`	Logical left shift of A register by `#Imm` shifts	`slli RD, RS1, #Imm`
`srli`	Logical right shift of A register by `#Imm` shifts	`srli RD, RS1, #Imm`
`srai`	Arithmetic left shift of A register by `#Imm` shifts	`srai RD, RS1, #Imm`
`sgti`	Set DEST to one if A is greater than a sign-extended immediate value	`sgti RD, RS1, #Imm`
`sgtui`	Set DEST to one if A is greater than a zero-extended immediate value	`sgtui RD, RS1, #Imm`
`sgei`	Set DEST to one if A is greater or equal than a sign-extended immediate value	`sgei RD, RS1, #Imm`
`sgeui`	Set DEST to one if A is greater or equal than a zero-extended immediate value	`sgeui RD, RS1, #Imm`

Continued on next page

`seqi`	Set DEST to one if A is equal to a sign-extended immediate value	`seqi RD, RS1, #Imm`
`snei`	Set DEST to one if A is not equal to a sign-extended immediate value	`snei RD, RS1, #Imm`
`slti`	Set DEST to one if A is lower than a sign-extended immediate value	`slti RD, RS1, #Imm`
`sltui`	Set DEST to one if A is lower than a zero-extended immediate value	`sltui RD, RS1, #Imm`
`slei`	Set DEST to one if A is lower or equal than a sign-extended immediate value	`slei RD, RS1, #Imm`
`sleui`	Set DEST to one if A is lower or equal than a zero-extended immediate value	`sleui RD, RS1, #Imm`
`multloi`	Signed multiplication (outputs the last 32-bits of the result)	`multloi RD, RS1, #Imm`
`multhii`	Signed multiplication (outputs the first 32-bits of the result)	`multhii RD, RS1, #Imm`
`divi`	Signed division	`divi RD, RS1, #Imm`
`lw`	Load data from memory	`lw RD, #Imm(RS1)`
`sw`	Store data inside the memory	`sw #Imm(RS1), RD`
`beqz`	Branch to target instruction if operand is equal to 0	`beqz RD, #Target`
`bnez`	Branch to target instruction if operand is not equal to 0	`bnez RD, #Target`

Table 2.2: I-type instructions

2.3.3 J-type instructions

In Table 2.3 all of J-type instructions are listed, together with a small description and a syntax example.

Operation	**Description**	**Instruction**
`j`	Branch to target instruction	`j #Target`
`jal`	Branch to target instruction while saving the address of the successive instruction in `r31`	`jal #Target`

Table 2.3: J-type instructions

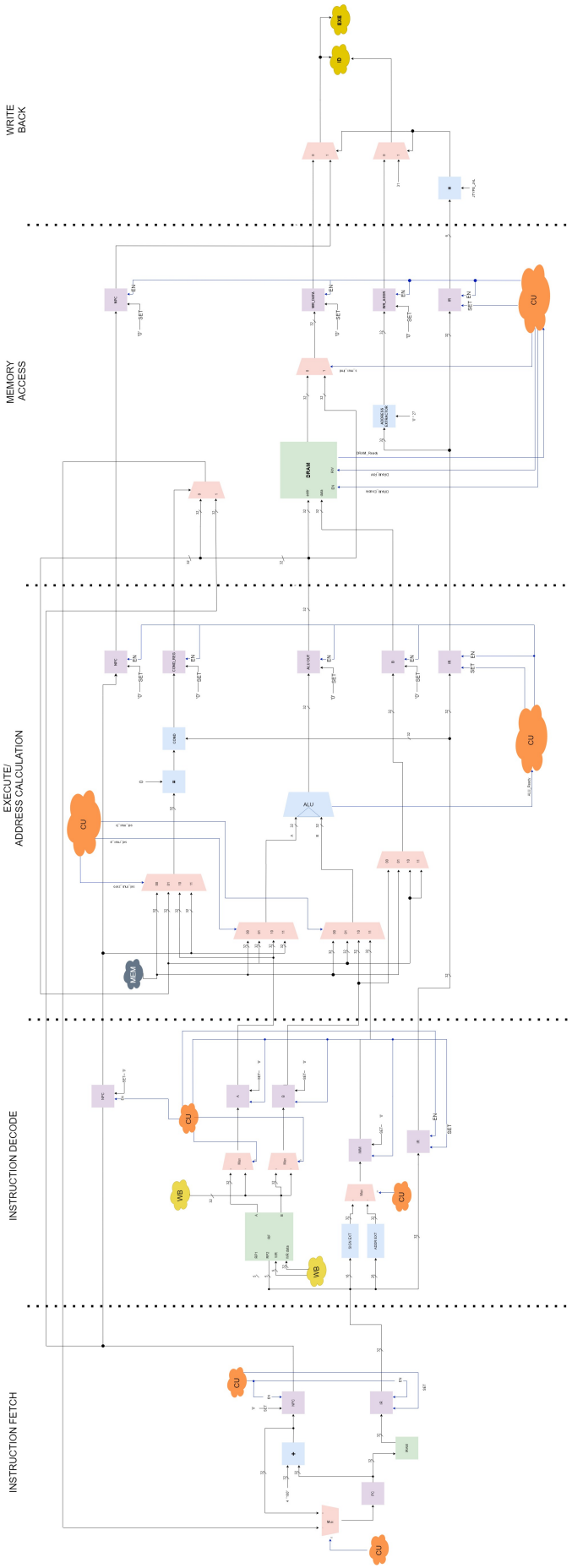

Figure 2.2: DLX datapath scheme.

CHAPTER 3

Blocks overview

In this chapter, a more in-depth analysis of the most relevant block composing the DLX architecture is offered. The two memories, IRAM and DRAM, even if important, are not described since their internal behaviour is either similar to that of the Register File or purely algorithmic, making the analysis repetitive and meaningless.

3.1 Control unit

The control unit is one of the most complex elements of the entire project. It allows the execution of concurrent instructions by analyzing and evaluating what has to be done in the different stages the datapath is divided in. This is due to four, independent, Mealy finite state machines that lie at its core, each associated to a single stage (except for the write back stage, which doesn't require one). They aren't managed in a hierarchical way but, instead, they signal to each other their status and the necessity of a stall in the pipeline, should an hazard arise. This impacts negatively the performances of the microprocessor, since a much longer delay is needed in order for the control signals to be stable, but, at the same time, allows it to be more reactive to occurring events, granting a faster response to avoid loss of data or incorrect data to proceed in the pipeline.

Each FSM has a state for managing the normal behaviour of the associated pipeline stage and a set of states for handling successive stages' requested stalls or current stage hazards. In particular, when a problem arises, the FSM tells the previous stage's FSM that its data have to remain unchanged for the instruction to be processed correctly when the problem will be solved, while it sends forward **NOP** operations, to avoid, in the meantime, any data to be computed and saved, either in the register file or memory. The previous FSM will change its state and, in turn, will tell to its previous stage's FSM the same thing and so on. When the hazard is no more present, the related FSM will change its state and communicate to the previous stage's FSM that the flow can continue as normal. The previous stage's FSM will tell the same thing to its predecessor and so on, until all the controllers have been updated of the new status. Stalls can occur when:

- the instruction memory takes more than one clock cycle to retrieve the next instruction;
- the forwarding unit detects an output-input relation between two subsequent instructions. In particular, this happens when a load follows some operation that requires a value stored in the memory to be correctly processed;
- the instruction takes more than one clock cycle for the ALU to produce its output (i.e. a division operation);
- the data memory takes more than one clock cycle to store or read data.

3.1.1 Untaken branch policy implementation

The detection of a branch instruction doesn't cause a stall in the pipeline, since these are managed using a untaken branch policy. Instead, the decode stage's FSM will change its state to a new one where a down counter, designated for these specific situations, is activated and observed. While the count has yet to reach zero, the instilled behaviour to the stage is similar to that of a non-branch instruction, allowing new instructions retrieved by the IF stage to be handled without problems. When it reaches zero, the state orders the program counter multiplexer to switch input, to let the computed address from the MEM stage be saved inside the program counter. This switch is commanded while the FSM changes state to the most appropriate one, since the branch situation has been fully handled. Afterwards, the used counter will be resetted, until a new branch has to be managed. This kind of management has two problems:

- since the counter is only one, it cannot allow two branches (of any type) to be placed sequentially in programs. The second one will always be disregarded by the DLX;
- during the time a branch operation is performed it is necessary, for obtaining correct results, that no hazards are inserted in the pipeline. Otherwise, some instructions won't be considered due to the inserted stalls, leading to incorrect results and unwanted flow;

The FSM doesn't handle the cleaning of the pipeline's instruction registers in case of a badly predicted condition. It is managed automatically by the Cond block's output, which will activate the **SET** of the instruction registers, forcing a series of **NOP** in the pipeline, to avoid committing mispredicted instructions' results. In case the branch was evaluated correctly the flow will continue without any problem.

3.1.2 Division management implementation

A similar approach to the untaken branch policy management has been implemented. A counter, specific to this situation, has been used to track how many cycles have passed since the division has been issued to the ALU block. While the counter has yet to reach zero, a stall in the pipeline is inserted, keeping the decod stage still and avoiding the IRAM to keep retrieving new instructions. To the MEM and WB stage, **NOP** operations are sent. When the counter reaches zero, the result of the division is pushed to the next stage and the previous stages can continue operating normally.

3.2 Register file

This small memory is at the base of every operation that is supported. It contains 32 registers, each with a width of 32-bits. It is characterized by an asynchronous read operation, performed by two independent ports, eventually at the same time, if both are enabled, and by a synchronous write operation, performed on one port only. Since its internal complexity, it was designed using a behaviour architecture, to allow more flexibility for the synthesizer during the optimization phase. In Figure 3.1, the block interface is represented.

3.3 Forwarding unit

The forwarding unit has been used to increase the performances of the pipeline. Its goal is to avoid putting on hold instructions which have their operands yet to be submitted inside the register file. To accomplish this task, the values calculated by the ALU and the MEM stages are forwarded back to several multiplexers placed inside the ID and EXE stages. These are used to choose which data, between the ones retrieved from the register file and the forwarded ones, has to be used as input. The

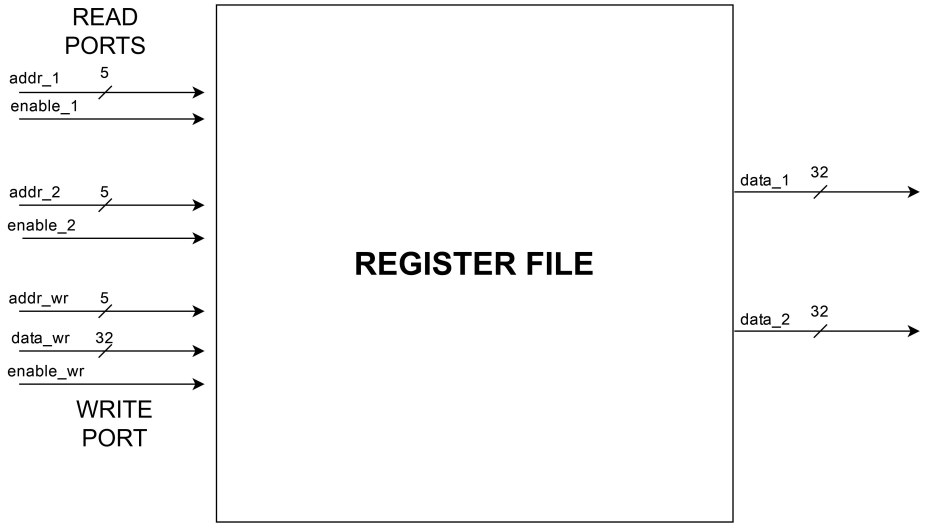

Figure 3.1: Register File component interface

selection for the multiplexers are determined by comparing the instructions' input operands received by the ID and EXE stages and the instructions' result operands in the EXE and MEM stages.

Beside the handling of the input operands for the ALU block, two critical situations that required careful design for this unit were:

- to prevent a stall caused by a request for an operand contained inside the register file in the same clock cycle where this is being overwritten by the write back's instruction. That same data is forwarded to two multiplexers, one for each input operand that can be retrieved from the RF, placed ahead of the A and B registers. In this way, the instruction doesn't have to wait the next clock cycle to retrieve the correct data but can directly recover it during its submission;
- to prevent forwarding to never happen when a load operation produces a value that is required by the following instruction. In this case, a stall has to be forced in order for the data to be forwarded when the instruction reaches the EXE stage, since it is taken from the data memory.

3.4 ALU layout

The ALU block, as can be seen in Figure 3.2, is composed of 7 main components:

- a **decoder**, used to interpret the selection signals sent by the control unit in order to forward to the other components the correct signals (i.e. selection for logic operation, arithmetic or logical shift, ...);
- a **Pentium 4 adder**, able to perform additions and subtractions between two 32-bits values with greater performances with respect to traditional blocks (i.e. ripple-carry adders). It is characterized by a Carry Look Ahead (CLA) Sparse Tree carry generator and by different carry select blocks operating on 4-bits each;
- a **comparator**, able to tell how two values relate to each other. It establishes its results on the **A - B** operation performed by the P4 adder;
- a three-stages **UltraSPARC T2 shifter**, able to generate 32-bits, logical and arithmetical, shifts on the input operand A based on some bits of input operand B;

- a **multiplier unit** based on Booth's algorithm, which reduces the delay of the whole multiplication process by halving the number of sums and by reducing the complexity of managing partial products using simple shift operations;
- an **UltraSPARC T2 logic unit**, based on a two-level combinational logic. It is able to perform AND, NAND, OR, NOR, XOR and XNOR operations between the two operands.
- a **divider** that uses a non-restoring algorithm to perform the division between two 32-bits operands.

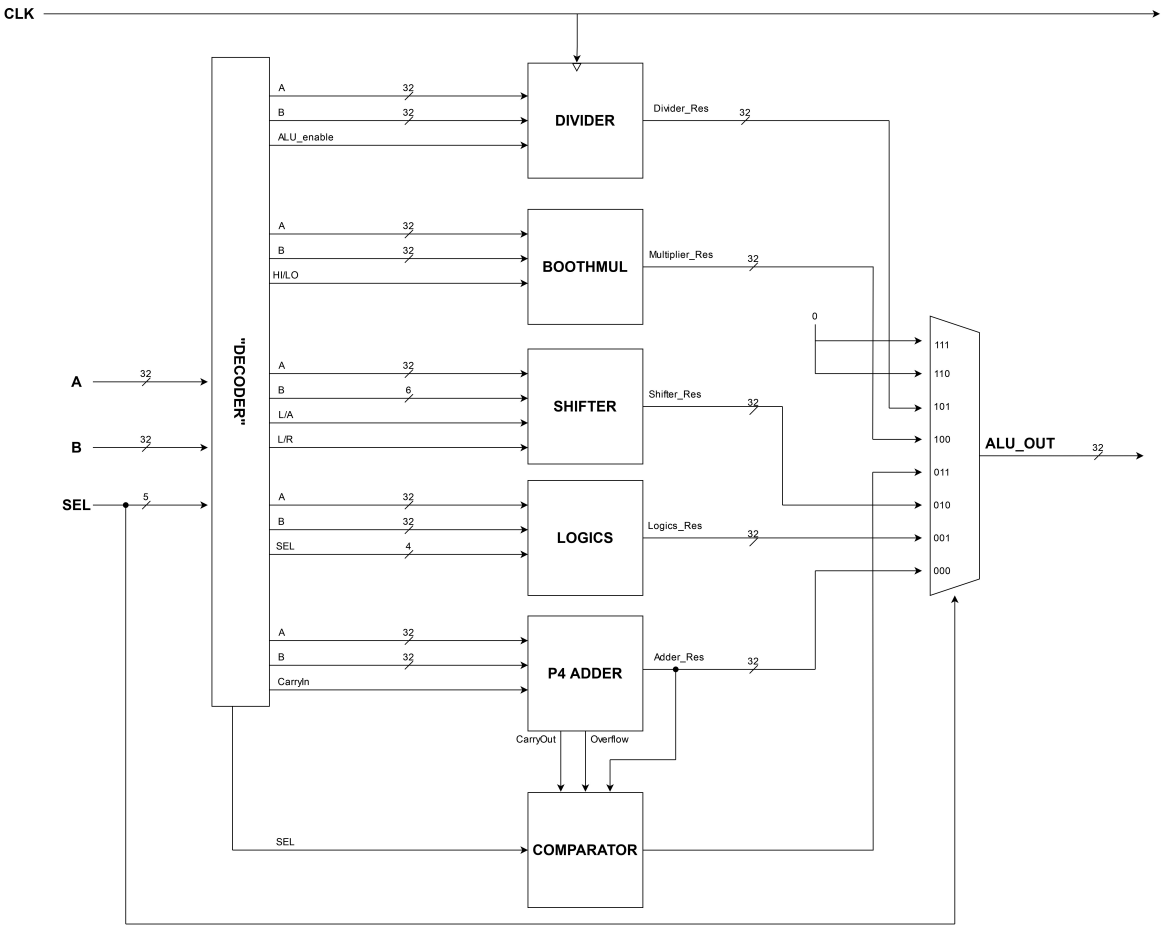

Figure 3.2: ALU internal structure

The decoder, while deciding which selection signals to generate and forward to the blocks, is designed to implement a power optimization strategy by applying to the different units input operands equal to 0 if they aren't used during that specific clock cycle. The rationale behind this is that, if for multiple clock cycles a specific unit is not requested, no transitions should happen inside that block and dynamic power consumption should lower. For example, if the P4 adder is not used for many clock cycles, the CLA and the sum generator should not have any of their blocks to transition inside since the inputs are maintained stable at value of 0, apart from the first clock cycle, when they are set to this value. Since the decoder is still a combinational block, transitions can still happen and this strategy may fail to reach its goal. For a better power optimization strategy, clock gating could have been used but at the cost of many more registers to save the data for the different arithmetical and logic units and more complexity in their management.

Given its complexity and the level of flexibility required to continue adding support for new operations, it has been realized using a behavioural description.

3.4.1 Pentium 4 adder

The adder has been realized following the Pentium 4 (P4) structure in order to achieve better performances on widely used operations, such as additions and subtractions. It operates on 32-bits values and generates a 32-bits result as well as a carry out and an overflow flag, to identify incorrect results due to the high positive or negative values summed. The general schema can be seen in Figure 3.3, where the two main blocks are highlighted.

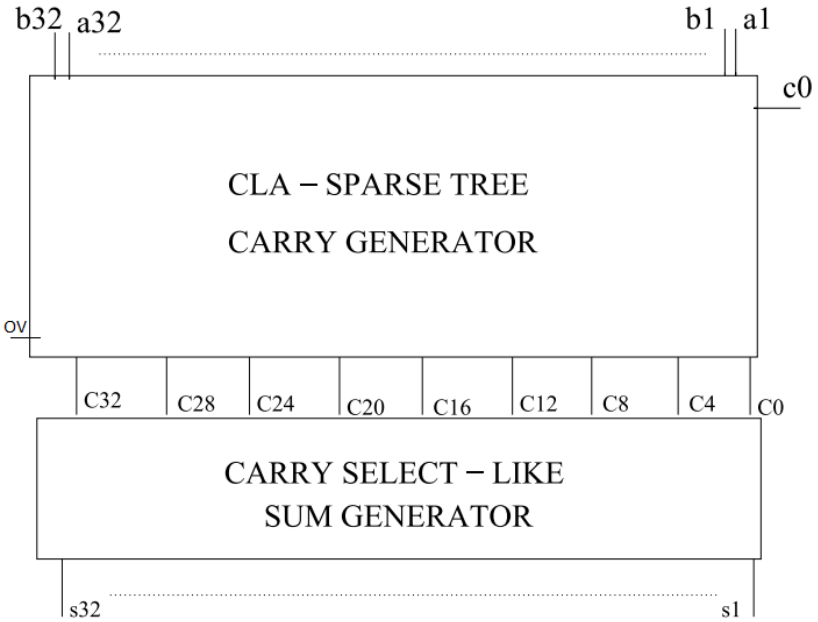

Figure 3.3: Pentium 4 adder structure

The adder is based on a sparse radix-2 carry merge tree (also called Sparse Tree) that generates every fourth carry. This allows to increase the performances by reducing the critical path of the CLA. It consists of a PG (propagate and generate) stage, followed by five stages of carry merge logic, as can be seen in Figure 3.4. Starting from this structure, some modifications were made to allow the production of the overflow flag, used by the comparator to better determine the relation between the operands. Based on the formula for its computation:

$$\mathrm{OV} = \mathrm{C31} \oplus \mathrm{C32}$$

the 31-st carry is needed. Since, because of the structure of Sparse Tree CLA, this is not calculated, a small propagate-generate network has been realized that, from the 28-th carry, predicts its value. This modification lengthens the path of the adder unit but not by an amount which negate the effect of harnessing the P4 layout.

The sum generator is based on eight carry select structures, each operating on four bits. Every block is composed of two ripple carry adders that sum the bits of the operands under two different carry in values. The computed values become the inputs of a multiplexer, which will output later the correct value based on the selection made by the carry bit generated by the CLA.

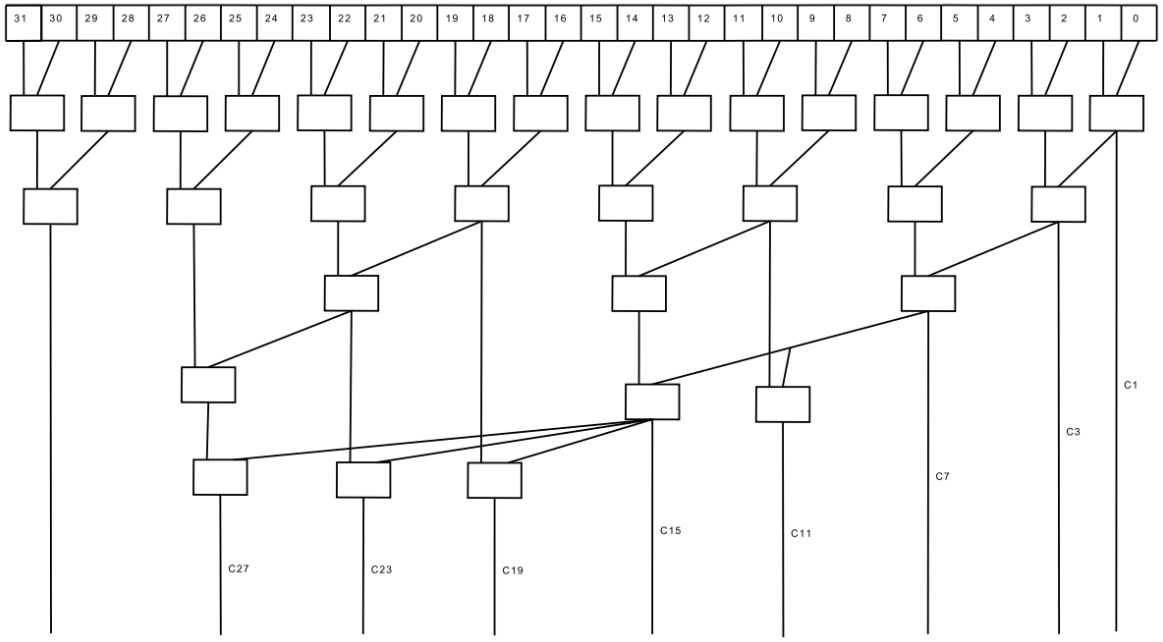

Figure 3.4: Sparse Tree Carry Look Ahead structure

3.4.2 Comparator

The comparison between two values A and B exploits the result of the difference between A and B coming from the P4 adder. It also takes under consideration whether the difference between the two numbers has caused an overflow. The basic idea behind the comparator is the following:

$$A > B \quad \Rightarrow \quad C \; and \; \overline{Z}$$

$$A > B \quad \Rightarrow \quad C \; and \; \overline{Z}$$

$$A >= B \quad \Rightarrow \quad C$$

$$A < B \quad \Rightarrow \quad \overline{C}$$

$$A <= B \quad \Rightarrow \quad \overline{C} \; and \; Z$$

$$A = B \quad \Rightarrow \quad Z$$

$$A \neq B \quad \Rightarrow \quad \overline{Z}$$

where A and B are the two numbers to be compared, C is the carry out and Z is the overflow flag, computed using the formula described in Section 3.4.1 (both C and Z come from the P4 adder). If this value is a logic 1, then the carry out value is inverted before performing the other calculations.

The structure of the comparator is shown in Fig. 3.5.

3.4.3 UltraSPARC T2 logic unit

The logic unit is based on the structure of the UltraSPARC T2 logic unit. It is composed of two NAND gates level: the first level is composed of four 3-input NAND gates, where each input is 32 bits long; while the second level is composed of a NAND that has as inputs the four outputs of the previous level, as shown in Fig. 3.6. Each first level gate has the R1 and R2 operands as inputs (or their negated value) and a selection signal (S0, S1, S2 or S3) extended on 32 bits. To generate the requested logic operation the signals have to be set as shown in Table 3.1:

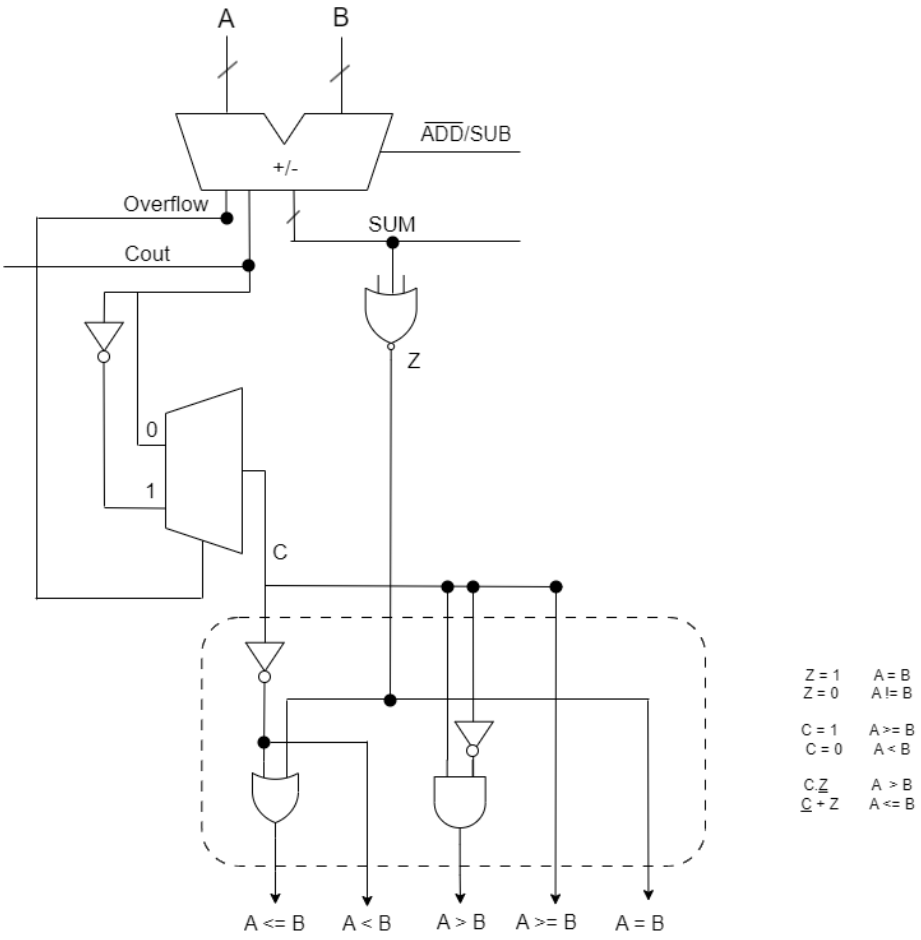

Figure 3.5: Comparator structure

S0	S1	S2	S3	Function
0	0	0	1	AND
1	1	1	0	NAND
0	1	1	1	OR
1	0	0	0	NOR
0	1	1	0	XOR
1	0	0	1	XNOR

Table 3.1: Signals for the logic unit.

3.4.4 Divider

The divider component implements the division between two signed operands A and B and gives the result Q in signed form. The algorithm used is non-restoring, with a redundant alphabet that assigns for each q_i in Q either the value $\overline{1}$ or 1. With respect to the restoring version, this algorithm is more complex but faster, since for each quotient bit there is only one decision, i.e., whether to choose $\overline{1}$ or 1 as q_i, and therefore only one addition or subtraction is necessary. The algorithm itself works with positive values only, but, at the beginning of the division, the signs of the two operands are stored to later transform the final quotient and remainder into their correct signed form. Of course,

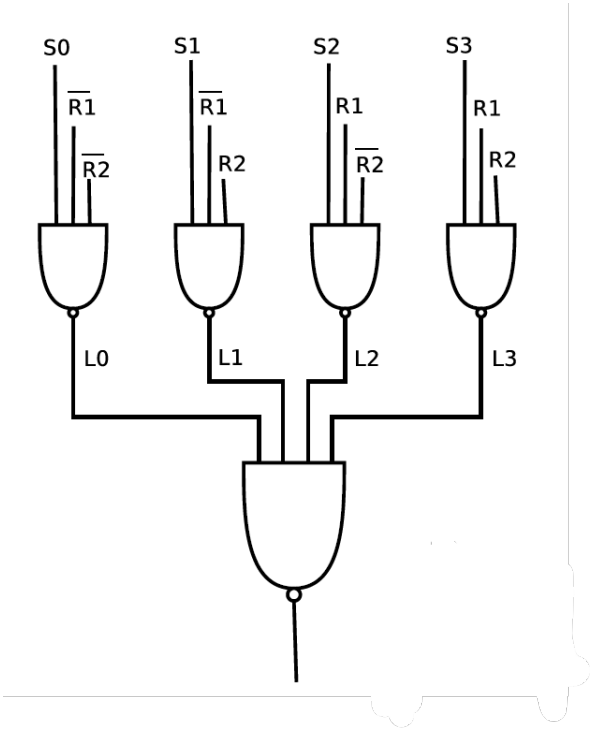

Figure 3.6: Logic unit structure

the better choice would have been the SRT radix-2 algorithm, but unfortunately, due to the scarcity of documentation regarding the normalization needed for the operands in order for the algorithm to work, that path had to be abandoned. Listing 3.1 shows how the non-restoring algorithm has been implemented while Table 3.2 shows an example of the procedure to retrieve the final value of Q.

```
R := A
B := B << n

FOR i IN n-1 TO 0 LOOP
    IF R >= 0 THEN
        Q(i) := 1
        R := 2 * R - D
    ELSE
        Q(i) := \overline{1}
         R := 2 * R + D
    END IF
END LOOP

-- A and B = the two operands, n = #bits of the operands
-- R = partial remainder, Q(i) = bit #i of the quotient.
-- Note: R and B need to be 2*n bits wide.
```

Listing 3.1: Non-restoring division algorithm.

All quotients computed by this algorithm are odd and the remainder is in the range $-D \leq R < D$. To convert to a positive remainder, it is then necessary to do a single restore after the final value of Q has been calculated, as shown in Listing 3.2. This value, though, is shifted left by n positions, so, to obtain the final remainder, in case it is of interest, it is necessary to shift right by n positions.

Starting point	$Q = 111\bar{1}1\bar{1}1\bar{1}$
Positive term	$P = 11101010$
Negative term	$N = 00010101$
Subtract P and N	$Q = 11010101$

Table 3.2: Example how to calculate Q in standard form.

```
IF R < 0 THEN
    Q := Q - 1
    R := R + D
END IF
```

Listing 3.2: Restore step to compute final Q and R.

Note: performing a division between 0 and 0 will yield a wrong quotient, so it is recommended to be sure to avoid that case.

3.4.5 UltraSPARC T2 shifter

The shifter component is based on the one used inside the UltraSPARC T2 microprocessor with some minor changes, to better adapt it to the used parallelism. It receives as input two operands: R1, a 32-bits operand to be shifted, and R2, a 6-bits value, used for defining the shift amount. Other two inputs are used to define if the shift should be left or right and if it should be arithmetical or logical. Its top level organization can be seen in Figure 3.7. The unit is organized on three levels:

1. it consists in preparing four possible *masks*, each representing R1 shifted of 0, 8, 16 and 24 positions, left or right depending on the provided configuration;
2. it performs a coarse grain shift: it chooses among the four *masks* the nearest one to the final result. The choice is operated using bit the first three bits operand R2 (5:3);
3. it receives the previously selected mask and it implements the real shift according to the last three bits of operand R2 (2:0).

The main difference with respect to the original is the number of generated masks and, as a consequence, the maximum shift amount. Since the DLX works with a 32-bits input, using eight masks would not be necessary, since four are enough to cover all the possible shifts (from 0 to 32 positions).

3.4.6 Booth multiplier

The multiplication unit is based on Booth's algorithm, thus the name BoothMul. With respect to the basic implementation of the multiplier, it highlights an important improvement in terms of performance, due to the reduction of the number of partial products by half, made possible by the technique of radix-4 Booth recording. The general structure, with a parallelism of 8 bits, of the multiplier unit is represented in Figure 3.8. The difference between this implementation and our implementation is the number of bits of the operands and the number of multiplexers, adder and encoder generated to compute all the necessary partial product.

The flow of operations of the BoothMul is the following:

1. two signed numbers are inputted to the unit, respectively named multiplier and multiplicand;

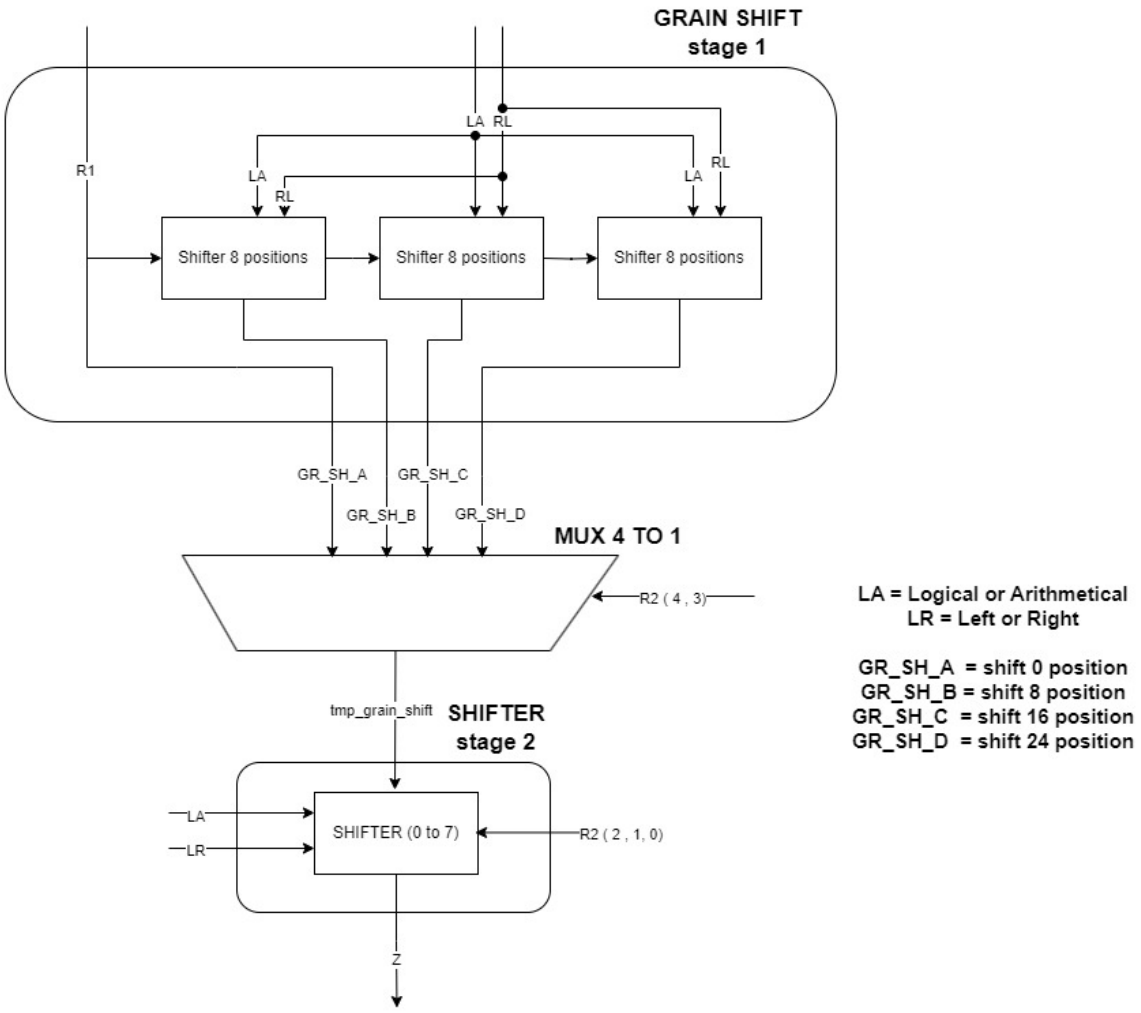

Figure 3.7: Shifter structure

2. an external component uses the multiplicand to create a set of masks that compose the inputs of the multiplexer;
3. during each cycle, the multiplier is scanned and, using 3 bits at a time, it is calculated the value of the multiplexer selector;
4. once the multiplexer's output is ready, it is sent to the adder component;
5. at every step, partial additions are calculated. Operations 2, 3 and 4 are repeated until all the bits of the multiplier have been scanned.

The multiplier implementation used in the DLX multiplies two signed values on 32 bits, generating a result on 64 bits. Since the parallelism of the processor is based on 32 bits, two instructions have been supported:

- multhi: it returns the first 32 bits of the resulting value (63:32);
- multlo: it returns the last 32 bits of the resulting value (31:0).

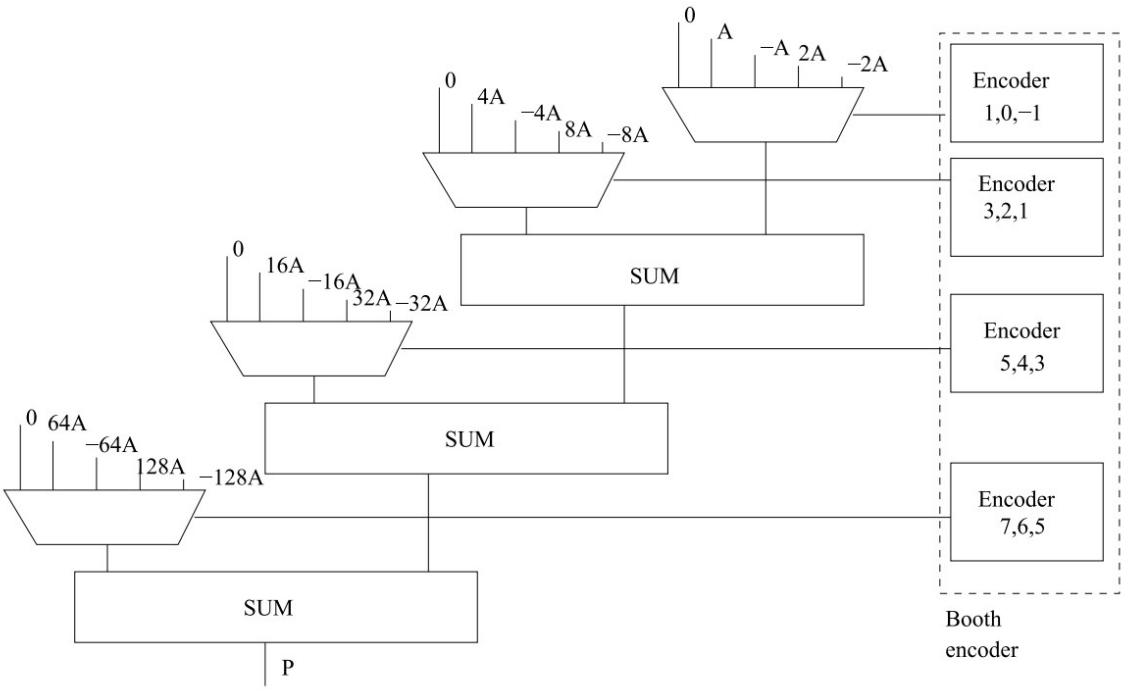

Figure 3.8: Booth multiplier structure 8 bits

3.5 Cond block

The Cond block is a critical component within the EXE stage. It is tasked with evaluating the requests for a jump, made by branching instructions. Depending on the instruction received and the value produced by the Zeros block (which checks if the input is equal to zero), it generates a single bit value: a **logic one** if the jump should be performed, a **logic 0** otherwise. It consists in a cascade of blocks, called IR_COMPARATOR, that are used to test the equivalence between the OPCODE of the instruction that the processor is executing and the OPCODE of four different type of branch instructions:

- ITYPE_BEQZ;
- ITYPE_BNEZ;
- JTYPE_J;
- JTYPE_JAL;

When the comparisons finish, the results go into a cascade of combinational logic to obtain the correct output value. This structure can be seen in Figure 3.9.

This block's output is used to activate the **SET** signals of the instruction registers, hence its criticality. If the result is wrong, the flow might be mismanaged, involving loss of data or wrong result committed inside the RF.

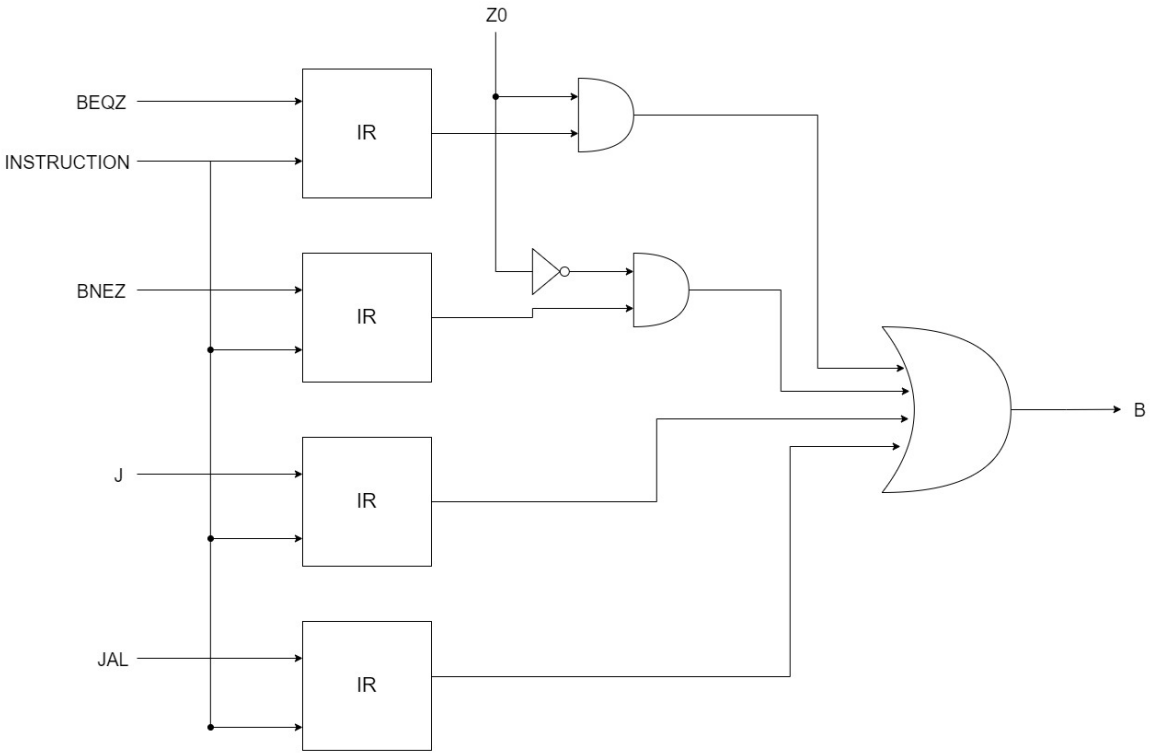

Figure 3.9: Cond block structure

CHAPTER 4

Synthesis

In this chapter, the results of the synthesis process are discussed and compared with the aim of evaluating how different parameters impacted them. First, though, a brief explanation of the applied strategy is given, to better explain how to replicate the obtained results.

4.1 Adopted strategy

The optimization phase's objective has been to analyze the performance parameters of the design, such as maximum delay, power consumption and area occupation, in presence of different clock signal configurations. To start defining its period, a first, unconstrained compilation of the architecture has been performed. The maximum delay obtained from this elaboration regards the path that, from the start registers of the write back stage, reaches the output port *Result* of the processor. This is due to the behaviour of the *compile* command, which always tries to measure the time taken from a register to an output port, if no conditions have been specified. The elaborated result is 0.73 ns.

Starting from this value, many compilations have been performed, with a growing period assigned to the clock signal, since the retrieved time represents one of the simplest stages, while the other ones require much more time to produce stable signals to their outputs. The included periods are reported in Table 4.1. Three compilations for each of these periods of the clock signal have been processed:

- an unconstrained one, to later compare how much the optimizations affected the final results;
- a constrained one, where to the synthesizer is asked to perform as much time, area and power optimizations as possible (constraints declared through the usage of *-map_effort high*, *-area_effort*

Period	**Frequency**
0.73 ns	1.37 GHz
1 ns	1 GHz
2 ns	500 MHz
5 ns	250 MHz
8 ns	125 MHz
10 ns	100 MHz
100 ns	10 MHz
1 μs	1 MHz
10 μs	100 KHz

Table 4.1: Periods and frequencies analyzed

high and *-power_effort high* parameters);

- a second constrained elaboration, based on the same constraints as the previous one and the additional implementation of the clock gating technique, to further reduce dynamic power consumption.

Each of these compilations is performed under the additional imposed rule of avoiding to modify too much the provided designs. This is commanded by specifying the *-ungroup_all* parameter. The final script can be found in Appendix A.1.

After evaluating the entire design, a brief focus has been placed upon the ALU, to verify that the included decoder really had an impact on dynamic power consumption. A second version of the block had been prepared and two compilations (one for each unit) have been conducted, to compare their total power consumption and, more specifically, their dynamic power consumption. The script used can be found in Appendix A.2.

4.2 Conclusions

As presumed, the timing delay acquired by not applying any restrictions on the designs is not enough for the other stages to cover their respective critical paths. The slack (difference from required time and arrived time of the signals at the output ports) is negative and it is equal to $-6.14\ ns$. From this result, it can be understood that neither 1, 2 or 5 ns are enough to be used to power the architecture properly.

The period of the clock that first grants a non-negative slack is 8 ns, with no real differences in the slack values of the three compilations. So, the designed DLX can work at 125 MHz to maximize the throughput. Increasing the clock period further doesn't produce any interesting results in terms of timing analysis. The only thing worth noting is that, in these cases, the data arrival time increases as a consequence of optimizing other parameters, prompting a lengthening of the critical paths. What is relevant for these configurations is how much the area and power are impacted. A showcase of the areas, unconstrained (**U**) and constrained (**C**), is shown in Table 4.2. The elaboration processed using the clock gating technique is not present because its results are only slightly different from the constrained implementation. From the table, it can be seen that lowering the frequency

Frequency $[MHz]$	**Area usage (U)** $[\mu m^2]$	**Area usage (C)** $[\mu m^2]$
125	26981.70	27092.89
100	26237.17	26260.84
10	26154.18	26171.20
1	26141.68	26153.91
0.1	26210.30	26225.47

Table 4.2: Area usage with different clock periods

and, as a consequence, permitting more lengthy critical paths allowed to reduce the number of cells. Transitioning from 125 MHz to 100 MHz impacted most the design, while the other frequencies allowed only small optimizations.

The influence of the elaborations on the power consumption can be found in Table 4.3. Here, lowering the used frequency has a bigger impact than what has been seen for area usage. Reducing it by an order of magnitude, in fact, cut the used energy off by almost four times. Inside Table 4.4, a comparison between the dynamic power consumption of the design in presence of the clock gating technique and the energy consumption when this method is not employed is portrayed. From this new data, it can be assumed that the total power consumption is mostly due to leakage (static power

Frequency $[MHz]$	**Total power consumption (U)** $[mW]$	**Total power consumption (C)** $[mW]$
125	2.0904	2.1338
100	1.7730	1.7976
10	0.7097	0.7128
1	0.6025	0.6030
0.1	0.5911	0.5927

Table 4.3: Total power consumption related to different clock periods

Frequency $[MHz]$	**Dynamic power consumption** $[mW]$	**Dynamic power consumption (CG)** $[mW]$
125	1.5201	1.4820
100	1.2047	1.2111
10	0.1203	0.1205
1	0.0119	0.0119
0.1	0.0011	0.0011

Table 4.4: Dynamic power consumption comparison

consumption), because, with the frequency decreasing, the switching operations of the transistors decrease as well, drastically reducing their dynamic power consumption, as can be confirmed from Table 4.4. Every order of magnitude removed from the frequency is also removed from the energy consumption. This confirms that their relation is linear, as suggested by the formula:

$$P_{dynamic} = \alpha \cdot C_{load} \cdot V_{dd}^2 \cdot f$$

where α is the switching activity of the gate, C_{load} is the load seen by the transistor and V_{dd} the power voltage.

Note on the constrained results:
All the tables display a flaw of the synthesis process: the values obtained from the constrained optimizations are always higher than when no constraint is applied. It is believed to be due to not specifying any threshold for power or area while using the compilation commands. The synthesizer tries to enhance as much as possible the design but, since it has no limits, it goes above the obtained value from the unconstrained elaborations. Even so, these were not the expected results.

4.2.1 Consequences of optimizing the ALU

In Table 4.5, the comparison between dynamic energy consumption of the ALU, in presence or not of the decoder optimizations, is presented. From the data, it can be assumed that the decoder, as already

	Dynamic power consumption (U) $[mW]$	**Dynamic power consumption (C)** $[mW]$
Standard ALU	2.5265	2.3404
Optimized ALU	2.5265	2.6627

Table 4.5: ALU decoder power analysis

theorized in Section 3.4, didn't serve well its purpose due to being a combinational block. To grant a lower consumption by using the clock gating technique, registers have to be placed ahead of each

unit contained in the ALU, using the outputs of the decoder itself and some enable signals to allow writing new data. In this way, data would be much more stable and even small transitions happening on the inputs would not be transmitted over to the outputs. This implementation, though, has an impact on time performances since it requires an additional clock cycle to complete an instruction. The alternative would be to apply the clock gating technique to the A and B registers between the ID and EXE stages, but it would not have the same effect since all the units would operate whenever these two signals change, causing a smaller reduction in power consumption.

CHAPTER 5

Physical Design

In this chapter, the results of the physical placing and routing process are discussed and shown.

5.1 Placing and routing

This represents the last step of designing the processor. The chosen design is based on 125 MHz frequency with clock gating applied by the synthesizer, since it had the best performances.

The first design that was produced had to be discarded due to some geometry violations, caused by the wrong placement of some pins. The violations, shown in Figure 5.1, are highlighted by a white cross. The solution adopted to correct this problems consisted in a better distribution of the pins

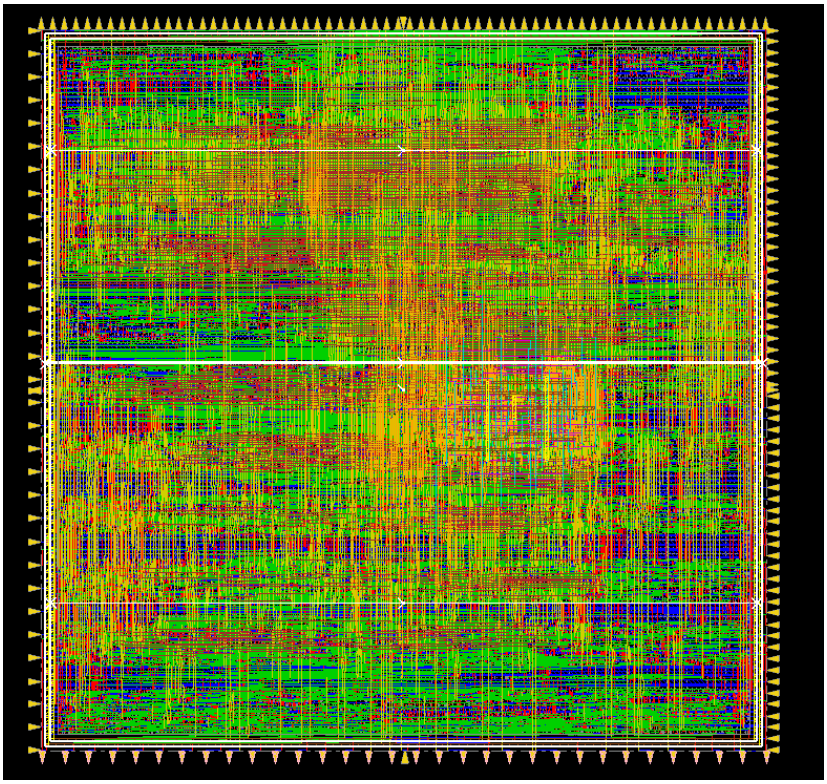

Figure 5.1: Shorts marked with a white cross.

along the four sides of the circuit. After different attempts, the final design is the one shown in Figure 5.3, while Figure 5.2 shows the log written by the program, which reports that no violations were found.

```
innovus 2> VERIFY_CONNECTIVITY use new engine.

******** Start: VERIFY CONNECTIVITY ********
Start Time: Tue Sep 19 10:51:32 2023

Design Name: DLX
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (225.9100, 221.4800)
Error Limit = 1000; Warning Limit = 50
Check all nets
**** 10:51:33 **** Processed 5000 nets.
**** 10:51:33 **** Processed 10000 nets.
**** 10:51:34 **** Processed 15000 nets.

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue Sep 19 10:51:34 2023
Time Elapsed: 0:00:02.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:01.3  MEM: 0.000M)
```

Figure 5.2: Verify Connectivity command showing there are no violations.

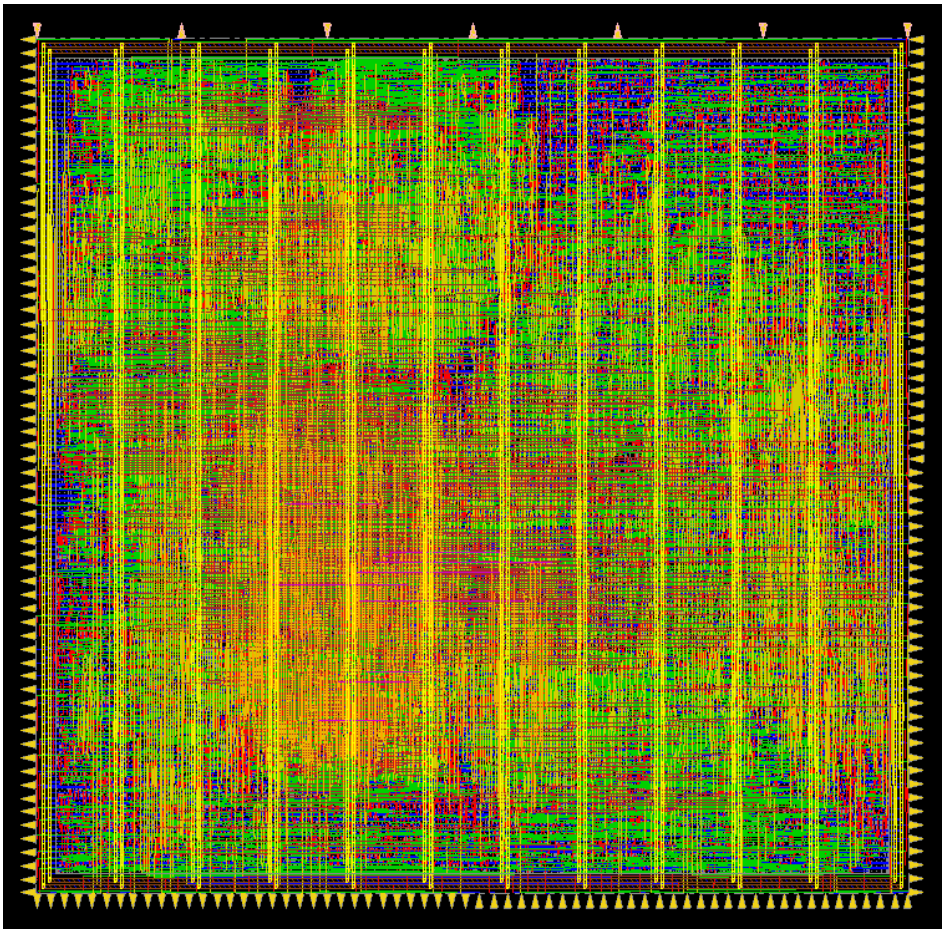

Figure 5.3: Final design.

The setup and hold time analysis reports that there are no paths that violate the constraints, as shown in Figure 5.4.

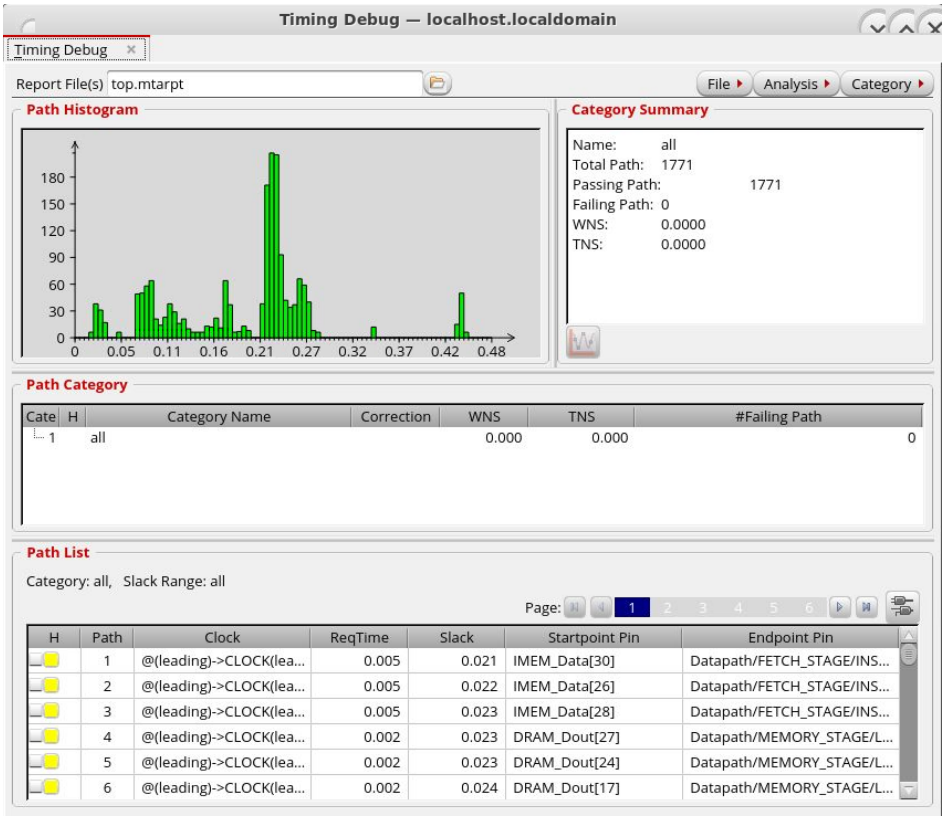

Figure 5.4: No timing violations are present in the design.

CHAPTER 6

Conclusions

The designed processor represents a starting point to developing a real, RISC based, CPU. As it is now, it supports a full five stage pipeline with the aid of a forwarding unit and the implementation of an untaken branch policy. It is able to fully perform addition, subtraction, multiplication and division, as well as all the operations that are based on them. Even so, its throughput could be further enhanced by managing more hazards and expanding the instruction set, thus preventing stalls from happening or avoiding too many instructions to perform a single task. Some ideas to achieve this goal are:

- the introduction of a **branch prediction unit**, to prevent hazards related to branch instructions (i.e. `beqz`, `bnez`);
- the usage of a **windowed register file**, to improve the performances of subroutine based program, given its support to the context switching operation;
- **using techniques to reduce power consumption from the architectural point of view**, like parallelization, isolation, clock gating, ...;
- implementing a **cache** to reduce the time taken from transferring data between memory and register file;
- adding **hardware related to instruction computation**, to expand the instruction set supported by the processor.

APPENDIX A

Synthesis scripts

A.1 Script for compilation of the DLX architecture

```
set PeriodList [list 0.73 1 2 5 7 8 10 100 1000 10000]
compile -exact_map
report_timing > reports/time/DLX_time_no_opt.rpt
report_area > reports/area/DLX_area_no_opt.rpt
report_power > reports/power/DLX_power_no_opt.rpt

foreach Period $PeriodList {

    # Define a clock to constrain the combinational network between registers
    # Start with the timing provided by the unconstrained elaboration
    create_clock -name "CLOCK" -period $Period clock

    # Check if clock signal is created correctly
    report_clock > "reports/clock/DLX_clocksignal_{$Period}.txt"

    # Compile after setting the clock
    compile -exact_map -ungroup_all
    report_timing > "reports/time/DLX_time_opt_clock_{$Period}.rpt"
    report_area > "reports/area/DLX_area_opt_clock_{$Period}.rpt"
    report_power > "reports/power/DLX_power_opt_clock_{$Period}.rpt"

    # Squeeze every optimization possible from Synopsys, with the same clock
    compile -ungroup_all -map_effort high -power_effort high -area_effort
    high
    report_timing > "reports/time/DLX_time_opt_max_clock_{$Period}.rpt"
    report_area > "reports/area/DLX_area_opt_max_clock_{$Period}.rpt"
    report_power > "reports/power/DLX_power_opt_max_clock_{$Period}.rpt"

    # Add the clock gating to techinique to increase power performances
    compile -ungroup_all -map_effort high -power_effort high -area_effort
    high -gate_clock

    #### Continued in the next page ####
```

```
    report_timing > "reports/time/DLX_time_opt_gate_clock_{$Period}.rpt"
    report_area > "reports/area/DLX_area_opt_gate_clock_{$Period}.rpt"
    report_power > "reports/power/DLX_power_opt_gate_clock_{$Period}.rpt"

    # Only when best optimization
    if {$Period >= 7  && $Period <= 10} {
        #Generate Verilog file for physical design
        write -hierarchy -f verilog -output reports/verilog/DLX_{$Period}.v

        #Generate Synopsis Design Constraint file
        write_sdc reports/verilog/DLX_constraint_{$Period}.sdc
    }
}
```

Listing A.1: Optimization script

A.2 Script for ALU compilation

```
for { set i 0 } { $i <= 1 } { incr i } {
    # Analyze the designs
    analyze ...
    if { $i == 0 } {
        # Optimized ALU
        analyze -library WORK -format vhdl {../ALU.vhd}
    } else {
        # Standard ALU
        analyze -library WORK -format vhdl {../ALU_no_opt_power.vhd}
    }

    # Elaborate the ALU
    elaborate ALU -architecture Structural

    # Clock period to check
    set Period 8
    create_clock -name "CLOCK" -period $Period clock

    # Compile after setting the clock
    compile -exact_map -ungroup_all
    report_timing > "reports/time/DLX_time_opt_clock_{$i}_{$Period}.rpt"
    report_area > "reports/area/DLX_area_opt_clock_{$i}_{$Period}.rpt"
    report_power > "reports/power/DLX_power_opt_clock_{$i}_{$Period}.rpt"

    # Add the clock gating to techinique to increase power performances
    compile -ungroup_all -map_effort high -power_effort high -area_effort
    high
    report_timing > "reports/time/DLX_time_opt_max_clock_{$i}_{$Period}.rpt"
    report_area > "reports/area/DLX_area_opt_max_clock_{$i}_{$Period}.rpt"
    report_power > "reports/power/DLX_power_opt_max_clock_{$i}_{$Period}.rpt"

    remove_design -designs
}
```

Listing A.2: ALU compilation script