Politecnico di Torino

Department of Electronics and Telecommunications

Microelectronic Systems
01NOYOQ

DLX Microprocessor
Design & Development

Final Project Report

Master degree in Electronics Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Group: ms23.32

Author: Riccardo Cuccu

Student Number: s253986

September 19, 2023

Contents

Summary

This project involves the engineering and deployment of an enhanced DLX-based processor, originally conceptualized by D. Patterson and J. Hennessy. Compared to the DLX-basic version, this enhanced version incorporates several advancements:

- **Extended Instruction Set**: inclusion of an additional 25 instructions, namely: addu, addui, subu, subui, mult, multu, sra, srai, seq, seqi, slt, sltu, slti, sltui, sgt, sgtu, sgti, sgtui, sle, sleu, slei, sleui, jr, jalr and lhi bringing the total count to 52.
- **Optimized ALU**: the arithmetic logic unit (ALU) has been optimized and enhanced with a Pentium 4 Adder and a Booth Multiplier Radix-4.
- **Forwarding Unit**: a Forwarding Unit has been implemented to manage data hazards, specifically those related to Read-After-Write (RAW).
- **Parametric Design**: the entire project is designed to be parametric. All dimensions and design parameters can be managed through a single file, `000-globals.vhd`.
- **Semi-Automated Workflow**: scripts were used to partially automate various stages of the project, including simulation, synthesis and place-and-route. These scripts are detailed in Appendix A.

In addition to the advancements outlined above, various minor optimizations have been incorporated into the code for improved performance. Special attention has also been given to code readability. Great care has been taken to provide comprehensive comments throughout the code files and the scripts, making it as understandable and maintainable as possible.

CHAPTER 1

Introduction

1.1 Architecture

The DLX processor serves as a **32-bit RISC architecture** that closely resembles the MIPS model. It operates on a **5-stage in-order pipeline** for integer processing, comprising stages for fetch, decode, execute, memory, and write back. Memory interactions occur via specific **load** and **store** commands, and they function in **big-endian mode**.

Initially, in the **fetch stage**, an instruction is retrieved from the *Instruction Memory*. After a single clock cycle, the instruction moves to the **decode stage** and here the addresses of the involved registers are identified and sent to the *Register File*, which subsequently reads and outputs the corresponding values. Simultaneously, if an immediate value is present, it is extracted and sign-extended for use in later stages. During the **execution stage**, the *Arithmetic Logic Unit (ALU)* receives the appropriate input values, which may either come directly from the previous stage or be forwarded from one of the two subsequent stages, courtesy of the *Forwarding Unit*. Concurrently, a specialized module evaluates whether the first of the two registers value is zero, particularly relevant for branch operations. In the **memory stage**, the *DRAM* is accessed as needed, and jump conditions are assessed. Finally, during the **write back stage**, output values are channeled through designated *multiplexers*.

The subsequent chapters will provide a detailed explanation of each of these stages, as well as the methodologies employed to mitigate hazards and ensure reliable operation of the processor.

1.2 Instruction Set

Instructions in this architecture have a uniform 32-bit length and they can be classified into three principal categories:

- `I-Type` operations;
- `R-Type` operations;
- `J-Type` operations.

Irrespective of the instruction class, a 6-bit opcode field is invariably included. The opcode's nature dictates how the remaining 26 bits are interpreted, as illustrated in Figure 1.1.

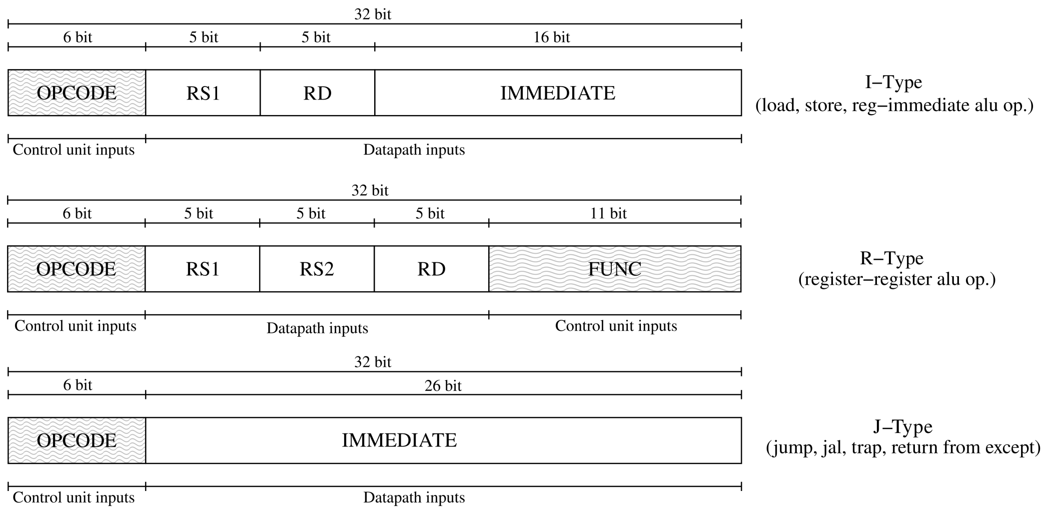

Figure 1.1: Instruction Formats

The processor is designed to implement a specific subset of the DLX ISA, comprising the following categories. Instructions shown as underlined are exclusive to the pro version of the processor.

1.2.1 Arithmetic Operations

These instructions perform basic arithmetic operations such as addition, subtraction and multiplication. The computed result is stored in a designated destination register. The operands for these operations can either be two source registers or a source register and an immediate numerical value as the second operand.

- `ADD`: an R-Type operation for adding the values in source registers.
- `ADDU`: an R-Type operation for the addition of unsigned integers from source registers.
- `ADDI`: an I-Type operation that adds an immediate value to a source register.
- `ADDUI`: an I-Type operation for adding an unsigned immediate value to a source register.
- `SUB`: an R-Type operation for subtracting the values of source registers.
- `SUBU`: an R-Type operation for the subtraction of unsigned integers from source registers.
- `SUBI`: an I-Type operation for subtracting an immediate value from a source register.
- `SUBUI`: an I-Type operation for subtracting an unsigned immediate value from a source register.
- `MULT`: an R-Type operation for multiplying the values of source registers.
- `MULTU`: an R-Type operation for multiplying unsigned values of a source register.

1.2.2 Shift Operations

These commands allow the shifting of binary digits within a register, either to the left or to the right. The right shifts can be either logical or arithmetic in nature.

- `SLL`: an R-Type operation that performs a logical left shift on the source register by the number of positions specified in another source register.
- `SLLI`: an I-Type operation that performs a logical left shift on the source register by the number of positions specified by the immediate value.
- `SRL`: an R-Type operation that executes a logical right shift on the source register by a count specified in another source register.
- `SRLI`: an I-Type operation that carries out a logical right shift on the source register by the number of positions dictated by the immediate value.
- `SRA`: an R-Type operation that conducts an arithmetic right shift on the source register by the number of positions indicated in another source register.
- `SRAI`: an I-Type operation that performs an arithmetic right shift on the source register by the number of positions given by the immediate value.

1.2.3 Logical Operations

These instructions help in bitwise manipulation and logical comparisons between source registers or between a source register and an immediate value.

- `AND`: an R-Type operation that performs a bitwise AND between the source registers.
- `ANDI`: an I-Type operation that executes a bitwise AND between a source register and an immediate value.
- `XOR`: an R-Type operation that performs a bitwise XOR between the source registers.
- `XORI`: an I-Type operation that carries out a bitwise XOR between a source register and an immediate value.
- `OR`: an R-Type operation that executes a bitwise OR between the source registers.
- `ORI`: an I-Type operation that performs a bitwise OR between a source register and an immediate value.

1.2.4 Comparison Operations

These instructions compare the values in source registers or a source register and an immediate value and write 1 to the destination register if the condition is satisfied.

- `SEQ`: an R-Type instruction comparing two source registers to determine if they are equal.
- `SEQI`: an I-Type instruction comparing a source register with an immediate value to determine if they are equal.
- `SNE`: an R-Type instruction comparing two source registers to determine if they are not equal.
- `SNEI`: an I-Type instruction comparing a source register with an immediate value to determine if they are not equal.
- `SGE`: an R-Type instruction comparing two source registers to ascertain if the first is greater than or equal to the second.

- `SGEU`: an R-Type instruction for unsigned comparison between two source registers to determine if the first is greater than or equal to the second.
- `SGEI`: an I-Type instruction comparing a source register and an immediate value to check if the first is greater than or equal to the second.
- `SGEUI`: an I-Type instruction for unsigned comparison between a source register and an immediate value, checking if the first is greater than or equal to the second.
- `SLT`: an R-Type instruction comparing two source registers to ascertain if the first is less than the second.
- `SLTU`: an R-Type instruction for unsigned comparison between two source registers to determine if the first is less than the second.
- `SLTI`: an I-Type instruction comparing a source register and an immediate value to check if the first is less than the second.
- `SLTUI`: an I-Type instruction for unsigned comparison between a source register and an immediate value, checking if the first is less than the second.
- `SGT`: an R-Type instruction comparing two source registers to ascertain if the first is greater than the second.
- `SGTU`: an R-Type instruction for unsigned comparison between two source registers to determine if the first is greater than the second.
- `SGTI`: an I-Type instruction comparing a source register and an immediate value to check if the first is greater than the second.
- `SGTUI`: an I-Type instruction for unsigned comparison between a source register and an immediate value, checking if the first is greater than the second.
- `SLE`: an R-Type instruction comparing two source registers to ascertain if the first is less than or equal the second.
- `SLEU`: an R-Type instruction for unsigned comparison between two source registers to determine if the first is less than or equal the second.
- `SLEI`: an I-Type instruction comparing a source register and an immediate value to check if the first is less than or equal the second.
- `SLEUI`: an I-Type instruction for unsigned comparison between a source register and an immediate value, checking if the first is less than or equal the second.

1.2.5 Jumps and Branches

These instructions handle conditional and unconditional branching. They redirect the program's execution flow to a specific instruction address, either by offsetting the current Program Counter (PC) or by setting it to an absolute address. Some of these instructions also save the return address in a designated register, specifically `R31`.

- `J`: a J-Type instruction that performs a relative jump to a target instruction, specified by adding an immediate value to the PC (Program Counter).

- `JAL`: a J-Type instruction that conducts a relative jump as specified by adding the PC and an immediate value. Additionally, stores the pre-jump PC incremented by 4 into the destination register.
- `JR`: a I-Type instruction that executes an absolute jump to the instruction address specified by the immediate value.
- `JALR`: a I-Type instruction that carries out an absolute jump to an instruction address specified by the immediate value. Before jumping, it also saves the current PC value incremented by 4 into the destination register.
- `BEQZ`: An I-Type instruction employed for conditional branching when the value in the specified register is equal to zero.
- `BNEZ`: An I-Type instruction employed for conditional branching when the value in the specified register is not equal to zero.

1.2.6 Load and Store Instructions

These instructions manage the transfer of data between the register file and the memory. The load instruction reads data from a specific memory address within the DRAM and stores it into a designated register in the Register File. Meanwhile, the store instruction writes data from the Register File to a specified memory location within the DRAM. In both cases, the target memory address is often calculated using an immediate value.

- `LW`: an I-Type instruction for reading a 32-byte signed value from the data memory.
- `SW`: an I-Type instruction to store a 32-bit value from an integer source register into the data memory.

1.2.7 General-purpose Operations

These instructions perform a general-purpose operations that may not fit cleanly into other categories.

- `NOP`: an I-Type instruction that performs no operation.
- `LHI`: an I-Type instruction that loads a 16-bit immediate value into the most significant half of the integer register.

Notes Programmers must be vigilant as the processor supports a branch delay slot. This means inserting three NOP instruction after each branching and jumping command to preclude unintended outcomes.

1.3 Software

This section offers a concise overview of the software tools deployed for the simulation, synthesis and physical implementation of the DLX architecture within the context of this project.

1.3.1 Questa Sim

Used for simulation, Questa Sim-64 version 2020.4 for Linux x86_64 is the core simulation and debug engine of the Questa verification solution by Siemens. Questa Sim also natively supports multiple key technologies such as SystemVerilog for Testbench, UPF, UCIS, and OVM/UVM, setting it apart from ModelSim in terms of performance, capacity, and features.

1.3.2 Design Compiler

For synthesis, we utilized Design Compiler Graphical, version S-2021.06-SP4 for Linux64 systems and developed by Synopsys. This tool can be accessed via a UNIX shell using either the `dc_shell`/`dc_shell-xg` commands or the `dc_shell-t`/`dc1_shell-xg-t` commands. The former set of commands utilizes a Synopsys-specific proprietary language, while the latter employs the more standardized Tcl language. The focus of this project is solely on the Tcl-based version of the Design Compiler.

1.3.3 Innovus Implementation System

The physical implementation for this project was carried out using the Cadence Innovus Implementation System, version v20.11-s130_1. Innovus is specifically designed to cater to the requirements of contemporary system-on-chip (SoC) developers. It offers cutting-edge advancements in power, performance, and area (PPA) metrics, while simultaneously accelerating time-to-market. Furthermore, Innovus incorporates machine learning-driven capabilities and is part of the Cadence Safety Solution.

CHAPTER 2

Datapath

2.1 Structure

As previously articulated, and as represented in detail with a representative schematic in Figure 2.1, the datapath segregates into five distinct phases. The initial phase, known as the **fetch stage**, is responsible for retrieving instructions from the Instruction RAM (IRAM). Subsequently, the **decode stage** is tasked with obtaining register values correlated with the addresses specified in the fetch stage and extending immediate values to 32 bits. Following this, the **execute stage** takes charge of performing all the arithmetic, logical, and shift operations. Subsequently the **memory stage**, acting as the fourth phase, manages read or write actions to the Data RAM (DRAM). Lastly, the **write back stage** is in charge of correctly routing the results generated either by the Arithmetic Logic Unit (ALU) or the DRAM itself.

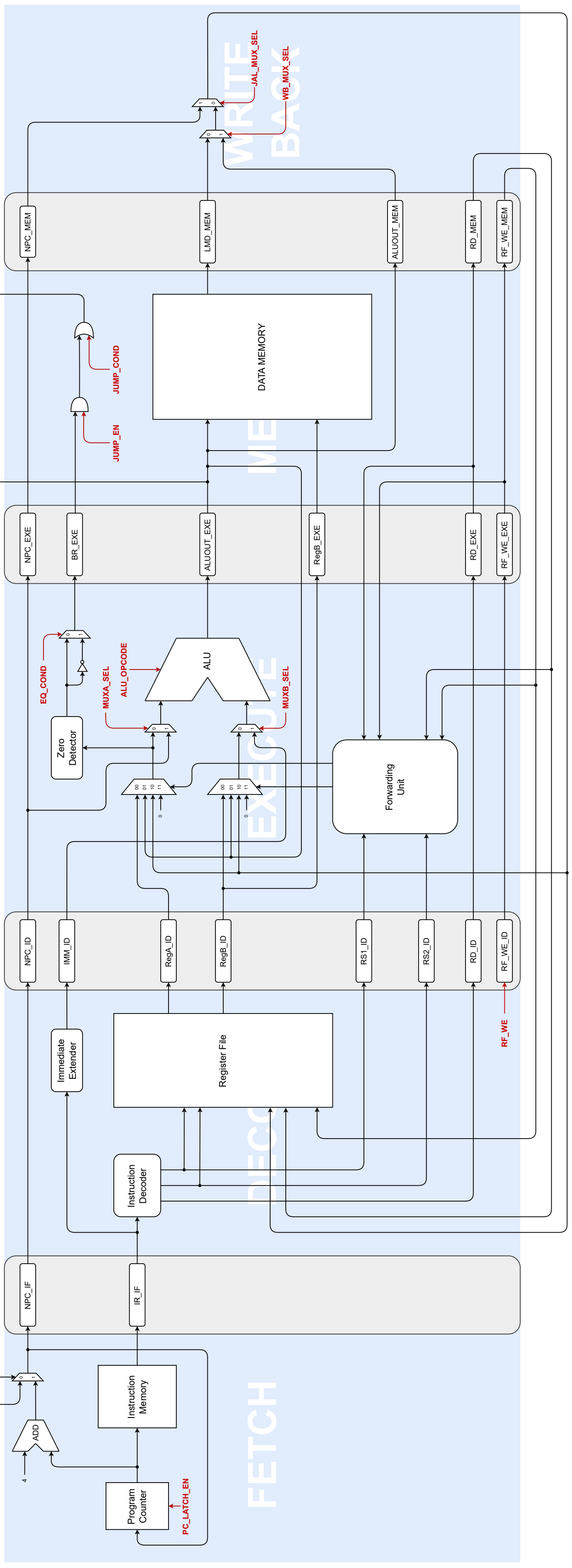

Figure 2.1: Datapath architecture

2.2 Fetch Stage

During the fetch stage, the next instruction to be executed is retrieved from the **Instruction Memory (IRAM)**, using the current address stored in the **Program Counter (PC)**. The instruction is then stored in the **Instruction Register (IR)** before proceeding to the next pipeline stage. In scenarios of sequential instruction execution, the PC is typically incremented by 4 to align with the 4-byte encoding standard of each instruction. In contrast, for control flow instructions like branches or jumps, the PC may be modified differently. Specifically, the **multiplexer** is activated in these cases, directing the output of the **Arithmetic Logic Unit** from the Memory stage to both the **Program Counter** and the **Next Program Counter Register**, as depicted in Figure 2.2.

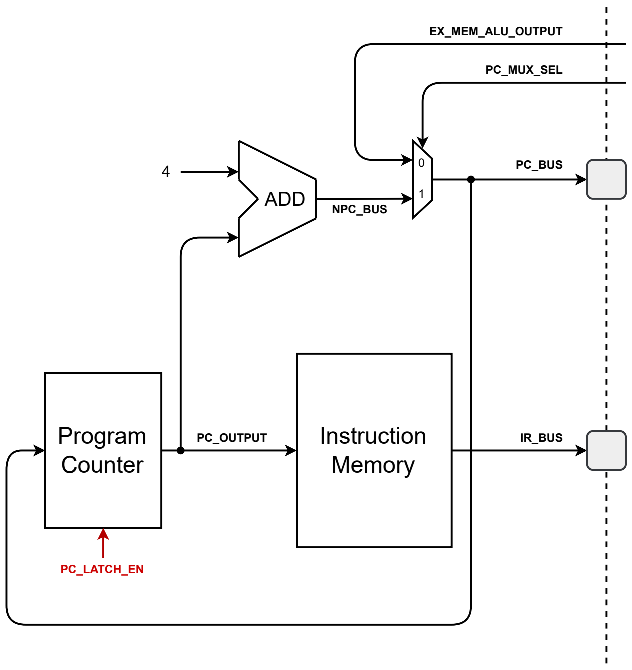

Figure 2.2: Fetch stage architecture

2.2.1 Program Counter Register (PC)

The Program Counter Register (PC) is a 32-bit register that holds the memory address of the instruction that is to be fetched and executed next. This register is updated continuously during the pipeline's operation to ensure smooth sequential instruction execution.

2.2.2 Instruction Register (IR)

The Instruction Register (IR) is another 32-bit register, which temporarily stores the instruction fetched from the Instruction Memory (IRAM). Once the instruction is loaded into the IR, it is forwarded to the decoding stage, where its opcode and operands are analyzed. Given the transient storage requirement for each fetched instruction, a latch-based D Register is used for implementing this component.

2.2.3 Next Program Counter Register (NPC)

The Next Program Counter Register (NPC) is a specialized 32-bit register designed to facilitate the forwarding of the Program Counter (PC) throughout the pipeline stages. This register is especially crucial when handling jump and branch instructions, as it can hold an alternative address to which control must be transferred. Like the Instruction Register, the NPC is implemented using a latch-based D Register.

2.2.4 Adder

This simple Adder calculates the address for the succeeding instruction by simply adding 4 to the current PC value $PC + 4$. This is done to match the 4-byte instruction encoding standard, allowing for a seamless transition to the next instruction in the memory.

2.2.5 Multiplexer

This Multiplexer is used to choice between the next sequential address $PC + 4$ and an alternative address provided by the Arithmetic Logic Unit (ALU). This component comes into play particularly during conditional branching or jumps, enabling the system to either continue with the sequential flow or divert to a different flow based on conditions met in the memory stage.

Additional Notes It's worth noting that while the Instruction Memory (IRAM) plays a critical role in the fetch stage by providing the instructions to be executed, it is not considered part of the datapath itself. IRAM is responsible for storing the entire instruction set that the DLX processor will execute, and its integration is essential for the pipeline's functioning. However a detailed discussion on the Instruction Memory will be covered in Chapter 4, which is dedicated to memory systems.

2.3 Decode Stage

During the decode stage, as shown in Figure 2.3, the pipeline prepares for the subsequent execution of the instruction initially fetched into the **Instruction Register (IR)**. The primary objective of this stage is to interpret the instruction and extract the operands and immediate values so that they can be utilized in the following stages of the pipeline.

At the heart of this operation is the **Instruction Decoder**, which breaks down the instruction to extract the register addresses that will act as operands for the upcoming operation. Working in concert with the decoder, the **Immediate Extender** performs sign-extension for immediate values. Complementing these components, the **Register File** stores the processor's registers, making these values accessible for the stages that follow.

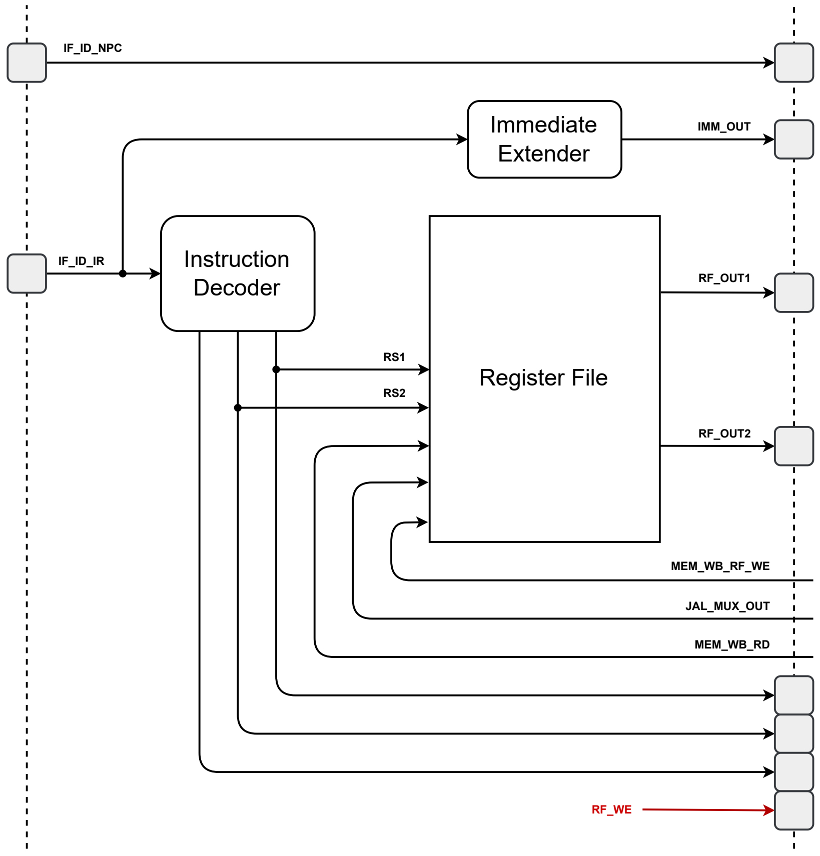

Figure 2.3: Decode stage architecture

2.3.1 Instruction Decoder

The Instruction Decoder takes the 32-bit instruction fetched into the Instruction Register (IR) and extracts the register addresses that will be used as operands for the upcoming operation, based on the last 6 bits that represent the opcode of the instruction.

It identifies the type of instruction, whether it's R-Type, I-Type, or J-Type, and retrieves the output addresses of the source registers (`RS1` and `RS2`) and the destination register (`RD`) that will be engaged in the actual operation. For instance, in the case of an R-Type instruction, all three fields are relevant. Conversely, for a standard `J` instruction, these fields may not be applicable, and they are set to zero, as previously shown in Figure 1.1.

The module also accommodates specificities in certain J-Type instructions: for example, when encountering a Jump and Link instruction (either `JAL` or `JALR`), the `RD` is automatically set to 31.

2.3.2 Immediate Extender

The Immediate Extender kicks in to perform sign-extension for immediate values. This ensures that immediate values, initially in either 16-bit or 26-bit formats, are accurately converted to a 32-bit format, enabling them to be treated as either signed or unsigned as the instruction dictates.

The Immediate Extender module is responsible for performing sign-extension of immediate values. This functionality ensures that immediate values, which may initially be in 16-bit or 26-bit formats, are accurately converted to a 32-bit format. The conversion allows these values to be treated as either signed or unsigned integers, in accordance with the specific instruction being executed.

This module determines the type of instruction, such as R-Type, I-Type, or J-Type, based on the opcode. For R-Type instructions, the immediate field is not applicable, and the module sets it to zero. On the other hand, J-Type instructions may contain a 26-bit immediate value, which undergoes sign-extension to align with the 32-bit architecture. I-Type instructions may involve either zero-extension or sign-extension of a 16-bit immediate value, depending on the particular operation being carried out.

For certain special cases like Load High Immediate (`LHI`), the higher 16 bits of the immediate value are used, while the lower 16 bits are zero-extended.

2.3.3 Register File

The Register File contains 32 general-purpose registers designed to hold runtime values for computation. These registers are 32-bit integers and can be addressed using 5-bit addresses, effectively ranging from `R0` to `R31`.

Certain registers receive special attention, as they are designated for specific roles in the processor's operation. For example, the `R0` register is read-only and is hardwired to always contain the value zero. On the other hand, the `R31` register is commonly used as the storage location for return addresses, particularly when executing Jump and Link instructions (`JAL` or `JALR`). Despite its specialized function, `R31` can also be employed for other general-purpose operations.

Reading from the Register File is always permitted, thereby obviating the need for a separate read-enable signal and simplifying the control logic. In contrast, a specific write-enable signal is generated by the Control Unit. This signal is carefully timed to ensure that data is written back into the register file only during the write-back stage. This synchronization is crucial for features controlled by the Forwarding Unit, which will be elaborated upon in a subsequent section.

The Register File is equipped with dual read data ports (**RD1** and **RD2**), enabling simultaneous reading from two different registers. This feature is essential for instructions that require two operands, as both can be fetched in a single clock cycle. Additionally, there is a single write data port that accepts the data to be written back into the register file. This data typically originates from the output of the final multiplexer in the write-back stage.

An active-low reset signal is also integrated into the design. When triggered, this signal clears the contents of all the registers. This functionality is particularly useful for initialization purposes or in situations requiring a rapid state reset.

2.4 Execution Stage

As illustrated in Figure 2.4, the execution stage is responsible for accurately routing requisite operands to the **Arithmetic Logic Unit (ALU)** for a variety of computational tasks, ranging from basic arithmetic to bitwise logical operations. To ensure the correct operands are supplied to the ALU, a tree structure of multiplexers in conjunction with the **Forwarding Unit** manages the delivery of operands from the two subsequent pipeline stages. This is particularly important for cases where the results have not yet been written back into the Register File. Additionally, a specialized module called **Zero Detector** is tasked with verifying whether the first operand is zero, thus preparing the pipeline for potential conditional branches.

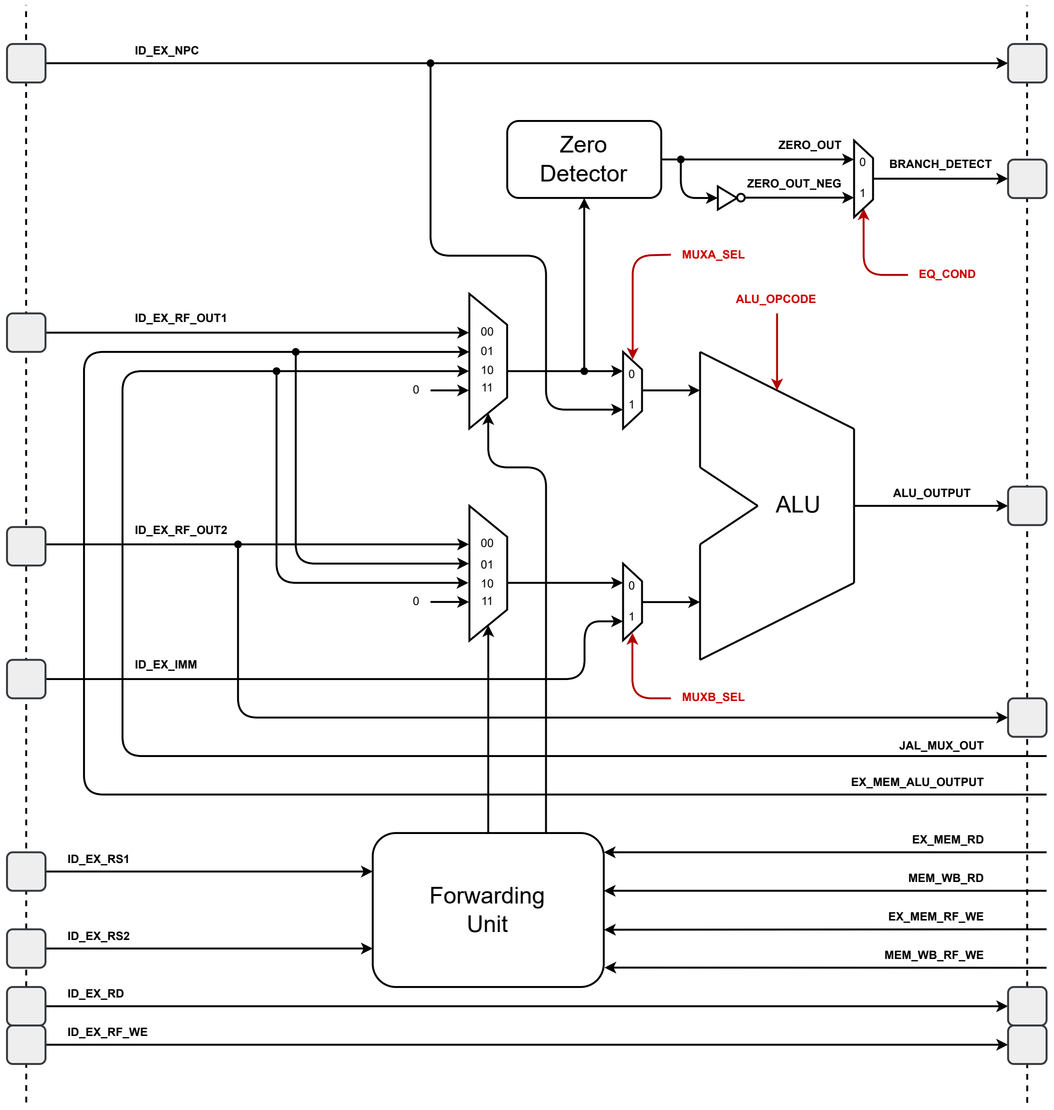

Figure 2.4: Execute stage architecture

2.4.1 Arithmetic Logic Unit (ALU)

The ALU, shown in Figure 2.5, serves not only as the computational core of the execution stage but also as the pivotal computational component of the entire architecture. It is designed to handle a variety of operations, including both signed and unsigned arithmetic. The ALU receives two 32-bit operands along with a signal that specifies the operation type, and produces a single 32-bit output accordingly.

The ALU's versatility is attributed to several internal functional components that have been incorporated into the project. These include:

- an adder module based on the **Pentium 4 Adder**, responsible for adding or subtracting operands;
- a specialized **Comparator Unit** for comparing both signed and unsigned numbers;
- a specialized **Logic Unit** for performing bitwise AND, OR and XOR operations between operands;
- two distinct **Barrel Shifters** for executing arithmetic and logic shifts;
- a **Radix-4 Booth Multiplier** for multiplication operations.

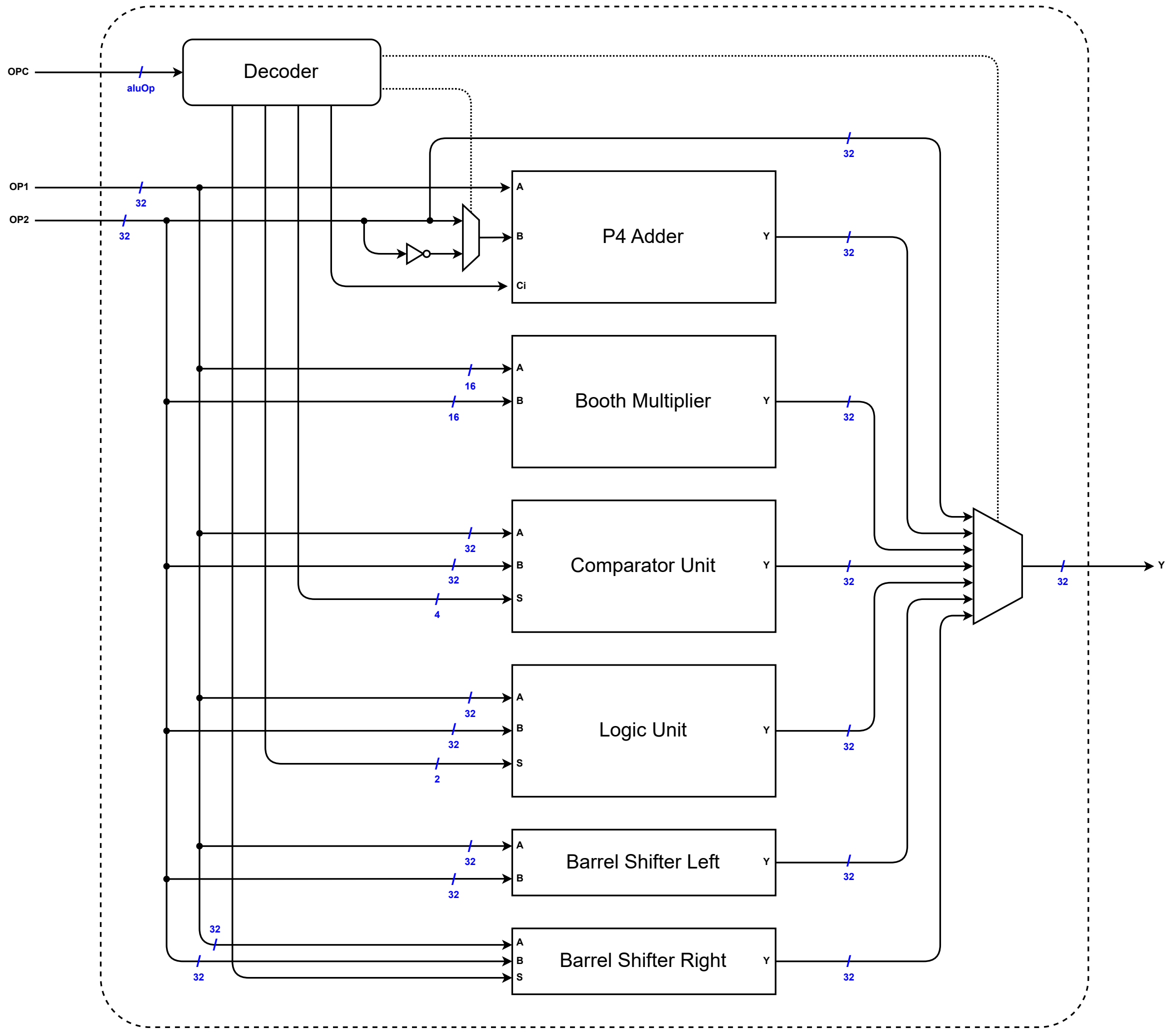

Figure 2.5: Arithmetic Logic Unit (ALU) architecture

Each of these components can handle both R-type operations, where values are sourced from the Register File, and I-type operations, where values are retrieved from the Register File as well as the Immediate field of the instruction being executed. Furthermore, the ALU also supports the computations required for generating the destination addresses for J-type instructions.

Pentium 4 Adder

The Pentium 4 Adder is a specialized 32-bit arithmetic unit capable of performing both addition and subtraction operations seamlessly. This adder builds upon the foundational netlists developed during laboratory sessions and is fundamentally divided into two primary subcomponents: the **Sum Generator** and the **Carry Generator**.

- **Carry Generator**: computes the intermediate bits and the final carry bit.
- **Sum Generator**: produces the final sum of the operands by combining the input operands and the intermediate carries.

It is worth noting that the design of the P4 Adder emphasizes parallelism, utilizing both the Sum Generator and the Carry Generator in tandem. This approach not only increases speed by reducing the ripple effect commonly observed in simpler adders but also boosts operational efficiency.

Comparator Unit

The Comparator Unit provides support for executing a variety of comparison operations. These operations include *less than* ($<$), *greater than* ($>$), *less than or equal to* ($\leq$), *greater than or equal to* ($\geq$), *equal to* ($=$), and *not equal to* ($\neq$). Furthermore, the module is capable of handling both signed and unsigned operand comparisons.

The core functionality is governed by a 4-bit selector signal, which determines the type of comparison operation to be carried out. Depending on the value of this selector, the module performs the appropriate comparison between the two 32-bit operands and generates a 32-bit output to represent the result. Notably, the output is not a single-bit flag; instead, it is a full 32-bit word. This design choice maintains consistency with the data bus width and allows for the possibility of chaining with other operations.

Logic Unit

Designed for the execution of bitwise logical calculations, specifically `AND`, `OR`, and `XOR`, the Logic Unit employs a 2-bit selector signal to choose the operation to be carried out.

If the selector signal reads "00" the unit conducts a bitwise `AND` operation on the operands. If the selector signal shows "01" the unit performs a bitwise `OR` operation and for "10" it carries out a bitwise `XOR` operation. Should the unit receive an invalid selector signal, it is configured to return a default 32-bit zero output.

Barrel Shifters

Within the ALU, two types of barrel shifters are incorporated to execute both left and right shifts. In the case of the latter, both logical and arithmetic shifts are supported.

- **Barrel Shifter Left (BSL)**: this module performs the left-shift operation on the first operand by a number specified by the second operand. The leftmost bits are discarded, and '0's are filled into the rightmost positions.

- **Barrel Shifter Right (BSR)**: this module performs the right-shift operation on the first operand by a number specified by the second operand. When executing a logical shift, '0's are introduced on the left. In contrast, during an arithmetic shift, the MSB (sign bit) is retained to ensure that the sign of the number remains unchanged. A shift type identifier is used to determine whether the shift is logical or arithmetic.

Both modules are structurally defined, and their architectural design employs a sequence of multiplexers. This design approach eliminates the need for iterative shifting one bit at a time, offering a significant speed advantage, especially for larger shifts.

Radix-4 Booth Multiplier

This project introduces a 32-bit signed integer multiplier based on the Booth's Multiplication Algorithm. The multiplier takes in two 16-bit operands and during each step of the partial multiplication process, the first operand is shifted one position, while the second operand is grouped into 3-bit segments. The summation of these partial products is executed using a tree-structured arrangement of Ripple-Carry Adders (RCA).

Booth's Multiplication Algorithm Booth's multiplication technique aims to optimize the multiplication process by reducing the number of partial products required. A sequence of consecutive '1's in a binary number can be replaced by generating a carry-out coupled with a subtraction at the least significant bit of the string. Mathematically, this is expressed as:

$$\sum_{n=0}^{N-1} 2^n = 2^N - 1$$

For example, 0111_2 can be re-expressed as $1000_2 - 0001_2$. Similar methods can be applied to sequences of '1's appearing at other positions in a number by appropriately left-shifting the result.

This optimization enables the reduction of the number of partial products involved in the multiplication. In this approach, the multiplier is represented in radix-4, allowing for digits between 0 and 3, as opposed to just 0 and 1. This halves the number of partial products needed since each multiplication operation now processes two bits at a time. However, multiplying by 3 presents a challenge, unlike multiplying by 0, 1, or 2, which only require simple shifts. To sidestep the difficulty of multiplying by 3, Booth's observations are applied to recode the digit set to include 2, 1, 0, -1, and -2. Positive digits yield simple partial products, whereas negative digits are handled by subtraction instead of addition, as shown in Table 2.1.

$Bi+1$	Bi	$Bi-1$	$E2i$
0	0	0	0
0	0	1	A
0	1	0	A
0	1	1	$2A$
1	0	0	$-2A$
1	0	1	$-A$
1	1	0	$-A$
1	1	1	0

Table 2.1: Booth Encoding Algorithm Table

2.4.2 Forwarding Unit

The Forwarding Unit serves as an essential component in enhancing pipeline efficiency and mitigating data hazards, specifically **read-after-write (RAW)** dependencies. It operates by identifying data dependencies between forthcoming instructions and instructions still in the pipeline but whose results have been fully computed.

This unit employs a comparison between register source encodings into the execution stage with those present in the memory and write-back stages. A match signifies a pending operand, already computed but not yet stored, that is required by the instruction decoded in the previous clock cycle.

Upon recognizing such data dependencies, the Forwarding Unit eliminates the need for **pipeline stalls** by routing the already-computed data directly to the required ALU input. This is achieved via a secondary set of multiplexer interconnections, as represented in the Figure 2.4. In doing so, the Forwarding Unit ensures that each computation utilizes the correct, most up-to-date data, allowing for an optimized execution pipeline.

2.4.3 Zero Detector

In this architecture, the Zero Detector module plays a critical role in conditional **branching instructions**. It examines the first operand to determine whether all its bits are set to zero. This functionality significantly aids in evaluating branch conditions such as `BEQZ` and `BNEZ`. By quickly establishing the **zero condition**, the Zero Detector expedites the decision-making process for branching.

2.4.4 Multiplexers

Five multiplexers are employed within the execution stage:

- **Two 32-bit 4-to-1 multiplexers** are responsible for selecting the first input for subsequent multiplexers. These are controlled by the Forwarding Unit, which allows for the selection of content originating from either the Register File in the previous stage, the memory stage, or the write-back stage.
- **Two 32-bit 2-to-1 multiplexers** serve to select the actual input value for the ALU. The choices are between the outputs of the preceding 4-to-1 multiplexers or, depending on their respective connections to the ALU, either the Next Program Counter value or the 32-bit sign-extended immediate value from the previous stage. These multiplexers are both governed by signals generated by the Control Unit.
- **A single-bit 2-to-1 multiplexer** that, depending on the type of branch to be executed (`BEQZ` or `BNEZ`), outputs either the output of the Zero Detector or its negated version.

2.5 Memory Stage

The memory stage functions as the fourth stage of the pipeline and serves as a straightforward segment for managing data as shown in Figure 2.6. In this stage, data is either read from or write into the **data memory**, accessible solely through load and store operations. Additional **logic ports** are employed here to handle branch and jump conditions, as well as to control the multiplexer in the fetch stage.

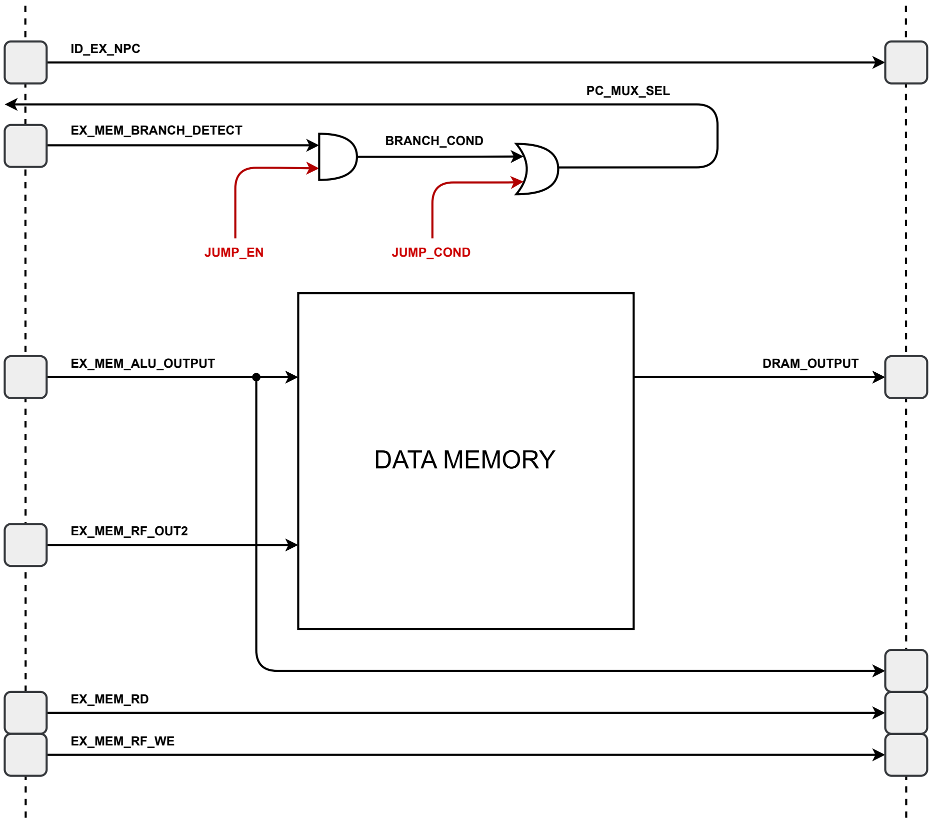

Figure 2.6: Memory stage architecture

2.5.1 Logic Ports

Two simple logic ports are utilized within the memory stage to manage jumps and branches effectively:

- An **AND logic gate** is used in conjunction with a signal generated by the Control Unit. This signal becomes active exclusively during the execution of `BEQZ` or `BNEZ` instructions, allowing the branch condition to be forwarded to the subsequent logic stage if satisfied.
- An **OR logic gate** is responsible for generating the signal that directs the multiplexer in the fetch stage to select the next address for the Program Counter. This gate outputs a one either when the branch condition from the preceding stage is met or when the Control Unit's signal, designated for unconditional jumps, is activated.

Additional Notes Similar to the role of the Instruction Memory (IRAM) in the fetch stage, the Data Memory (DRAM) is crucial in the pipeline but is not inherently a part of the datapath. The DRAM functions within the memory stage, where it manages read or write operations as instructed by the pipeline. Essentially, DRAM serves as the storage hub for data that the DLX processor will manipulate throughout its operation. Unlike the IRAM, which stores only instructions, the DRAM accommodates data that can be both read from and written into. This flexibility makes it instrumental for the load and store operations performed in the memory stage. A comprehensive discussion on the Data Memory will also be covered in Chapter 4, which focuses on memory systems in greater detail.

2.6 Write-Back Stage

The write-back stage serves as the phase in which the computational results are stored back into the designated register, as identified during the decode stage of the instruction pipeline. A two-level multiplexer tree is utilized here, as illustrated in Figure 2.7, to deliver the value required by the current instruction to the system. The write-enable signal, originally generated by the Control Unit, is routed to the Register File at this stage to facilitate the acceptance of the new value.

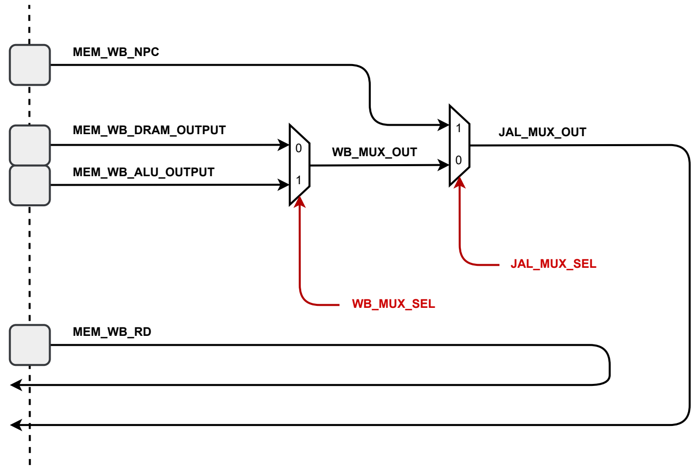

Figure 2.7: Write-Back stage architecture

2.6.1 Multiplexers

A straightforward hierarchy comprising two 32-bit 2-to-1 multiplexers is utilized in this stage:

- The **first-level multiplexer**, controlled by a specific signal generated by the Control Unit, selects between the output of the ALU and the output from the DRAM in the event that a load operation (`LW`) has been executed.
- The **second-level multiplexer**, similarly governed by a dedicated signal from the Control Unit designed for handling jump-and-link instructions like `JAL` or `JALR`, chooses between the output of the preceding multiplexer and the value of the Next Program Counter that has been forwarded to this stage.

CHAPTER 3

Control Unit

3.1 Hardwired Control Unit

The Control Unit (CU) is responsible for orchestrating the flow of control signals across the pipeline. It takes in the output from Instruction Memory (IRAM), which is determined by the current value of the Program Counter (PC), and contains the instruction to be executed.

The architecture of the CU is designed using a hard-wired approach to maximize both reliability and simplicity. Two **Look-Up Tables** are employed within the combinational logic to generate an **18-bit Control Word** signal and the **ALU operational code**. These control signals are then segmented and distributed to the various pipeline stages: fetch, decode, execute, memory and write-back.

As previously discussed in Section 1.2, the instruction set is categorized into three primary types: register-to-register (R-Type), immediate operations (I-Type) and jump instructions (J-Type). The encoding format for each instruction type varies as follows:

- **R-Type**: 6-bit OPCODE, 5-bit operand A (RS1), 5-bit operand B (RS2), 5-bit destination register (RD), and 11-bit FUNC.
- **I-Type**: 6-bit OPCODE, 5-bit operand A (RS1), 5-bit destination register (RD), and 16-bit immediate.
- **J-Type**: 6-bit OPCODE and 26-bit immediate.

Upon receiving an instruction, the CU extracts the **OPCODE** and, in the case of R-type instructions, the **FUNC** field. The OPCODE informs the CU about the specific type of instruction to be executed, while the FUNC field guides the ALU to perform the corresponding operation. Implicit encoding is chosen for the ALU opcode to improve readability during both the coding and testbench phases.

The pipeline accepts a new instruction during each clock cycle, enabling the simultaneous execution of up to five instructions, each at a different stage of the pipeline. To keep the control signals in sync with the pipeline stages, the Control Word and the ALU operational code are shifted by one position with every positive edge of the clock cycle. This ensures that control signals are delivered to the correct pipeline stage in a timely manner, as depicted in Figure 3.1.

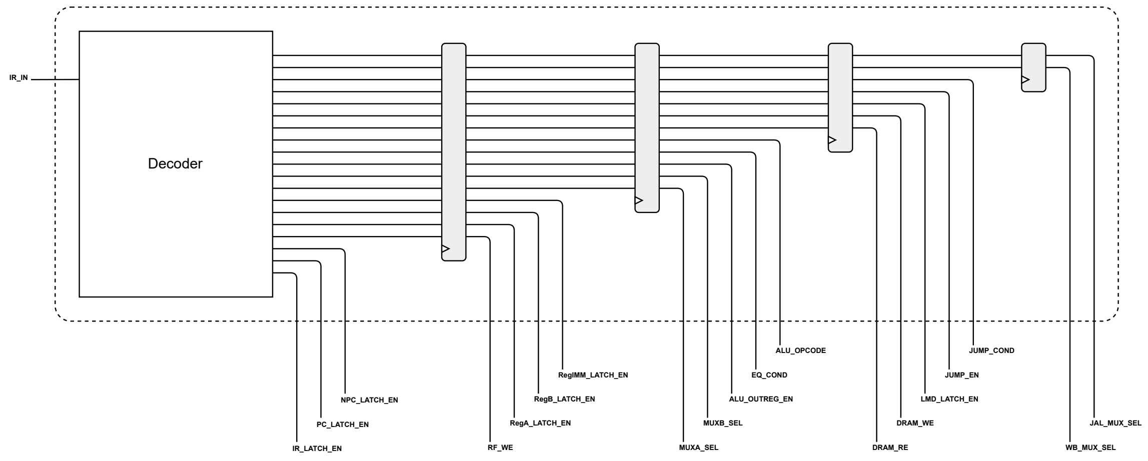

Figure 3.1: Control Unit

3.2 Control Word

This section describes the control signals produced by the Control Unit to manage the various stages of the DLX pipeline. With the exception of `ALU_OPCODE`, that specifies the operation to be performed by the ALU, all other signals are part of the Control Word and are thus 1-bit signals.

The signals are presented in reverse order compared to their definitions in the Control Unit module. This is because the Most Significant Bit (MSB) of the Control Word represents the first signal sent to the datapath, specifically `IR_LATCH_EN`. Conversely, position 0 of the Control Word corresponds to the last signal sent to the write-back stage, which is `JAL_MUX_SEL`.

- **WB Control Signals**

  0. `JAL_MUX_SEL`: selects between the output of the previous multiplexer and the Next Program Counter in case of Jump and Link.
  1. `WB_MUX_SEL`: selects between ALU and DRAM output.

- **MEM Control Signals**

  2. `JUMP_COND`: enables the unconditional jump condition.
  3. `JUMP_EN`: enables the jump condition due to branch instructions.
  4. `LMD_LATCH_EN`: enables the latching of the Load Memory Data register.
  5. `DRAM_WE`: enables write operations to the Data RAM.
  6. `DRAM_RE`: enables read operations from the Data RAM.

- **EX Control Signals**

  7. `EQ_COND`: enables a branch if the condition is (or is not) zero.
  8. `ALU_OUTREG_EN`: enables the output register of the ALU.
  9. `MUXB_SEL`: drives the second input of the ALU selects the output of the second 2-to-1 multiplexer.
  10. `MUXA_SEL`: drives the first input of the ALU selects the output of the first 2-to-1 multiplexer.

- **ID Control Signals**

11. `RegIMM_LATCH_EN`: enables the latching of the immediate value.
12. `RegB_LATCH_EN`: enables the latching of the content of Register B.
13. `RegA_LATCH_EN`: enables the latching of the content of Register A.
14. `RF_WE`: enables write operations to the Register File.

- **IF Control Signals**

15. `NPC_LATCH_EN`: Enables the latching of the Next Program Counter value.
16. `PC_LATCH_EN`: Enables the latching of the Program Counter value.
17. `IR_LATCH_EN`: Enables the latching of the current instruction into the Instruction Register.

3.3 Look-up Table

In Table 3.1, the Control Words corresponding to the implemented instructions stored inside the respective Look-up Table are displayed.

General instructions		**Control Words - Bit Position**																	
OPCODE	Mnemonic	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	`RTYPE`	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0
0x02	`J`	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	1	1	0
0x03	`JAL`	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	1	1	1
0x04	`BEQZ`	1	1	1	0	1	0	1	1	1	1	0	0	0	0	1	0	1	0
0x05	`BNEZ`	1	1	1	0	1	0	1	1	1	1	1	0	0	0	1	0	1	0
0x08	`ADDI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x09	`ADDUI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x0A	`SUBI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x0B	`SUBUI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x0C	`ANDI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x0D	`ORI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x0E	`XORI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x0F	`LHI`	1	1	1	1	0	0	1	0	1	1	0	0	0	0	0	0	1	0
0x12	`JR`	1	1	1	0	1	0	0	0	1	1	0	0	0	0	0	1	1	0
0x13	`JALR`	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1
0x14	`SLLI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x15	`NOP`	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0x16	`SRLI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x17	`SRAI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x18	`SEQI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x19	`SNEI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x1A	`SLTI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x1B	`SGTI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x1C	`SLEI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x1D	`SGEI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x23	`LW`	1	1	1	1	1	1	1	0	1	1	0	1	0	1	0	0	0	0
0x2B	`SW`	1	1	1	0	1	1	1	0	1	1	0	0	1	0	0	0	1	0
0x3A	`SLTUI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x3B	`SGTUI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x3C	`SLEUI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0
0x3D	`SGEUI`	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	1	0

Table 3.1: Control Word Look-up Table

Notes This version of the Control Unit does not provide any type of stall or support for branch prediction. This limitation necessitates the insertion of three `NOP` instructions into the assembly code whenever a jump or branch operation is executed to ensure correct functioning. Alternatively, the compiler could be modified to automatically include these three `NOP` instructions. This aspect, as well as possible alternatives for further implementation, is discussed in Chapter 8.

CHAPTER 4

Memories

Though not directly incorporated into the datapath, both the instruction and data memories play critical roles in the fetch and memory stages, respectively. It is worth noting that these memory components have neither been synthesized nor had their physical designs realized.

4.1 Instruction Memory (IRAM)

The Instruction Memory is straightforward and serves as RAM storage, with each row accommodating 32-bits, which contain a single instruction.

Two principal procedures exist within the Instruction Memory:

- **Reading**: the initiation of the reading operation is triggered when the Reset signal `RST` is asserted to be 0. This process sequentially reads the contents from the file named `firmware.asm.mem`, storing them in the internal memory. Each row within the file contains a single 32-bit instruction.
- **Writing**: this operation simply updates a 32-bit data value whenever there is a modification in the address.

4.2 Data Memory (DRAM)

The Data Memory component serves as a specialized memory module that is responsible for storing data that can be accessed and modified through specific load and store instructions, such as `LW` and `SW`.

For the load operation, data is fetched from the DRAM and transferred to one of the 32 general-purpose registers in the Register File. Conversely, the store operation writes data from a specific general-purpose register back into the Data Memory.

The Data Memory operation can be summarized through two principal procedures:

- **Reading**: when the read enable signal is asserted, data is fetched from the memory location pointed to by the current address input. This fetched data is then sent to the output port to be further stored in the Register File. Reading is asynchronous and the output is updated as soon as the read enable signal is high.

- **Writing**: writing occurs synchronously with the rising edge of the clock signal. When the write enable signal is high, the data at the input port is written into the memory location specified by the current address input.

The Data Memory employs a singular address input port, which is used for both reading and writing operations. However, reading and writing operations cannot occur simultaneously.

When the reset is activated, all memory locations within the DRAM are cleared, effectively setting them to zero, to ensure a clean state for subsequent operations.

CHAPTER 5

Simulation

5.1 Assembler

Thanks to the initial files provided for this project, it has been possible to automate the loading of instruction code from an external assembly file. This automation is facilitated by the `assembler.sh` file, which serves as an interface for the actual Perl language assembler, `dlxasm.pl`.

The `dlxasm.pl` file was modified solely to enable the execution of the `MULT` and `MULTU` instructions by changing the OPCODE to R-type. Meanwhile, the `assembler.sh` file was slightly modified to better manage the generation of test files and maintain as clean a working directory as possible (see Appendix A.1.1).

Furthermore, nineteen assembly files have been either created or modified, building upon the files initially provided.

5.2 Scripts

The `runsim.sh` script, fully detailed in Appendix A.1.2, is designed to manage the simulation process for the DLX project using the QuestaSim software, as mentioned in Section 1.3. This script allows users to specify the test file to be used for the simulation and to choose whether to run the simulation in the background. Two specific flags can be employed for these purposes:

- `-f`: specifies the test filename. By default, the filename is `test.asm`.
- `-b`: allows the user to choose whether to run the simulation in the background (‘y’ for yes, ‘n’ for no). The default setting is to run in the background.

To execute the script, navigate to the `dlx/sim` directory, which contains `runsim.sh`, and run the following command:

```
> ./runsim.sh
```

As mentioned earlier, running the simulation with default settings will use the default file (`test.asm`) and execute in the background. To run a custom test file (`mytest.asm`) in the background, use the following command:

```
> ./runsim.sh -f mytest.asm
```

To run a custom test file (`mytest.asm`) in the foreground, use the following command:

```
> ./runsim.sh -f mytest.asm -b n
```

To run the script again within the QuestaSim program, you must instead launch the `resim.sh` script, detailed in Appendix A.1.3, using the following command:

```
> do resim.tcl
```

Both of these files set up the optimal environment for running the simulation and then execute the `sim.tcl` file, which serves as the core of the simulation. This file, detailed in Appendix A.1.4, contains the commands for compiling the VHDL files and managing the waveforms of the entire system's signals. It offers a rudimentary control panel at the beginning of the file, which allows the user to select the unit to execute (choices include DLX, Control Unit, Datapath, IRAM, DRAM, or ALU) and/or the stage to analyze (among all, fetch, decode, execute, memory, and write back).

Thanks to this type of display selection, a single testbench file could be used, enabling quicker identification of any bugs and delays. Furthermore, the more complex modules, such as the P4 Adder, Booth Multiplier and Control Unit, had already been tested with specialized testbenches during the course, although they were slightly modified for this project.

CHAPTER 6

Synthesis

6.1 Scripts

The synthesis procedure for the DLX architecture was carried out using the Synopsys Design Compiler, as mentioned in Section 1.3. The UMC 45nm technology library was employed for this purpose.

For the simulation phase, the bash script `runsyn.sh` (see Appendix A.2.1) was developed to initiate and streamline the synthesis process. This script also accommodates the same two optional flags see in Section 5.2 for simulation:

- **-b**: allows execution in the background when paired with the `y` label, alternatively, using the `n` label initiates foreground execution, which is also the default behavior in this case.
- **-f**: enables the synthesis of a specific target file. In the absence of a specified file, the default action is to execute all types of synthesis files within the current subdirectory.

To execute the script, navigate to the `dlx/syn` directory that contains `runsyn.sh`. Then, run the following command:

```
> ./runsyn.sh
```

6.2 Workflow

The subsequent section provides a structured walkthrough of the command sequence employed within the Design Compiler for conducting the synthesis of the architecture:

1. The VHDL source files for the DLX architecture were imported using the `analyze` command.
2. To facilitate subsequent power analysis, RTL naming was retained in the generated netlist, using the command `set power_preserve_rtl_hier_names true`.
3. Design Compiler was instructed to use generic gates to elaborate the specified entities. The top entity, representing the DLX core, was synthesized with the command: `elaborate DLX -architecture DLX_RTL -library WORK`.
4. For the purpose of warning reduction, unconnected ports were purged from the netlist via the command `remove_unconnected_ports [get_cells -hierarchical *]`.

5. Wire load models were set using the command `set_wire_load_model -name 5K_hvratio_1_4` to precisely calculate wire capacitance. This ensures accurate delay estimations and therefore contributes to a more reliable synthesis and timing analysis.

In the workflow outlined earlier, the first two steps have been automated and generalized using the `analyze.tcl` script, detailed in Appendix A.2.2. This script is applicable to all three synthesis files and configures the Synopsys Design Compiler to read the `file_list.lst` file (see Appendix A.2.3), which contains the paths to the VHDL files. This setup enables centralized editing and should modules be added or removed from the project, only this single file would require modification. Furthermore, the script is designed to automatically exclude empty lines and any paths in `file_list.lst` that are commented out.

As previously stated, the subsequent tasks of compilation and report generation are managed by two different scripts defined in a TCL file.

- **Unconstrained** - The architecture was synthesized without any specific constraints imposed (Appendix A.2.4).
- **Clock Bound** - Synthesis was carried out with clock frequency optimizations (Appendix A.2.5).

For the **Unconstrained** variant, the synthesis process was limited to the execution of the `compile` command, along with the subsequent generation of reports.

For the **Clock Bounded** iteration, the following step was undertaken in synthesis, aimed at generating quantitative benchmarks for comparative analysis with the other configurations.

6. Clock constraints were established by creating a symbolic clock signal, binding it to the actual clock pin, and defining the frequency constraints. Initially, a clock period of zero was specified to determine the theoretical minimum clock period through the `create_clock -name "CK" -period 0 "CLK"` command.

Based on previous outcomes, a first clock period constraint was established and the following steps were undertaken.

7. Buffering and DC optimization were disabled for the clock signal, achieved by the command `set_dont_touch_network CK`.
8. Clock jitter constraints were also defined with `set_clock_uncertainty 0.07 [get_clocks CK]`.

Upon executing the `compile_ultra -timing_high_effort_script` command to initiate the optimized synthesis, it becomes crucial to scrutinize the resultant metrics through specific commands:

- `report_area` enumerates the design's combinational and non-combinational areas, along with their aggregate.
- `report_timing` serves to reveal the outcome of the timing analysis undertaken during synthesis. Notably, it ascertains whether the design's longest path adheres to the clock frequency constraint specified earlier. The status is explicitly marked as either *VIOLATED* or *MET*. In either scenario, the slack value signals by what extent the frequency needs adjustment to realize the design's minimum permissible clock period.
- `report_power` is employed for quantifying the design's power usage.

6.3 Results

A slack of -0.68 ns was observed during synthesis with a clock period constraint of zero: this corresponds to a theoretical maximum frequency of 1.47 GHz. Than different versions of the architecture were synthesized with clock bounds ranging from 0.7 ns to 0.3 ns. For additional context, metrics related to area and timing are detailed in Table 6.1, while metrics concerning power are provided in Table 6.2.

Version	**Frequency**	**Area**	**Slack**
Unconstrained	-	14021.92 μm^2	-
CK 0 ns	∞	4558.17 μm^2	-0.68 ns
CK 0.3 ns	3.3 GHz	4590.89 μm^2	-0.10 ns
CK 0.4 ns	2.5 GHz	4588.23 μm^2	-0.02 ns
CK 0.5 ns	2.0 GHz	4559.51 μm^2	0.00 ns
CK 0.6 ns	1.7 GHz	4558.44 μm^2	0.01 ns
CK 0.7 ns	1.4 GHz	4558.18 μm^2	0.02 ns

Table 6.1: Comparison between areas and timing

Version	**Leakage Power**	**Dynamic Power**	**Total Power**
Unconstrained	259.67 μW	997.83 μW	1257.50 μW
CK 0 ns	93.28 μW	95.12 μW	188.40 μW
CK 0.3 ns	95.21 μW	5211.7 μW	5306.91 μW
CK 0.4 ns	95.36 μW	4100.1 μW	4195.43 μW
CK 0.5 ns	94.46 μW	3119.9 μW	3214.36 μW
CK 0.6 ns	94.44 μW	2600.3 μW	2694.74 μW
CK 0.7 ns	94.38 μW	2228.6 μW	2322.98 μW

Table 6.2: Comparison between power consumptions

Area The area metrics reveal some intriguing patterns. The **Unconstrained** design unsurprisingly yields the largest area, serving as a reference point for the architectural design before any optimizations. Interestingly, area usage seems to reach a plateau in the clock-bounded versions from 0.5 ns to 0.7 ns, hovering around 4558 μm^2. These findings suggest that constraints on the clock period have a minimal impact on the chip area beyond the initial clockless design, thus showing the design's resilience to area-based optimizations.

Timing Concerning the timing, the clock-bound versions of the design consistently show an improvement in slack as the clock constraint becomes less stringent. The slack in the design with a 0 ns clock constraint is, as previously mentioned, at a negative value of -0.68 ns. This indicates that this particular design configuration is clearly infeasible without adjustments. However, as the clock constraint is relaxed, the slack gradually approaches a positive value, reaching 0 ns at a constraint of 0.5 ns and becoming positive thereafter. The corresponding frequencies also adjust in a manner that is intuitively sensible, decreasing from a theoretically infinite value (which is unattainable) to more practical bounds as the slack improves.

Power When examining power consumption, the CK 0.3 ns version stands out due to its exceptionally high total power consumption, reaching up to 5306.91 μW. This substantial consumption is primarily driven by its dynamic power, suggesting that operating at such a high frequency of 3.3 GHz incurs significant energy costs. In stark contrast, the design with a clock period constraint of 0 ns is remarkably efficient, consuming only 188.40 μW in total. However, this efficiency is likely attributable to the very low power consumption of the architecture's registers. Since they are connected to a clock with a period of 0 ns, their power consumption is not calculated realistically. Interestingly, leakage power remains largely consistent across the different clock-bounded versions, indicating that the dynamic power is the primary variable influenced by clock period constraints. These findings provide invaluable insights into the power-performance trade-offs inherent in the DLX architecture, and merit further exploration for optimization opportunities.

CHAPTER 7

Physical Design

7.1 Scripts

The completion of this project entails converting the synthesized netlists into a physical layout for the DLX processor, using Cadence Innovus as discussed in Section 1.3.

As with the simulation and synthesis phases, a script named **runphy.sh** was developed (see Appendix A.3.1). This script initializes the working environment and automatically retrieves the updated files for placement and routing from the folders designated for architecture synthesis.

To execute the script, navigate to the appropriate directory, specifically `dlx/phy`, and run the following command:

```
> ./runphy.sh
```

Note that this script does not automatically initiate the design-generating software. To launch Innovus, you must manually enter the following command in the terminal:

```
> innovus
```

Within Innovus, a significant portion of the design process was automated using logs generated by the software. By executing three simple commands, one can complete the design generation.

The first command runs the `dlx.globals` script (see Appendix A.3.1) that was provided as part of the course materials to import the design:

```
> source dlx.globals
```

The second command executes the `phy.tcl` script (see Appendix A.3.3) to generate the actual physical design:

```
> source phy.tcl
```

Lastly, the command `analyses.tcl` (see Appendix A.3.4) is used to generate the required reports:

```
> source analyses.tcl
```

7.2 Workflow

The overall procedure was conducted in alignment with the guidelines and protocols established in the corresponding laboratory session. The comprehensive workflow included several critical steps, each essential for successful physical design implementation. These phases are detailed as follows:

- **Design importation**: this is the initial phase where Innovus is configured to locate the paths to the libraries essential for the design. The synthesized description of the circuit is read, and a configuration file is imported, setting the correct references to the libraries. The design is then fully imported, complete with the correct cell references and the RTL Verilog description.
- **Floorplan construction**: in this stage, Innovus establishes the area that will be designated for the cells, as well as the space for power supply routing. Parameters like Core aspect ratio and Utilization are specified, and the core margins are defined. This lays the groundwork for how cells will be placed in the layout.
- **Power network planning and routing**: this phase involves creating power and ground rings around the core boundary using high metal layers to minimize congestion. The channel defined will be filled with two metal rings for power and ground, respectively. Additional vertical stripes are added to better distribute power across the chip.
- **Cell arrangement**: during this phase, all the standard cells are systematically placed within the predefined rows of the floorplan. Care is taken to place the cells away from power and ground stripes to avoid congestion issues.
- **Signal trace routing**: here, the logical connections between the cell pins are turned into actual metal traces. This stage involves defining the routes for signal propagation throughout the chip. Various metal layers are employed to ensure efficient connections while minimizing the likelihood of issues such as signal interference or routing congestion.
- **Timing analysis and design evaluation**: the final phase focuses on optimizing the design to meet timing constraints. After the routing is done, Clock-Tree-Synthesis (CTS) and Post-CTS optimizations can be performed to make sure that the design meets the specified timing requirements.

Each of these steps contributes to the successful realization of the chip's physical design, ensuring not only its functional capabilities but also its operational efficiency.

7.3 Results

The culmination of the physical design process is captured in a comprehensive post-routing schematic. A detailed graphical representation of this final layout is presented in Figure 7.1.

Area Complementing the schematic representation, a set of quantitative metrics has also been assembled to provide additional layers of evaluation. These metrics offer a numerical perspective on the design's complexities and efficiencies. The salient details concerning the gate count, the number of cells utilized, and the total area occupied by each hierarchical level of modules in the design are summarized in Table 7.1.

This analysis offers valuable insights into the distribution of gates, cells and area across the hierarchical design modules. With an overall gate count of 16992 and an aggregated physical design area of 13560.1 μm^2, the architecture exhibits a relatively small hardware footprint. The Datapath module accounts for almost the entire area, constituting approximately 98% of the overall gate count and physical area. Within this module, the Register File and Arithmetic Logic Unit (ALU) are the dominant sub-modules, occupying 5056.9 μm^2 and 3863.4 μm^2 of area respectively.

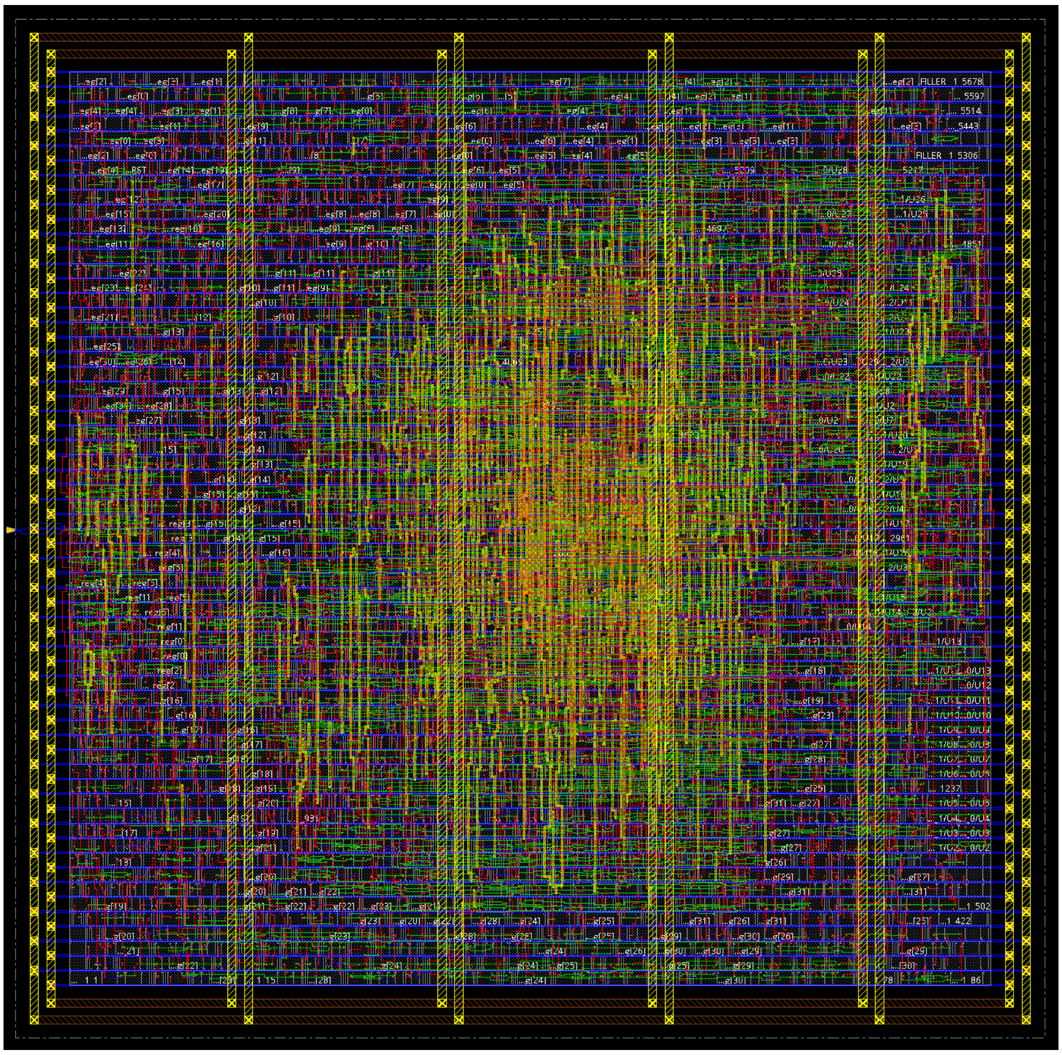

Figure 7.1: Finalized schematic illustrating the placement and routing outcome

Level	Module	Gates	Cells	Area [μm^2]
0	DLX	16992	9127	13560.1
1	CU	276	165	220.8
1	DATAPATH	16708	8959	13333.5
2	DATAPATH/RF	6337	2755	5056.9
2	DATAPATH/ALU	4841	3592	3863.4
3	DATAPATH/ALU/MUL	1792	1244	1430.3
3	DATAPATH/ALU/SUM	772	566	616.1

Table 7.1: Detailed summary of gate counts and area measurements

Upon further examination of the ALU, it is noteworthy that the Multiplier (MUL) sub-module alone occupies around 1430.3 μm^2, highlighting the complexity associated with implementing multiplication operations. The ALU's P4 Adder (SUM) sub-module also proves to be a significant component of the ALU, ranking second in terms of area with 3863.4 μm^2.

The Control Unit (CU) is exceptionally compact, featuring only 276 gates and 220.8 μm^2 of area. This underscores the efficiency of the control logic in the DLX architecture.

Interestingly, the average gate area is calculated to be 0.7980 μm^2, serving as a benchmark for evaluating the granularity of the design.

Timing The Tables 7.2 and 7.3 provide a comprehensive evaluation of the setup and hold times for various routing modes, both pre- and post-optimization. Prior to optimization, the Worst Negative Slack (WNS) in the setup mode for **all** paths is marginally better than the **default** mode at 0.014 ns and also the **reg2reg** mode shows a slightly higher value of 0.090 ns. No timing violations were reported. After Clock Tree Synthesis (CTS), the setup time improved slightly in the **all** and **default** modes to 0.008 ns.

In the hold mode, a single violating path was observed pre-optimization for the **all** and **default** modes, indicating a minor issue with data stability. This was resolved post-optimization as evidenced by the absence of violating paths and a WNS of 0.000 ns. The **reg2reg** mode was robust in both scenarios, demonstrating no hold time violations.

Setup Mode	all	reg2reg	default
WNS [ns]:	0.014	0.090	0.014
TNS [ns]:	0.000	0.000	0.000
Violating Paths:	0	0	0
All Paths:	224	54	217
Hold Mode	all	reg2reg	default
WNS [ns]:	-0.001	0.008	-0.001
TNS [ns]:	-0.001	0.000	-0.001
Violating Paths:	1	0	1
All Paths:	224	54	217

Table 7.2: optDesign postRoute summary

Setup Mode	all	reg2reg	default
WNS [ns]:	0.008	0.087	0.008
TNS [ns]:	0.000	0.000	0.000
Violating Paths:	0	0	0
All Paths:	224	54	217
Hold Mode	all	reg2reg	default
WNS [ns]:	0.000	0.009	0.000
TNS [ns]:	0.000	0.000	0.000
Violating Paths:	0	0	0
All Paths:	224	54	217

Table 7.3: optDesign postCTS summary

These observations underscore the efficacy of the optimization strategies implemented, especially in resolving the minor hold time violation while maintaining excellent setup times.

CHAPTER 8

Further Implementation

As the DLX architecture presented in this report serves as a baseline, there exist several avenues for further enhancement and customization. This chapter aims to explore potential areas where the architecture could be augmented to improve performance, functionality and automation. Topics covered include branch prediction, out-of-order execution, floating-point operations, ISA extensions and more advanced automation of simulation and design stages.

8.1 Branch Prediction

Effective branch prediction is pivotal for achieving high instruction throughput and minimizing pipeline stalls. Both static and dynamic techniques can be employed to optimize the accuracy of branch prediction, each with its own set of trade-offs in terms of hardware complexity and performance.

Static techniques are usually employed at the compiler level and do not require significant changes to the hardware. Some commonly used static methods include:

- **Always Taken**: this is the simplest form of branch prediction where the outcome is always assumed to be taken. Even with this straightforward approach, notable improvements can be achieved, especially in loops where the branch is often taken. Implementing the 'Always Taken' strategy would require minimal changes to the hardware, such as setting a single default bit in the control logic.
- **Branch Direction**: the compiler could assumes that forward branches are not taken and backward branches are taken. This heuristic is based on common programming constructs: forward branches often correspond to conditional statements, while backward branches are usually part of loops. This strategy improves prediction accuracy with no additional hardware requirements.

Dynamic techniques require hardware support for maintaining and updating prediction data. These methods are generally more accurate but add complexity to the hardware design. Notable dynamic techniques include:

- **One-Bit Prediction**: a single bit is used to predict the outcome based on the most recent execution. A '1' might indicate that the branch is likely to be taken, and a '0' would indicate otherwise. This scheme can also be extended to use two bits for a more reliable prediction.
- **Branch History Table (BHT)**: the BHT keeps a history of the outcomes of recent branches and uses this data to predict future branches.

- **Branch Target Buffer (BTB)**: the BTB stores the target addresses of recently taken branches. When a branch instruction is encountered, the BTB is looked up to fetch the likely target address, speeding up the instruction fetch stage.

By focusing on implementing the 'Always Taken' technique as a starting point, one can achieve a balance between improved performance and hardware simplicity. This strategy serves as an excellent stepping stone for more advanced branch prediction techniques that can be incorporated in future iterations of the architecture.

8.2 Out-of-Order Execution

The existing architecture utilizes a conventional 5-stage pipeline comprising instruction fetch, decode, execute, memory access and write-back stages. While this in-order execution model is straightforward and effective for many applications, it leaves room for optimization, particularly in scenarios involving data hazards or long-latency operations.

To transition to an out-of-order execution model, several significant hardware modifications would be required:

- **Reorder Buffer**: A Reorder Buffer (ROB) would be needed to keep track of the original program order of instructions. This allows for the instructions to be executed out-of-order but committed (written back) in-order, ensuring program correctness. The ROB stores information about the instruction, its status, and the destination for the result.
- **Reservation Stations**: Reservation stations would be introduced to hold instructions that are ready to be executed but are waiting for the availability of operand data. Each reservation station would be associated with a specific functional unit, like an ALU or FPU, and would store the opcode and operands for each instruction.
- **Issue Logic**: A more complex issue logic would be required to scan the reservation stations and determine which instructions can be issued for execution based on operand availability. This logic would also handle the forwarding of results to dependent instructions.
- **Register Renaming**: To alleviate data hazards caused by the limited number of physical registers, register renaming would be implemented. This technique dynamically assigns physical registers to logical registers to avoid write-after-read and write-after-write hazards.
- **Functional Units**: Additional functional units may be required to support parallel execution of multiple instructions. This could include extra ALUs, FPUs, or specialized units for complex operations.
- **Complex Control Unit**: The control unit would become significantly more complicated, as it would now have to manage the dynamic scheduling of instructions, handle data hazards, and control multiple functional units. Advanced algorithms would be employed to efficiently manage these tasks.

8.3 Floating-Point Operations

The current architecture lacks support for floating-point operations, which are essential for scientific computing, graphics, and other high-precision tasks. Implementing floating-point support would involve several significant hardware-level changes:

- **Floating-Point Unit (FPU)**: a dedicated FPU would need to be integrated into the existing pipeline. This unit would handle floating-point arithmetic operations like addition, subtraction, multiplication, and division. Depending on the performance requirements, it might be beneficial to include multiple FPUs to support parallel execution of floating-point instructions.
- **Register File Extension**: the existing register file would need to be extended or an entirely new Register File would need to be created to store floating-point numbers. These registers would hold the operands and results of floating-point operations.
- **Instruction Set Architecture (ISA) Extensions**: the ISA would need to be extended to include new floating-point instructions. This would require updating also the Control Unit to recognize and handle these new instructions appropriately.

By implementing these hardware-level changes, the architecture would gain the capability to perform high-precision floating-point operations, thereby extending its applicability to a broader range of computational tasks.

8.4 ISA Extensions

The current Instruction Set Architecture (ISA) is designed for general-purpose computing but has potential for enhancements to meet specialized computational needs. Extending the ISA would necessitate various hardware-level modifications, which would entail strengthening all the architectures present within the processor.

Unimplemented instructions: `MOVI2S`, `MOVS2I`, `MOVF`, `MOVD`, `MOVFP2I`, `MOVI2FP`, `MOVI2T`, `MOVT2I`, `ADDF`, `SUBF`, `MULTF`, `DIVF`, `ADDD`, `SUBD`, `MULTD`, `DIVD`, `CVTF2D`, `CVTF2I`, `CVTD2F`, `CVTD2I`, `CVTI2F`, `CVTI2D`, `DIV`, `EQF`, `NEF`, `LTF`, `GTF`, `LEF`, `GEF`, `DIVU`, `EQD`, `NED`, `LTD`, `GTD`, `LED`, `GED`, `BFPT`, `BFPF`, `RFE`, `TRAP`, `LB`, `LH`, `LBU`, `LHU`, `LF`, `LD`, `SB`, `SH`, `SF`, `SD` and `ITLB`.

8.5 Script Enhancement

The current design flow incorporates various scripts that automate tasks across the simulation, synthesis and physical design phases. Although effective, there is room for additional improvements to establish a more integrated and robust automation framework. Below are some considerations and potential enhancements:

- **Parameterization**: to facilitate the testing of different hardware parameters, particularly during the synthesis and physical design phases, the scripts could be extended to accept configurable parameters. This would enable quick exploration of the design space.
- **Error Handling and Reporting**: incorporating error-handling mechanisms into the scripts would improve the robustness of the automation process. Additionally, generating logs or reports after each phase could prove beneficial for debugging and performance analysis.
- **Automated Testing**: integrating automated test suites into the scripting environment would help verify the design's correctness after each modification, thereby enhancing the overall reliability of the project.

By implementing these enhancements, the automation framework would not only expedite prototyping but also provide a more robust and configurable environment for testing and validation, particularly in the synthesis and physical design stages.

8.6 Conclusion

Implementing these improvements would not only augment the processor's performance but also broaden its application spectrum. However, it's important to note that these modifications would substantially increase area and power consumption of the datapath and the processor as a whole.

Bibliography

[1] M. Graziano. "*Microelectronic systems*". Politecnico di Torino, Master's degree in Computer Engineering. Torino, March - June 2021. Videolectures.

[2] J. Hennessy, D. Patterson. "*Computer Architecture, Fifth Edition: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*" (5th ed.) Morgan Kauf- mann Publishers Inc. San Francisco, CA, USA, 2011. Book.

[3] D. Patterson, J. Hennessy. "*Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface*" (4th ed.) Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2008. Book.

APPENDIX A

Scripts

A.1 Simulation

A.1.1 assembler.sh

```
#!/bin/bash

if [ ! -r $1 ]
then
        echo "Usage:␣$0␣<dlx_assembly_file>.asm"
        exit 1
fi

asmfile=‘echo $1 | sed s/[.].*//g | sed ’s/asm_example\///’‘
perl ./assembler.bin/dlxasm.pl -o $asmfile.bin -list $asmfile.list $1
rm $asmfile.bin.hdr
cat $asmfile.bin | hexdump -v -e ’/1␣"%02X"␣/1␣"%02X"␣/1␣"%02X"␣/1␣"%02X\n"’ > firmware.asm.mem
rm $asmfile.bin
rm $asmfile.list
```

A.1.2 runsim.sh

```
#!/bin/bash

#-----------------------------------------------------------------------------------------------
# Description:  This bash script manages the simulation process for the DLX project. It allows
#               users to select the test file for simulation and choose whether to run the
#               simulation in the background.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/19
#-----------------------------------------------------------------------------------------------

# Initialize default values for variables
filename=test.asm
background=y

# Parse command-line options for test filename and background execution
while getopts b:f: flag
do
        case "${flag}" in
                b) background=${OPTARG};;       # Set whether to run in the background
                f) filename=${OPTARG};;         # Set the test filename
        esac
done

# Initialize environment
source /eda/scripts/init_questa_core_prime &> /dev/null #setmentor
```

```
# Create a new working directory
vlib work

# Run assembler script for the chosen test file
source ./assembler.sh asm_example/$filename > /dev/null

#./assembler.bin/dlxasm.pl asm_example/$filename
#./assembler.bin/conv2memory asm_example/{$filename}.exe > {$filename}.txt

sleep 1

# Execute the simulation based on the background option; within the "QuestaSim" program
# it is possible to re-run the simulation simply using the "do resim.tcl" command
if [ $background == "y" ]
then

        # Execute the simulation in the background
        vsim -do sim.tcl &

elif [ $background == "n" ]
then

        # Execute the simulation in the foreground
        vsim -do sim.tcl

fi


# Wait for the last background job to finish (if any)
wait $!

# Cleanup
rm -r -f work
rm firmware.asm.mem
rm transcript
```

A.1.3 resim.tcl

```
quit -sim

vdel -all -lib work
vlib work

do sim.tcl
```

A.1.4 sim.tcl

```
#-----------------------------------------------------------------------------------------------
# Description:  This module is responsible for setting up the simulation environment in QuestaSim
#               for the DLX microarchitecture. It compiles VHDL files related to the various
#               components of DLX and sets up simulation waveforms based on user-specified
#               configurations for better debugging. Users can select between different waveform
#               settings and focus on specific pipeline stages for detailed observations.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/19
#-----------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------
# Control Panel
#-----------------------------------------------------------------------------------------------

# User-configurable settings for simulation

# 0 = DLX; 1 = CONTROL UNIT; 2 = DATAPATH; 3 = IRAM ; 4 = DRAM; 5 = ALU
quietly set dlx_unit 0
```

```
# 0 = All; 1 = Fetch; 2 = Decode; 3 = Execute; 4 = Memory; 5 = Write Back
quietly set pipe_stage 0

#-----------------------------------------------------------------------------------------------
# Compile VHDL files
#-----------------------------------------------------------------------------------------------

# Constants & functions (000)
vcom -quiet ../src/000-functions.vhd
vcom -quiet ../src/000-globals.vhd

## Basic Logic Gates
vcom -quiet ../src/000-globals/not.vhd
vcom -quiet ../src/000-globals/and.vhd
vcom -quiet ../src/000-globals/nand.vhd
vcom -quiet ../src/000-globals/or.vhd
vcom -quiet ../src/000-globals/nor.vhd
vcom -quiet ../src/000-globals/xor.vhd
vcom -quiet ../src/000-globals/xnor.vhd

## Memory Elements
vcom -quiet ../src/000-globals/ffd.vhd
vcom -quiet ../src/000-globals/ffdr.vhd
vcom -quiet ../src/000-globals/ld.vhd
vcom -quiet ../src/000-globals/ldr.vhd

## Multiplexers
vcom -quiet ../src/000-globals/mux21.vhd
vcom -quiet ../src/000-globals/mux21_logic.vhd
vcom -quiet ../src/000-globals/mux41.vhd
#vcom -quiet ../src/000-globals/mux41_logic.vhd
vcom -quiet ../src/000-globals/mux81.vhd
#vcom -quiet ../src/000-globals/mux81_logic.vhd

# Arithmetic Components
vcom -quiet ../src/000-globals/fa.vhd
vcom -quiet ../src/000-globals/rca.vhd
vcom -quiet ../src/000-globals/zero_detector.vhd

## Control Unit (a.a)
vcom -quiet ../src/a.a-cu.vhd

## Datapath (a.b)
vcom -quiet ../src/a.b-datapath.vhd
vcom -quiet ../src/a.b.a-register_file.vhd
vcom -quiet ../src/a.b.b-sign_extend.vhd
vcom -quiet ../src/a.b.c-register_addresser.vhd
vcom -quiet ../src/a.b.e-forwarding_unit.vhd

### ALU (a.b.d)
vcom -quiet ../src/a.b.d-alu.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.c-comparator.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.d-logic.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.e-barrel_shifter_left.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.f-barrel_shifter_right.vhd

#### P4 Adder (a.b.d.a)
vcom -quiet ../src/a.b.d-alu/a.b.d.a-p4_adder.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.a.d-pg_block.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.a.c-propagate.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.a.b-generate.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.a.a-pg_row.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.a-carry_generator.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.b.a-carry_select_block.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.a.b-sum_generator.vhd

#### Booth Multiplier (a.b.d.b)
vcom -quiet ../src/a.b.d-alu/a.b.d.b-booth_multiplier.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.b.a-booth_encoder.vhd
vcom -quiet ../src/a.b.d-alu/a.b.d.b.a.a-encoder.vhd

## IRAM (a.c)
vcom -quiet ../src/a.c-iram.vhd
```

```
## DRAM (a.d)
vcom -quiet ../src/a.d-dram.vhd

# DLX (a)
vcom -quiet ../src/a-dlx.vhd

# Testbench
#vcom -quiet ../src/test_bench/tb-datapath.vhd
vcom -quiet ../src/test_bench/tb-dlx.vhd

# Simulation
vsim -quiet -t 10ps work.CFG_TB -voptargs=+acc

#-----------------------------------------------------------------------------------------------
# Waveform Setup
#-----------------------------------------------------------------------------------------------

# Configure signal name width in waveform for better readability
config wave -signalnamewidth 1

# CONTROL UNIT specific waves

if {$dlx_unit eq 0 || $dlx_unit eq 1} {

        add wave -height 50 -divider {CONTROL UNIT} -height 19 \
        -color white sim:/tb_dlx/U1/CLK \
        -color gray  sim:/tb_dlx/U1/RST

        add wave -divider {INPUTS} -position insertpoint -radix hex \
        sim:/tb_dlx/U1/CU_I/IR_IN \
        sim:/tb_dlx/U1/CU_I/IR_opcode \
        sim:/tb_dlx/U1/CU_I/IR_func

        add wave -divider {ALU} -position insertpoint \
        sim:/tb_dlx/U1/CU_I/aluOpcode1 \
        sim:/tb_dlx/U1/CU_I/aluOpcode2 \
        sim:/tb_dlx/U1/CU_I/aluOpcode3 \
        sim:/tb_dlx/U1/CU_I/ALU_OPCODE

        add wave -divider {CONTROL WORDS} -position insertpoint -radix binary \
        sim:/tb_dlx/U1/CU_I/cw1 \
        sim:/tb_dlx/U1/CU_I/cw2 \
        sim:/tb_dlx/U1/CU_I/cw3 \
        sim:/tb_dlx/U1/CU_I/cw4 \
        sim:/tb_dlx/U1/CU_I/cw5

        add wave -divider {OUTPUTS - FETCH} -position insertpoint \
        sim:/tb_dlx/U1/CU_I/IR_LATCH_EN \
        sim:/tb_dlx/U1/CU_I/PC_LATCH_EN \
        sim:/tb_dlx/U1/CU_I/NPC_LATCH_EN

        add wave -divider {OUTPUTS - DECODE} -position insertpoint \
        sim:/tb_dlx/U1/CU_I/RF_WE \
        sim:/tb_dlx/U1/CU_I/RegA_LATCH_EN \
        sim:/tb_dlx/U1/CU_I/RegB_LATCH_EN \
        sim:/tb_dlx/U1/CU_I/RegIMM_LATCH_EN

        add wave -divider {OUTPUTS - EXECUTE} -position insertpoint \
        sim:/tb_dlx/U1/CU_I/MUXA_SEL \
        sim:/tb_dlx/U1/CU_I/MUXB_SEL \
        sim:/tb_dlx/U1/CU_I/ALU_OUTREG_EN \
        sim:/tb_dlx/U1/CU_I/EQ_COND \
        sim:/tb_dlx/U1/CU_I/ALU_OPCODE

        add wave -divider {OUTPUTS - MEMORY} -position insertpoint \
        sim:/tb_dlx/U1/CU_I/DRAM_RE \
        sim:/tb_dlx/U1/CU_I/DRAM_WE \
        sim:/tb_dlx/U1/CU_I/LMD_LATCH_EN \
        sim:/tb_dlx/U1/CU_I/JUMP_EN \
        sim:/tb_dlx/U1/CU_I/JUMP_COND

        add wave -divider {OUTPUTS - WRITE BACK} -position insertpoint \
        sim:/tb_dlx/U1/CU_I/WB_MUX_SEL \
        sim:/tb_dlx/U1/CU_I/JAL_MUX_SEL
```

```
}

# DATAPATH specific waves

if {$dlx_unit eq 0 || $dlx_unit eq 2} {

        # FETCH (IF)

        if {$pipe_stage eq 0 || $pipe_stage eq 1} {

                add wave -height 50 -divider {FETCH (IF)} -height 19 \
                -color white sim:/tb_dlx/U1/CLK \
                -color gray  sim:/tb_dlx/U1/RST

                add wave -divider {IF Control Signals} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IR_LATCH_EN \
                sim:/tb_dlx/U1/DATAPATH_I/PC_LATCH_EN \
                sim:/tb_dlx/U1/DATAPATH_I/NPC_LATCH_EN

                add wave -divider {STAGE} -position insertpoint -radix binary \
                sim:/tb_dlx/U1/DATAPATH_I/IR_OUT

                add wave -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ALU_LABEL

                add wave -divider {PC_MUX} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/NPC_BUS \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_ALU_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/PC_MUX_SEL \
                sim:/tb_dlx/U1/DATAPATH_I/PC_BUS

                add wave -divider {PROGRAM_COUNTER} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/PC_LATCH_EN \
                sim:/tb_dlx/U1/DATAPATH_I/PC_BUS \
                sim:/tb_dlx/U1/DATAPATH_I/PC_OUTPUT

                add wave -divider {NEXT_PROGRAM_COUNTER} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/NPC_LATCH_EN \
                sim:/tb_dlx/U1/DATAPATH_I/PC_BUS \
                sim:/tb_dlx/U1/DATAPATH_I/NPC_OUT

                add wave -divider {INSTRUCTION_REGISTER} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IR_BUS \
                sim:/tb_dlx/U1/DATAPATH_I/IR_OUT

                add wave -divider {IF-ID Pipeline} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_IR

        }

        # DECODE (ID)

        if {$pipe_stage eq 0 || $pipe_stage eq 2} {

                add wave -height 50 -divider {DECODE (ID)} -height 19 \
                -color white sim:/tb_dlx/U1/CLK \
                -color gray  sim:/tb_dlx/U1/RST

                add wave -divider {ID CONTROL SIGNALS} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/RegA_LATCH_EN \
                sim:/tb_dlx/U1/DATAPATH_I/RegB_LATCH_EN \
                sim:/tb_dlx/U1/DATAPATH_I/RegIMM_LATCH_EN

                add wave -divider {STAGE} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_IR \
                sim:/tb_dlx/U1/DATAPATH_I/ID_ALU_LABEL

                add wave -divider {REGISTER_ADDRESSER} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_IR \
                sim:/tb_dlx/U1/DATAPATH_I/REGISTER_ADDRESSER/IR_OPC \
                sim:/tb_dlx/U1/DATAPATH_I/RS1 \
                sim:/tb_dlx/U1/DATAPATH_I/RS2 \
                sim:/tb_dlx/U1/DATAPATH_I/RD

                add wave -divider {REGISTER_FILE} -position insertpoint -radix decimal \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/RS1 \
                sim:/tb_dlx/U1/DATAPATH_I/RS2 \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RD \
                sim:/tb_dlx/U1/DATAPATH_I/WB_MUX_OUT \
                sim:/tb_dlx/U1/DATAPATH_I/RF_OUT1 \
                sim:/tb_dlx/U1/DATAPATH_I/RF_OUT2 \
                sim:/tb_dlx/U1/DATAPATH_I/REGISTER_FILE/REG

                add wave -divider {SIGN_EXTEND} -position insertpoint -radix decimal \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_IR \
                sim:/tb_dlx/U1/DATAPATH_I/SIGN_EXTEND/IR_OPC \
                sim:/tb_dlx/U1/DATAPATH_I/SIGN_EXTEND/IMM_I \
                sim:/tb_dlx/U1/DATAPATH_I/SIGN_EXTEND/IMM_J \
                sim:/tb_dlx/U1/DATAPATH_I/IMM_OUT

                add wave -divider {IF-ID Pipeline} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/IF_ID_IR

                add wave -divider {ID-EX Pipeline} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RS1 \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RS2 \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RD \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT1 \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT2 \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_IMM

        }

        # EXECUTE (EX)

        if {$pipe_stage eq 0 || $pipe_stage eq 3} {

                add wave -height 50 -divider {EXECUTE (EX)} -height 19 \
                -color white sim:/tb_dlx/U1/CLK \
                -color gray  sim:/tb_dlx/U1/RST

                add wave -divider {EX CONTROL SIGNALS} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/MUXA_SEL \
                sim:/tb_dlx/U1/DATAPATH_I/MUXB_SEL \
                sim:/tb_dlx/U1/DATAPATH_I/ALU_OUTREG_EN \
                sim:/tb_dlx/U1/DATAPATH_I/EQ_COND \
                sim:/tb_dlx/U1/DATAPATH_I/ALU_OPCODE

                add wave -divider {STAGE} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/EX_ALU_LABEL

                add wave -divider {FORWARDING UNIT} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RS1 \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RS2 \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RD \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RD \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/FORWARD_A \
                sim:/tb_dlx/U1/DATAPATH_I/FORWARD_B

                add wave -divider {ALU_PRE_MUX1} -position insertpoint -radix decimal \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT1 \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_ALU_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/JAL_MUX_OUT \
                sim:/tb_dlx/U1/DATAPATH_I/FORWARD_A \
                sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP1

                add wave -divider {ALU_MUX1} -position insertpoint -radix decimal \
                sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP1 \
                sim:/tb_dlx/U1/DATAPATH_I/ID_EX_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/MUXA_SEL \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_OP1

		add wave -divider {ALU_PRE_MUX2} -position insertpoint -radix decimal \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT2 \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_ALU_OUTPUT \
		sim:/tb_dlx/U1/DATAPATH_I/JAL_MUX_OUT \
		sim:/tb_dlx/U1/DATAPATH_I/FORWARD_B \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP2

		add wave -divider {ALU_MUX2} -position insertpoint -radix decimal \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP2 \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_IMM \
		sim:/tb_dlx/U1/DATAPATH_I/MUXB_SEL \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_OP2

		add wave -divider {ALU} -position insertpoint -radix decimal \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_OP1 \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_OP2 \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_OPCODE \
		sim:/tb_dlx/U1/DATAPATH_I/ALU_OUTPUT

		add wave -divider {ZERO_DETECTOR} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT1 \
		sim:/tb_dlx/U1/DATAPATH_I/ZERO_OUT \
		sim:/tb_dlx/U1/DATAPATH_I/ZERO_OUT_NEG \
		sim:/tb_dlx/U1/DATAPATH_I/BRANCH_DETECT

		add wave -divider {ID-EX Pipeline} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_NPC \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_WE \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RS1 \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RS2 \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RD \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT1 \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT2 \
		sim:/tb_dlx/U1/DATAPATH_I/ID_EX_IMM

		add wave -divider {EX-MEM Pipeline} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_NPC \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_BRANCH_DETECT \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_ALU_OUTPUT \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RF_WE \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RD \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RF_OUT2

	}

	# MEMORY (MEM)

	if {$pipe_stage eq 0 || $pipe_stage eq 4} {

		add wave -height 50 -divider {MEMORY (MEM)} -height 19 \
		-color white sim:/tb_dlx/U1/CLK \
		-color gray  sim:/tb_dlx/U1/RST

		add wave -divider {MEM CONTROL SIGNALS} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/DRAM_RE \
		sim:/tb_dlx/U1/DATAPATH_I/DRAM_WE \
		sim:/tb_dlx/U1/DATAPATH_I/LMD_LATCH_EN\
		sim:/tb_dlx/U1/DATAPATH_I/JUMP_EN \
		sim:/tb_dlx/U1/DATAPATH_I/JUMP_COND

		add wave -divider {STAGE} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/MEM_ALU_LABEL

		add wave -divider {JUMP} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_BRANCH_DETECT \
		sim:/tb_dlx/U1/DATAPATH_I/JUMP_EN \
		sim:/tb_dlx/U1/DATAPATH_I/BRANCH_COND \
		sim:/tb_dlx/U1/DATAPATH_I/JUMP_COND \
		sim:/tb_dlx/U1/DATAPATH_I/PC_MUX_SEL

		add wave -divider {LATCHES} -position insertpoint \
		sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_DRAM_OUTPUT_NEXT
```

```
                add wave -divider {EX-MEM Pipeline} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_BRANCH_DETECT \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_ALU_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RD \
                sim:/tb_dlx/U1/DATAPATH_I/EX_MEM_RF_OUT2

                add wave -divider {MEM-WB Pipeline} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_DRAM_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_ALU_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RD

        }

        # WRITE BACK (WB)

        if {$pipe_stage eq 0 || $pipe_stage eq 5} {

                add wave -height 50 -divider {WRITE BACK (WB)} -height 19 \
                -color white sim:/tb_dlx/U1/CLK \
                -color gray  sim:/tb_dlx/U1/RST

                add wave -divider {WB CONTROL SIGNALS} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/WB_MUX_SEL \
                sim:/tb_dlx/U1/DATAPATH_I/JAL_MUX_SEL

                add wave -divider {STAGE} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/WB_ALU_LABEL

                add wave -divider {WB MUX} -position insertpoint \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_DRAM_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_ALU_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/WB_MUX_SEL \
                sim:/tb_dlx/U1/DATAPATH_I/WB_MUX_OUT

                add wave -divider {JAL MUX} -position insertpoint -radix decimal \
                sim:/tb_dlx/U1/DATAPATH_I/WB_MUX_OUT \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/JAL_MUX_SEL \
                sim:/tb_dlx/U1/DATAPATH_I/JAL_MUX_OUT

                add wave -divider {MEM-WB Pipeline} -position insertpoint -radix decimal \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_NPC \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_DRAM_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_ALU_OUTPUT \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RF_WE \
                sim:/tb_dlx/U1/DATAPATH_I/MEM_WB_RD

        }

}

# IRAM specific waves

if {$dlx_unit eq 0 || $dlx_unit eq 3} {

        add wave -height 50 -divider {IRAM} -height 19 -position insertpoint \
        -color white sim:/tb_dlx/U1/CLK \
        -color gray  sim:/tb_dlx/U1/RST

        add wave -position insertpoint \
        sim:/tb_dlx/U1/IRAM_I/ADDR \
        sim:/tb_dlx/U1/IRAM_I/DOUT \

        add wave -position insertpoint -radix decimal \
        sim:/tb_dlx/U1/IRAM_I/IRAM_mem

        add wave -divider {STAGE} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/IF_ALU_LABEL
```

```
}

# DRAM specific waves

if {$dlx_unit eq 0 || $dlx_unit eq 4} {

        add wave -height 50 -divider {DRAM} -height 19 -position insertpoint \
        -color white sim:/tb_dlx/U1/CLK \
        -color gray  sim:/tb_dlx/U1/RST \

        add wave -position insertpoint -radix decimal \
        sim:/tb_dlx/U1/DRAM_I/RE \
        sim:/tb_dlx/U1/DRAM_I/WE \
        sim:/tb_dlx/U1/DRAM_I/ADDR \
        sim:/tb_dlx/U1/DRAM_I/DIN \
        sim:/tb_dlx/U1/DRAM_I/DOUT \

        add wave -position insertpoint -radix decimal \
        sim:/tb_dlx/U1/DRAM_I/DRAM_mem

        add wave -divider {STAGE} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/MEM_ALU_LABEL

}

# ALU-specific waves
if {$dlx_unit eq 5} {

        add wave -divider {TB_DATAPATH} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/CLK \
        sim:/tb_dlx/U1/DATAPATH_I/RST

        add wave -divider {EX CONTROL SIGNALS} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/MUXA_SEL \
        sim:/tb_dlx/U1/DATAPATH_I/MUXB_SEL \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_OUTREG_EN \
        sim:/tb_dlx/U1/DATAPATH_I/EQ_COND \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_OPCODE

        add wave -divider {STAGE} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/EX_ALU_LABEL

        add wave -divider {ALU_PRE_MUX1} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/ID_EX_NPC \
        sim:/tb_dlx/U1/DATAPATH_I/MUXA_SEL \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP1

        add wave -divider {ALU_MUX1} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT1 \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP1 \
        sim:/tb_dlx/U1/DATAPATH_I/MUXA_SEL \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_OP1

        add wave -divider {ALU_PRE_MUX2} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/ID_EX_IMM_NEXT \
        sim:/tb_dlx/U1/DATAPATH_I/MUXB_SEL \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP2

        add wave -divider {ALU_MUX2} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/ID_EX_RF_OUT2 \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_PREOP2 \
        sim:/tb_dlx/U1/DATAPATH_I/MUXB_SEL \
        sim:/tb_dlx/U1/DATAPATH_I/ALU_OP2

        add wave -divider {ALU I/O} -position insertpoint -radix decimal \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/OP1 \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/OP2 \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/OPC \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/Y_TMP \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/Y \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/Z

        add wave -divider {P4 ADDER} -position insertpoint -radix decimal \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/SUM/*
        add wave -divider {P4 - CARRY GENERATOR} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/SUM/CARRY_GENERATOR_INSTANCE/*

        add wave -divider {P4 - SUM GENERATOR} -position insertpoint \
        sim:/tb_dlx/U1/DATAPATH_I/ARITHMETIC_LOGIC_UNIT/SUM/SUM_GENERATOR_INSTANCE/*

}

#-------------------------------------------------------------------------------------------------
# Simulation Run
#-------------------------------------------------------------------------------------------------

# Suppress warnings
quietly set StdArithNoWarnings 1
quietly set NumericStdNoWarnings 1

# Run 0 ns of simulation
run 0 ns;

# Re-enable warnings
quietly set StdArithNoWarnings 0
quietly set NumericStdNoWarnings 0

# Run the real simulation
run 100 ns

# Restore cursor and zoom settings for better visibility in wave view
WaveRestoreCursors {0 ns}
#WaveRestoreZoom {0 fs} [simtime]
WaveRestoreZoom {0 ns} {30 ns}

# Uncomment to quit the simulation automatically
#quit
```

A.2 Synthesis

A.2.1 runsyn.sh

```bash
#!/bin/bash

#-----------------------------------------------------------------------------------------------
# Description:  This bash script is designed to manage the synthesis process for the DLX project.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/19
#-----------------------------------------------------------------------------------------------

# Initialize default values for variables
path="./"
background=n
file_name="all"
file_list=()

# Populate file_list with scripts that match the regular expression
for file in $path*.tcl; do
        if [[ $file =~ syn_(.+)\.tcl ]]; then
                file_list+=("$file")
        fi
done

# Parse command-line options for test filename and background execution
while getopts b:f: flag
do
        case "${flag}" in
                b) background=${OPTARG};;       # Set whether to run in the background
                f) file_name=${OPTARG};;        # Set the test filename
        esac
done

# Initialize environment
source /eda/scripts/init_design_vision &> /dev/null #setsynopsys

# Cleanup and create new working directories
#rm -r -f constraints designs reports
mkdir -p constraints designs reports

# Check whether a specific filename was provided or run all by default
if [ $file_name != "all" ]; then
        file_list=($path$file_name)
fi

# Execute the synthesis based on the background option and filename
for file in "${file_list[@]}"
do
        if [ $background == "y" ]; then

                # Execute the simulation in the background
                dc_shell-xg-t -f $file > /dev/null

                # Cleanup and create new working directories
                rm -r -f work
                mkdir -p work

        elif [ $background == "n" ]; then

                # Execute the simulation in the foreground
                dc_shell-xg-t -f $file

                # Cleanup and create new working directories
                rm -r -f work
                mkdir -p work

        fi
done

# Cleanup
```

```
rm -r -f work
rm -f *.log *.svf
```

A.2.2 analyze.tcl

```
#-----------------------------------------------------------------------------------------
# Description:  This TCL script is part of the DLX project and is designed to analyze VHDL modules
#               that are part of the current design. It reads a list of VHDL files to be analyzed
#               from a file named "file_list.lst". The script then iterates through this list,
#               skipping any lines that are either empty or start with a "#" character, and
#               performs VHDL analysis on each file.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/14
#-----------------------------------------------------------------------------------------

# Suppress warning messages
suppress_message ELAB-130       # The initial value for signal '%s' is not supported for synthesis
suppress_message ELAB-311       # DEFAULT branch of CASE statement cannot be reached
suppress_message VHDL-290       # A dummy net '%s' is created to connect open pin '%s'

# Open file_list.lst
set file_id [open "file_list.lst" r]

# Initialize an empty list to hold the filenames
set file_list {}

# Read the file line-by-line
while {[gets $file_id line] >= 0} {

        # Remove whitespace from the line
        set line [string trim $line]

        # Skip empty lines
        if {[string length $line] == 0} {
                continue
        }

        # Skip lines that start with '#'
        if {[string index $line 0] == "#"} {
                continue
        }

        # Add the line to the list
        lappend file_list $line
}

# Close the file
close $file_id

# Perform VHDL analysis on each file
foreach VHD_file $file_list {
        analyze -library WORK -format vhdl $VHD_file
}

# Preserve RTL names in the netlist
set power_preserve_rtl_hier_names true
```

A.2.3 file_list.lst

```
# Constants & functions (000)
../src/000-functions.vhd
../src/000-globals.vhd

## Basic Logic Gates
../src/000-globals/not.vhd
../src/000-globals/and.vhd
../src/000-globals/nand.vhd
```

```
../src/000-globals/or.vhd
../src/000-globals/nor.vhd
../src/000-globals/xor.vhd
../src/000-globals/xnor.vhd

## Memory Elements
../src/000-globals/ffd.vhd
../src/000-globals/ffdr.vhd
../src/000-globals/ld.vhd
../src/000-globals/ldr.vhd

## Multiplexers
../src/000-globals/mux21.vhd
../src/000-globals/mux21_logic.vhd
../src/000-globals/mux41.vhd
#../src/000-globals/mux41_logic.vhd
../src/000-globals/mux81.vhd
#../src/000-globals/mux81_logic.vhd

# Arithmetic Components
../src/000-globals/fa.vhd
../src/000-globals/rca.vhd
../src/000-globals/zero_detector.vhd

## Control Unit (a.a)
../src/a.a-cu.vhd

## Datapath (a.b)
../src/a.b-datapath.vhd
../src/a.b.a-register_file.vhd
../src/a.b.b-sign_extend.vhd
../src/a.b.c-register_addresser.vhd
../src/a.b.e-forwarding_unit.vhd

### ALU (a.b.d)
../src/a.b.d-alu.vhd
../src/a.b.d-alu/a.b.d.c-comparator.vhd
../src/a.b.d-alu/a.b.d.d-logic.vhd
../src/a.b.d-alu/a.b.d.e-barrel_shifter_left.vhd
../src/a.b.d-alu/a.b.d.f-barrel_shifter_right.vhd

#### P4 Adder (a.b.d.a)
../src/a.b.d-alu/a.b.d.a-p4_adder.vhd
../src/a.b.d-alu/a.b.d.a.a.d-pg_block.vhd
../src/a.b.d-alu/a.b.d.a.a.c-propagate.vhd
../src/a.b.d-alu/a.b.d.a.a.b-generate.vhd
../src/a.b.d-alu/a.b.d.a.a.a-pg_row.vhd
../src/a.b.d-alu/a.b.d.a.a-carry_generator.vhd
../src/a.b.d-alu/a.b.d.a.b.a-carry_select_block.vhd
../src/a.b.d-alu/a.b.d.a.b-sum_generator.vhd

#### Booth Multiplier (a.b.d.b)
../src/a.b.d-alu/a.b.d.b-booth_multiplier.vhd
../src/a.b.d-alu/a.b.d.b.a-booth_encoder.vhd
../src/a.b.d-alu/a.b.d.b.a.a-encoder.vhd

## IRAM (a.c)
./memories/a.c-iram.vhd

## DRAM (a.d)
./memories/a.d-dram.vhd

# DLX (a)
../src/a-dlx.vhd
```

A.2.4 syn_unconstrained.tcl

```
#--------------------------------------------------------------------------------------------
# Description:  DLX synthesis unconstrained.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/19
#-------------------------------------------------------------------------------------------

# File Name Extraction
set full_name [info script]
set file_name [file tail $full_name]

# Label Generation
regexp {syn_(.+)\.tcl} $file_name -> label

# Analyze Components
source "./analyze.tcl" > /dev/null

# Select Design
elaborate DLX -architecture DLX_RTL -update -library WORK

# Remove Unconnected Ports
remove_unconnected_ports [get_cells -hierarchical *]

# Set Wire Load
set_wire_load_model -name 5K_hvratio_1_4

# Compile Design
compile

# Reports
report_area > reports/dlx_area_${label}.txt
report_timing > reports/dlx_timing_${label}.txt
report_power > reports/dlx_power_${label}.txt

# Save Design
write -hierarchy -format ddc -output designs/dlx_${label}.ddc

# Generate VHDL and Verilog
write -hierarchy -format vhdl -output designs/dlx_${label}.vhd
write -hierarchy -format verilog -output designs/dlx_${label}.v

# Generate SDC script
write_sdc constraints/dlx_${label}.sdc

quit
```

A.2.5 syn_clock_bound.tcl

```
#-------------------------------------------------------------------------------------------
# Description:  DLX synthesis with bounded clock.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/19
#-------------------------------------------------------------------------------------------

# Control Panel
set period 0.7
set power 20
set uncertainty 0.07

# File Name Extraction
set full_name [info script]
set file_name [file tail $full_name]

# Label Generation
regexp {syn_(.+)\.tcl} $file_name -> label

# Analyze Components
source "./analyze.tcl" > /dev/null

# Select Design
elaborate DLX -architecture DLX_RTL -library WORK

# Remove Unconnected Ports
remove_unconnected_ports [get_cells -hierarchical *]
```

```
# Set Wire Load
set_wire_load_model -name 5K_hvratio_1_4

# Set Clock
create_clock -name "CK" -period $period {"CLK"}

# Do not optimize clock network
set_dont_touch_network CK

# Set clock uncertainty (for jittering)
set_clock_uncertainty $uncertainty [get_clocks CK]

# Set Constraints
#set_max_dynamic_power $power uW

# Compile Design
#compile
#compile -map_effort high
compile_ultra -timing_high_effort_script

# Reports
report_area > reports/dlx_area_${label}_${period}.txt
report_timing > reports/dlx_timing_${label}_${period}.txt
report_power > reports/dlx_power_${label}_${period}.txt

# Save Design
write -hierarchy -format ddc -output designs/dlx_${label}_${period}.ddc

# Generate VHDL and Verilog
write -hierarchy -format vhdl -output designs/dlx_${label}_${period}.vhd
write -hierarchy -format verilog -output designs/dlx_${label}_${period}.v

# Generate SDC script
write_sdc constraints/dlx_${label}_${period}.sdc

quit
```

A.3 Physical Design

A.3.1 runphy.sh

```
#!/bin/bash

#-----------------------------------------------------------------------------------------------
# Description:  This bash script is designed to manage the physical design process for the
#               DLX project.
#
# Author:       Riccardo Cuccu
# Date:         2023/09/19
#-----------------------------------------------------------------------------------------------

# Initialize environment
source /eda/scripts/init_cadence_2020-21 &> /dev/null #setinnovus

# Cleanup and create new working directories
rm -r -f constraints designs
mkdir -p constraints designs

# Copy Verilog files
cp -r ../syn/designs/*.v ./designs
cp -r ../syn/constraints/*.sdc ./constraints

#innovus

# After launching Innovus:
# 1. Execute the "source dlx.globals" command to import the design.
# 2. Execute the "source phy.tcl" command to generate the physical design.
# 3. Execute the "source analyses.tcl" command to generate the reports.

# Cleanup
rm -f *.log *.logv *.cmd
```

A.3.2 dlx.globals

```
###############################################################
#   Generated by:      Cadence Encounter 13.13-s017_1
#   OS:                Linux x86_64(Host ID centos-microb)
#   Generated on:      Thu May 29 16:17:28 2014
#   Design:            DLX
#   Command:           save_global dlx.globals
###############################################################
#
# Version 1.1
#


set defHierChar {/}
set delaycal_input_transition_delay {0.1ps}
set fpIsMaxIoHeight 0
set init_gnd_net {gnd}


set init_mmmc_file {default.view}
set init_oa_search_lib {}
set init_pwr_net {vdd}
set init_verilog {designs/dlx_clock_bound_0.5.v}
set lsgOCPGainMult 1.000000

set LEF_DIR /eda/dk/nangate45/lef
set LEF_list [list ${LEF_DIR}/NangateOpenCellLibrary.lef]

set init_lef_file "${LEF_list}"

set LIB_DIR /eda/dk/nangate45/liberty
set MyTimingLibNom ${LIB_DIR}/NangateOpenCellLibrary_typical_ecsm_nowlm.lib
set MyTimingLibSlow ${LIB_DIR}/NangateOpenCellLibrary_slow_ecsm.lib
set MyTimingLibFast ${LIB_DIR}/NangateOpenCellLibrary_fast_ecsm.lib

set MycapTable $LEF_DIR/captables/NCSU_FreePDK_45nm.capTbl
```

A.3.3 phy.tcl

```
init_design
getIoFlowFlag
setIoFlowFlag 0
floorPlan -coreMarginsBy die -site FreePDK45_38x28_10R_NP_162NW_34O -r 1.0 0.6 5 5 5 5
uiSetTool select
getIoFlowFlag
fit
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
set sprCreateIeRingLayers {}
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
set sprCreateIeRingLayers {}
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeThreshold 1.0
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeThreshold 1.0
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
set sprCreateIeRingLayers {}
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeThreshold 1.0
setAddRingMode -ring_target default -extend_over_row 0 -ignore_rows 0 -avoid_short 0
     -skip_crossing_trunks none -stacked_via_top_layer metal10 -stacked_via_bottom_layer metal1
     -via_using_exact_crossover_size 1 -orthogonal_only true -skip_via_on_pin {  standardcell }
     -skip_via_on_wire_shape {  noshape }
addRing -nets {vdd gnd} -type core_rings -follow core -layer {top metal9 bottom metal9 left
     metal10 right metal10} -width {top 0.8 bottom 0.8 left 0.8 right 0.8} -spacing {top 0.8 bottom
      0.8 left 0.8 right 0.8} -offset {top 1.8 bottom 1.8 left 1.8 right 1.8} -center 1 -threshold
     0 -jog_distance 0 -snap_wire_center_to_grid None
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
set sprCreateIeRingLayers {}
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
set sprCreateIeRingLayers {}
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeThreshold 1.0
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeThreshold 1.0
set sprCreateIeRingOffset 1.0
set sprCreateIeRingThreshold 1.0
set sprCreateIeRingJogDistance 1.0
set sprCreateIeRingLayers {}
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeThreshold 1.0
setAddStripeMode -ignore_block_check false -break_at none -route_over_rows_only false
     -rows_without_stripes_only false -extend_to_closest_target none -stop_at_last_wire_for_area
     false -partial_set_thru_domain false -ignore_nondefault_domains false
     -trim_antenna_back_to_shape none -spacing_type edge_to_edge -spacing_from_block 0
     -stripe_min_length stripe_width -stacked_via_top_layer metal10 -stacked_via_bottom_layer
     metal1 -via_using_exact_crossover_size false -split_vias false -orthogonal_only true
     -allow_jog { padcore_ring  block_ring } -skip_via_on_pin {  standardcell }
     -skip_via_on_wire_shape {  noshape   }
addStripe -nets {vdd gnd} -layer metal10 -direction vertical -width 0.8 -spacing 0.8
     -set_to_set_distance 20 -start_from left -start_offset 15 -switch_layer_over_obs false
     -max_same_layer_jog_length 2 -padcore_ring_top_layer_limit metal10
     -padcore_ring_bottom_layer_limit metal1 -block_ring_top_layer_limit metal10
     -block_ring_bottom_layer_limit metal1 -use_wire_group 0 -snap_wire_center_to_grid None
setSrouteMode -viaConnectToShape { noshape }
```

```
sroute -connect { blockPin padPin padRing corePin floatingStripe } -layerChangeRange { metal1(1)
    metal10(10) } -blockPinTarget { nearestTarget } -padPinPortConnect { allPort oneGeom }
    -padPinTarget { nearestTarget } -corePinTarget { firstAfterRowEnd } -floatingStripeTarget {
    blockring padring ring stripe ringpin blockpin followpin } -allowJogging 1
    -crossoverViaLayerRange { metal1(1) metal10(10) } -nets { vdd gnd } -allowLayerChange 1
    -blockPin useLef -targetViaLayerRange { metal1(1) metal10(10) }
setPlaceMode -prerouteAsObs {1 2 3 4 5 6 7 8}
setPlaceMode -fp false
place_design
fit
fit
fit
getPinAssignMode -pinEditInBatch -quiet
setPinAssignMode -pinEditInBatch true
editPin -fixOverlap 1 -unit MICRON -spreadDirection clockwise -side Left -layer 1 -spreadType
    center -spacing 0.14 -pin CLK
setPinAssignMode -pinEditInBatch false
getPinAssignMode -pinEditInBatch -quiet
setPinAssignMode -pinEditInBatch true
editPin -fixOverlap 1 -unit MICRON -spreadDirection clockwise -side Left -layer 1 -spreadType
    center -spacing 0.14 -pin RST
setPinAssignMode -pinEditInBatch false
getPinAssignMode -pinEditInBatch -quiet
setPinAssignMode -pinEditInBatch true
editPin -pinWidth 0.07 -pinDepth 0.07 -fixOverlap 1 -unit MICRON -spreadDirection clockwise -side
    Left -layer 1 -spreadType center -spacing 0.14 -pin RST
setPinAssignMode -pinEditInBatch false
setOptMode -fixCap true -fixTran true -fixFanoutLoad false
optDesign -postCTS
optDesign -postCTS -hold
getFillerMode -quiet
addFiller -cell FILLCELL_X1 FILLCELL_X2 FILLCELL_X4 FILLCELL_X8 FILLCELL_X16 FILLCELL_X32 -prefix
    FILLER
setNanoRouteMode -quiet -timingEngine {}
setNanoRouteMode -quiet -routeWithSiPostRouteFix 0
setNanoRouteMode -quiet -drouteStartIteration default
setNanoRouteMode -quiet -routeTopRoutingLayer default
setNanoRouteMode -quiet -routeBottomRoutingLayer default
setNanoRouteMode -quiet -drouteEndIteration default
setNanoRouteMode -quiet -routeWithTimingDriven false
setNanoRouteMode -quiet -routeWithSiDriven false
routeDesign -globalDetail
setAnalysisMode -analysisType onChipVariation
setOptMode -fixCap true -fixTran true -fixFanoutLoad false
optDesign -postRoute
optDesign -postRoute -hold
saveDesign dlx
```

A.3.4 analyses.tcl

```
set_analysis_view -setup {default} -hold {default}
reset_parasitics
extractRC
rcOut -setload dlx.setload -rc_corner standard
rcOut -setres dlx.setres -rc_corner standard
rcOut -spf dlxspf -rc_corner standard
rcOut -spef dlx.spef -rc_corner standard
redirect -quiet {set honorDomain [getAnalysisMode -honorClockDomains]} > /dev/null
timeDesign -postRoute -pathReports -drvReports -slackReports -numPaths 50 -prefix DLX_postRoute
    -outDir timingReports
redirect -quiet {set honorDomain [getAnalysisMode -honorClockDomains]} > /dev/null
timeDesign -postRoute -hold -pathReports -slackReports -numPaths 50 -prefix DLX_postRoute -outDir
    timingReports
getAnalysisMode -checkType
getAnalysisMode -checkType
get_time_unit
report_timing -machine_readable -max_paths 10000 -max_slack 0.75 -path_exceptions all -late >
    top.mtarpt
load_timing_debug_report -name default_report top.mtarpt
verifyConnectivity -type all -error 1000 -warning 50
reportGateCount -level 5 -limit 100 -outfile dlx.gateCount
```

```
saveNetlist dlx.v
all_hold_analysis_views
all_setup_analysis_views
write_sdf  -ideal_clock_network dlx.sdf
```

APPENDIX B

HDL

B.1 Globals

B.1.1 000-functions.vhd

```vhdl
-----------------------------------------------------------------------------------------------
-- Description: This VHDL package contains utility functions used throughout the DLX architecture.
--              The package includes three primary functions: 'divide', 'log2', and 'pow2'.
--              1. The 'divide' function performs integer division and rounds up if there is a
--                 remainder.
--              2. The 'log2' function calculates the base-2 logarithm of an integer, also
--                 rounding up.
--              3. The 'pow2' function calculates 2 raised to the power of a given integer.
--
-- Author:      Riccardo Cuccu
-- Date:        2023/09/09
-----------------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

package functions is

        -- Performs division and rounds up if there's a remainder
        function divide (n:integer; m:integer) return integer;

        -- Calculates the base-2 logarithm of an integer, rounding up
        function log2 (n:integer) return integer;

        -- Calculates 2 to the power of n
        function pow2 (n:integer) return integer;

end functions;

package body functions is

        function divide (n:integer; m:integer) return integer is
        begin

                if (n mod m) = 0 then
                        return n/m;
                else
                        return (n/m) + 1;
                end if;

        end divide;

        function log2 (n:integer) return integer is
        begin
```

```
                if n <=2 then
                        return 1;
                else
                        return 1 + log2(divide(n,2));
                end if;
        end log2;

        function pow2 (n:integer) return integer is
                variable result: integer := 1;
        begin
                if n = 0 then
                        return 1;
                else
                        for i in 1 to n loop
                                result := result * 2;
                        end loop;
                        return result;
                end if;
        end pow2;

end functions;
```

B.1.2 000-globals.vhd

```
-------------------------------------------------------------------------------------------------
-- Description: This package sets up constants and types for universal use across the DLX project.
--              It contains defaults for sizes, ALU operations, instruction types, and so on,
--              functioning as a centralized control point for various parameters to simplify
--              system configuration and maintenance.
--
-- Author:      Riccardo Cuccu
-- Date:        2023/09/19
-------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.functions.all;

package constants is

        constant SIZE_GLOBAL                    : integer := 32;

        -- Control unit input sizes
        constant OPC_SIZE_GLOBAL                : integer := 6;
        constant FUNC_SIZE_GLOBAL               : integer := 11;
        constant RS_SIZE_GLOBAL                 : integer := 5;

        -- Control unit registers sizes
        constant IR_SIZE_GLOBAL                 : integer := SIZE_GLOBAL;
        constant PC_SIZE_GLOBAL                 : integer := SIZE_GLOBAL;
        constant MMEM_SIZE_GLOBAL               : integer := 64;
        --constant RMEM_SIZE_GLOBAL             : integer := 64;
        constant CW_SIZE_GLOBAL                 : integer := 18;

        -- Instruction memory
        constant IRAM_SIZE_GLOBAL               : integer := 2**8;

        -- Register file
--      constant RF_ADDRESSES_GLOBAL            : integer := 5;
        constant RF_SIZE_GLOBAL                 : integer := IR_SIZE_GLOBAL;

        -- ALU registers sizes
        constant ALU_OP_SIZE_GLOBAL             : integer := SIZE_GLOBAL;
        constant ALU_BLOCK_SIZE_GLOBAL          : integer := 8;
        constant ALU_BITBLOCK_SIZE_GLOBAL       : integer := ALU_OP_SIZE_GLOBAL/
             ALU_BLOCK_SIZE_GLOBAL;
        constant ALU_OP_MUX_SIZE_GLOBAL         : integer := ALU_BITBLOCK_SIZE_GLOBAL;
        constant ALU_OP_RCA_SIZE_GLOBAL         : integer := ALU_BITBLOCK_SIZE_GLOBAL;
        constant ALU_EXPP4_GLOBAL               : integer := log2(ALU_OP_SIZE_GLOBAL);
        constant ALU_MUL_RADIX_GLOBAL           : integer := 8;
        constant ALU_MUL_ENCODER_GLOBAL         : integer := ALU_OP_SIZE_GLOBAL/2;
        -- DRAM
        constant DRAM_SIZE_GLOBAL               : integer := 2**8;
        constant DRAM_WORD_SIZE_GLOBAL          : integer := SIZE_GLOBAL;

        -- ALU Operations
        type aluOp is (         -- FUNC labels
                                OP_SLL , OP_SRL , OP_SRA , OP_ADD , OP_ADDU , OP_SUB , OP_SUBU , OP_AND ,
                                     OP_OR , OP_XOR , OP_SEQ , OP_SNE , OP_SLT , OP_SGT , OP_SLE , OP_SGE ,
                                      OP_MOVI2S , OP_MOVS2I , OP_MOVF , OP_MOVD , OP_MOVFP2I ,
                                     OP_MOVI2FP , OP_MOVI2T , OP_MOVT2I , OP_SLTU , OP_SGTU , OP_SLEU ,
                                     OP_SGEU ,
                                OP_ADDF , OP_SUBF , OP_MULTF , OP_DIVF , OP_ADDD , OP_SUBD , OP_MULTD ,
                                     OP_DIVD , OP_CVTF2D , OP_CVTF2I , OP_CVTD2F , OP_CVTD2I , OP_CVTI2F
                                     , OP_CVTI2D , OP_MULT , OP_DIV , OP_EQF , OP_NEF , OP_LTF , OP_GTF ,
                                     OP_LEF , OP_GEF , OP_MULTU , OP_DIVU , OP_EQD , OP_NED , OP_LTD ,
                                     OP_GTD , OP_LED , OP_GED ,

                                -- OPCODE labels
                                OP_BEQZ , OP_BNEZ , OP_BFPT , OP_BFPF , OP_ADDI , OP_ADDUI , OP_SUBI ,
                                     OP_SUBUI , OP_ANDI , OP_ORI , OP_XORI , OP_LHI , OP_RFE , OP_TRAP ,
                                     OP_JR , OP_JALR , OP_SLLI , OP_SRLI , OP_SRAI , OP_SEQI , OP_SNEI ,
                                     OP_SLTI , OP_SGTI , OP_SLEI , OP_SGEI , OP_LB , OP_LH , OP_LW ,
                                     OP_LBU , OP_LHU , OP_LF , OP_LD , OP_SB , OP_SH , OP_SW , OP_SF ,
                                     OP_SD , OP_ITLB , OP_SLTUI , OP_SGTUI , OP_SLEUI , OP_SGEUI ,
                                OP_J , OP_JAL ,
                                OP_NOP

                        );

        -- FUNC labels
        type ALU_label is (     -- FUNC labels
                                L_RTYPE_SLL , L_RTYPE_SRL , L_RTYPE_SRA , L_RTYPE_ADD , L_RTYPE_ADDU ,
                                     L_RTYPE_SUB , L_RTYPE_SUBU , L_RTYPE_AND , L_RTYPE_OR ,
                                     L_RTYPE_XOR , L_RTYPE_SEQ , L_RTYPE_SNE , L_RTYPE_SLT ,
                                     L_RTYPE_SGT , L_RTYPE_SLE , L_RTYPE_SGE , L_RTYPE_MOVI2S ,
                                     L_RTYPE_MOVS2I , L_RTYPE_MOVF , L_RTYPE_MOVD , L_RTYPE_MOVFP2I ,
                                     L_RTYPE_MOVI2FP , L_RTYPE_MOVI2T , L_RTYPE_MOVT2I , L_RTYPE_SLTU ,
                                      L_RTYPE_SGTU , L_RTYPE_SLEU , L_RTYPE_SGEU ,
                                L_RTYPE_ADDF , L_RTYPE_SUBF , L_RTYPE_MULTF , L_RTYPE_DIVF ,
                                     L_RTYPE_ADDD , L_RTYPE_SUBD , L_RTYPE_MULTD , L_RTYPE_DIVD ,
                                     L_RTYPE_CVTF2D , L_RTYPE_CVTF2I , L_RTYPE_CVTD2F , L_RTYPE_CVTD2I
                                     , L_RTYPE_CVTI2F , L_RTYPE_CVTI2D , L_RTYPE_MULT , L_RTYPE_DIV ,
                                     L_RTYPE_EQF , L_RTYPE_NEF , L_RTYPE_LTF , L_RTYPE_GTF ,
                                     L_RTYPE_LEF , L_RTYPE_GEF , L_RTYPE_MULTU , L_RTYPE_DIVU ,
                                     L_RTYPE_EQD , L_RTYPE_NED , L_RTYPE_LTD , L_RTYPE_GTD ,
                                     L_RTYPE_LED , L_RTYPE_GED ,
                                L_RTYPE_NOP ,

                                -- OPCODE labels
                                L_RTYPE , L_ITYPE , L_JTYPE ,
                                L_ITYPE_BEQZ , L_ITYPE_BNEZ , L_ITYPE_BFPT , L_ITYPE_BFPF ,
                                     L_ITYPE_ADDI , L_ITYPE_ADDUI , L_ITYPE_SUBI , L_ITYPE_SUBUI ,
                                     L_ITYPE_ANDI , L_ITYPE_ORI , L_ITYPE_XORI , L_ITYPE_LHI ,
                                     L_ITYPE_RFE , L_ITYPE_TRAP , L_ITYPE_JR , L_ITYPE_JALR ,
                                     L_ITYPE_SLLI , L_ITYPE_NOP , L_ITYPE_SRLI , L_ITYPE_SRAI ,
                                     L_ITYPE_SEQI , L_ITYPE_SNEI , L_ITYPE_SLTI , L_ITYPE_SGTI ,
                                     L_ITYPE_SLEI , L_ITYPE_SGEI , L_ITYPE_LB , L_ITYPE_LH , L_ITYPE_LW
                                     , L_ITYPE_LBU , L_ITYPE_LHU , L_ITYPE_LF , L_ITYPE_LD , L_ITYPE_SB
                                     , L_ITYPE_SH , L_ITYPE_SW , L_ITYPE_SF , L_ITYPE_SD , L_ITYPE_ITLB
                                     , L_ITYPE_SLTUI , L_ITYPE_SGTUI , L_ITYPE_SLEUI , L_ITYPE_SGEUI ,
                                L_JTYPE_J , L_JTYPE_JAL ,
                                L_NOP

                        );


-- R-Type register-registe instruction -> FUNC field
        constant RTYPE_SLL      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "04";
        constant RTYPE_SRL      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "06";
        constant RTYPE_SRA      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "07";
        constant RTYPE_ADD      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "20";
        constant RTYPE_ADDU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "21";
        constant RTYPE_SUB      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "22";
        constant RTYPE_SUBU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "23";
        constant RTYPE_AND      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "24";
        constant RTYPE_OR       : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "25";
        constant RTYPE_XOR      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "26";
        constant RTYPE_SEQ      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "28";
        constant RTYPE_SNE      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
            "29";
        constant RTYPE_SLT      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"2A
            ";
        constant RTYPE_SGT      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"2B
            ";
        constant RTYPE_SLE      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"2C
            ";
        constant RTYPE_SGE      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"2D
            ";
--      constant RTYPE_MOVI2S   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "30";
--      constant RTYPE_MOVS2I   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "31";
--      constant RTYPE_MOVF     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "32";
--      constant RTYPE_MOVD     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "33";
--      constant RTYPE_MOVFP2I  : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "34";
--      constant RTYPE_MOVI2FP  : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "35";
--      constant RTYPE_MOVI2T   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "36";
--      constant RTYPE_MOVT2I   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "37";
        constant RTYPE_SLTU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"3A
            ";
        constant RTYPE_SGTU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"3B
            ";
        constant RTYPE_SLEU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"3C
            ";
        constant RTYPE_SGEU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"3D
            ";

-- R-Type floating-point instruction -> FUNC field
--      constant RTYPE_ADDF     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "00";
--      constant RTYPE_SUBF     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "01";
--      constant RTYPE_MULTF    : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "02";
--      constant RTYPE_DIVF     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "03";
--      constant RTYPE_ADDD     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "04";
--      constant RTYPE_SUBD     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "05";
--      constant RTYPE_MULTD    : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "06";
--      constant RTYPE_DIVD     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "07";
--      constant RTYPE_CVTF2D   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "08";
--      constant RTYPE_CVTF2I   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "09";
--      constant RTYPE_CVTD2F   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"0A
    ";
```

```
--      constant RTYPE_CVTD2I   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"0B
    ";
--      constant RTYPE_CVTI2F   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"0C
    ";
--      constant RTYPE_CVTI2D   : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"0D
    ";
--      constant RTYPE_MULT     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"0E
    ";
        constant RTYPE_MULT     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"1E
            ";
--      constant RTYPE_DIV      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"0F
    ";
--      constant RTYPE_EQF      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "10";
--      constant RTYPE_NEF      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "11";
--      constant RTYPE_LTF      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "12";
--      constant RTYPE_GTF      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "13";
--      constant RTYPE_LEF      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "14";
--      constant RTYPE_GEF      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "15";
--      constant RTYPE_MULTU    : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "16";
        constant RTYPE_MULTU    : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"1F
            ";
--      constant RTYPE_DIVU     : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "17";
--      constant RTYPE_EQD      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "18";
--      constant RTYPE_NED      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x
    "19";
--      constant RTYPE_LTD      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"1A
    ";
--      constant RTYPE_GTD      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"1B
    ";
--      constant RTYPE_LED      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"1C
    ";
--      constant RTYPE_GED      : std_logic_vector(FUNC_SIZE_GLOBAL - 1 downto 0) := "000" & x"1D
    ";

-- R-Type instruction -> OPCODE field
        constant RTYPE          : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"0";

-- I-Type instruction -> OPCODE field
        constant ITYPE_BEQZ     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"4";
        constant ITYPE_BNEZ     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"5";
--      constant ITYPE_BFPT     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"6";
--      constant ITYPE_BFPF     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"7";
        constant ITYPE_ADDI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"8";
        constant ITYPE_ADDUI    : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"9";
        constant ITYPE_SUBI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"A";
        constant ITYPE_SUBUI    : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"B";
        constant ITYPE_ANDI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"C";
        constant ITYPE_ORI      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"D";
        constant ITYPE_XORI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"E";
        constant ITYPE_LHI      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"F";
--      constant ITYPE_RFE      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"0";
--      constant ITYPE_TRAP     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"1";
        constant ITYPE_JR       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"2";
        constant ITYPE_JALR     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"3";
        constant ITYPE_SLLI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"4";
        constant ITYPE_NOP      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"5";
        constant ITYPE_SRLI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"6";
        constant ITYPE_SRAI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"7";
        constant ITYPE_SEQI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"8";
        constant ITYPE_SNEI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"9";
        constant ITYPE_SLTI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"A";
        constant ITYPE_SGTI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"B";
        constant ITYPE_SLEI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"C";
        constant ITYPE_SGEI     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "01" & x"D";
--      constant ITYPE_LB       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"0";
--      constant ITYPE_LH       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"1";
        constant ITYPE_LW       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"3";
--      constant ITYPE_LBU      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"4";
--      constant ITYPE_LHU      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"5";
--      constant ITYPE_LF       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"6";
--      constant ITYPE_LD       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"7";
--      constant ITYPE_SB       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"8";
--      constant ITYPE_SH       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"9";
        constant ITYPE_SW       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"B";
--      constant ITYPE_SF       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"E";
--      constant ITYPE_SD       : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "10" & x"F";
--      constant ITYPE_ITLB     : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "11" & x"8";
        constant ITYPE_SLTUI    : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "11" & x"A";
        constant ITYPE_SGTUI    : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "11" & x"B";
        constant ITYPE_SLEUI    : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "11" & x"C";
        constant ITYPE_SGEUI    : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "11" & x"D";

-- J-Type instruction -> OPCODE field
        constant JTYPE_J        : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"2";
        constant JTYPE_JAL      : std_logic_vector(OPC_SIZE_GLOBAL - 1 downto 0) := "00" & x"3";

end constants;
```