Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master's degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors:	Group 24
Badas Lorenzo	s329861
Santarossa Lisa	s324561
Quellerba Lorenzo	s328326

October 11, 2025

Contents

CHAPTER 1

Introduction

This document presents the design and implementation of a DeLuX (DLX) microprocessor as the final project for the Microelectronic Systems course. The primary objective of this project is to develop a complete DLX processor core from Register-Transfer Level (RTL) description down to physical design, traversing the entire digital design flow.

The DLX architecture, originally introduced by Hennessy and Patterson, serves as the foundation for this implementation. It represents a classic Reduced Instruction Set Computer (RISC) architecture characterized by a load-store design with 32 general-purpose registers and a minimal instruction set.

1.1 Project Overview

The project is structured around two main implementation tiers:

- DLX-Basic
- DLX-Pro

1.1.1 DLX-Basic

The **DLX-Basic** version constitutes the mandatory core requirements that every implementation must fulfill. This includes:

- A fully pipelined architecture organized around the classic five-stage pipeline (IF, ID, EX, MEM, WB)
- Implementation of a comprehensive instruction subset including arithmetic, logical, control flow, and memory operations
- Complete datapath description in VHDL at RTL level
- Synthesis of both control unit and datapath
- Physical design implementation including placement and routing

1.1.2 DLX-Pro

The **DLX-Pro** version represents an enhanced implementation that extends beyond the basic requirements. Teams can selectively implement more advanced features from an extensive list of optional specifications, including:

- Extended instruction set with additional operations
- Microarchitecture optimizations for critical path reduction
- Advanced control hazard prevention techniques
- Power optimization through architectural improvements
- Cache memory implementation
- Advanced synthesis and physical design techniques

1.2 Implementation Methodology

The implementation utilizes VHDL (VHSIC Hardware Description Language), an industry-standard Hardware Description Language (HDL) for Electronic Design Automation (EDA). VHDL enables the description of digital systems at various abstraction levels, from structural to behavioral, and supports simulation, synthesis, and verification of complex digital circuits.
The design process follows a systematic approach:

1. RTL description of the datapath and control unit components in VHDL
2. Integration of the complete processor architecture
3. Functional verification through simulation using Synopsys' ModelSim
4. Logic synthesis and optimization using Synopsys' Design Compiler
5. Physical design, including placement and routing, using Cadence's Innovus

This report details the complete design process for our DLX-pro implementation, from architectural specification and VHDL implementation to synthesis and physical design, providing a complete overview of the DLX microprocessor development.

CHAPTER 2

DLX Architecture

The DLX is a 32-bit RISC Instruction Set Architecture (ISA) born as a simplified version of MIPS for educational purposes.

2.1 Instruction Set

All instructions in the DLX architecture are 32-bit wide and can be grouped into three different types. The first 6 bits are always dedicated to the `OPCODE` field, which determines the type and behavior of the instruction. The remaining bits are divided differently depending on the instruction format:

- **R-type**.
- **I-type**.
- **J-type**.

R-type Register-to-register operations. These instructions perform Arithmetic-Logic Unit (ALU) operations using two source registers (RS1 and RS2) and store the result in a destination register (RD). The `FUNC` field refines the operation specified by the `OPCODE`.

The encoding is shown in Table 2.1.

6-bit	5-bit	5-bit	5-bit	11-bit
OPCODE	**RS1**	**RS2**	**RD**	**FUNC**

Table 2.1: R-type instruction scheme

I–type Instructions involving a source register (RS) and an immediate value to perform an ALU operation. With a few exceptions, the result is stored in a destination register specified by the RD field. Loads and stores, branches and arithmetical operations are part of this category.

The encoding is shown in Table 2.2.

6-bit	5-bit	5-bit	16-bit
OPCODE	**RS**	**RD**	**IMMEDIATE**

Table 2.2: I-type instruction scheme

J-type Jump instructions used to alter the control flow by jumping to a relative offset specified in the `IMMEDIATE` field.

The encoding is shown in Table 2.3.

6-bit	26-bit
OPCODE	**IMMEDIATE**

Table 2.3: J-type instruction scheme

2.2 DLX-Pro Implementation

We propose here a DLX-Pro implementation which, in addition to the DLX-Basic requirements, features:

1. A significant increase in the number of supported instructions.
2. Full hazard handling managing all control, structural, and data hazards.
3. Forwarding paths to resolve data dependencies with reduced or eliminated pipeline stalls.
4. Early branch and jump resolution logic reducing branch penalty to one clock cycle.
5. Python-based DLX emulator developed for functional verification.
6. Fully automated design flow from VHDL generation through functional simulation, synthesis, and physical implementation.

2.2.1 Overview

Figure 2.1 shows the block diagram of the implemented architecture, which includes:

- **Control Unit**: Hard-wired unit which generates a control word for each implemented instruction. These control words determine the datapath behavior.
- **Datapath**: Pipelined unit that manages data processing.
- **Hazard Unit**: Unit that handles control hazards caused by branch and jump instructions and some data hazards, by stalling some stages of the datapath.
- **Forwarding Unit**: Unit which manages extra paths to quickly resolve Read-After-Write (RAW) data hazards.
- **Instruction Memory**: External ROM that stores all processor instructions.
- **Data Memory**: External RAM that stores all the data, with a byte-level granularity.

The DLX pipeline consists of the classical five execution stages, namely:

- Instruction Fetch
- Instruction Decode
- Execution
- Memory Access
- Write-Back

The five-stage pipeline structure is clearly illustrated in Figure 2.2, which depicts the main functional blocks within the DLX core.

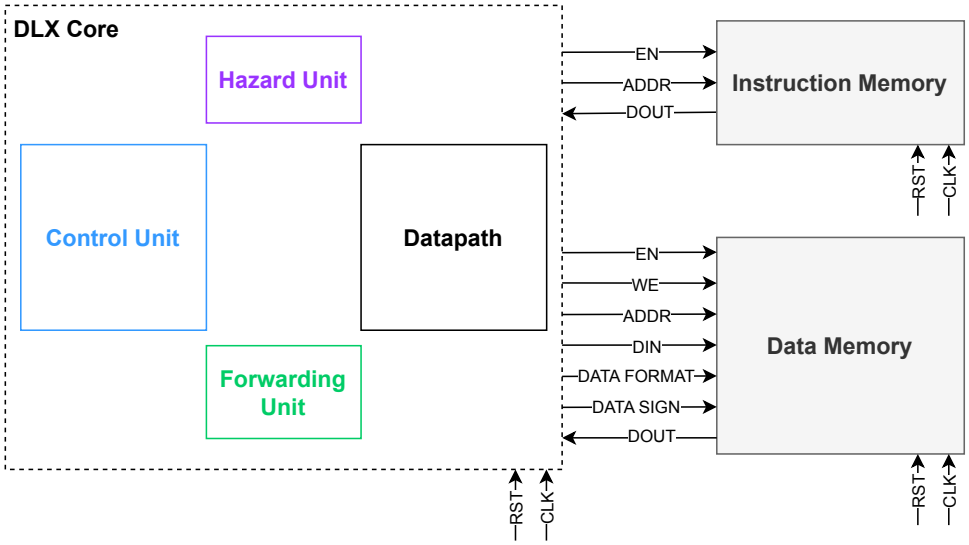

Figure 2.1: DLX Block Diagram.

2.2.2 Supported Instructions

This implementation supports an extended instruction set that fulfills all DLX-Basic requirements while incorporating additional instructions for Pro-level functionality. The supported instruction set is categorized as follows:

- **R-type instructions**: `add`, `sub`, `and`, `or`, `xor`, `sll`, `srl`, `sra`, `seq`, `sne`, `slt`, `sgt`, `sle`, `sge`, `addu`, `subu`, `sltu`, `sgtu`, `sleu`, `sgeu`, `nop`
- **Jump instructions**: `j`, `jal`, `jr`, `jalr`
- **Branch instructions**: `beqz`, `bnez`
- **I-type arithmetic/logic**: `addi`, `subi`, `andi`, `ori`, `xori`, `slli`, `srli`, `srai`, `seqi`, `snei`, `slti`, `sgti`, `slei`, `sgei`, `addui`, `subui`, `sltui`, `sgtui`, `sleui`, `sgeui`
- **Load instructions**: `lb`, `lbu`, `lh`, `lhu`, `lw`
- **Store instructions**: `sb`, `sh`, `sw`

Table 2.4 indicates the mandatory DLX-Basic instruction subset, while Table 2.5 lists the additional instructions implemented to achieve DLX-Pro functionality.

Mandatory Instruction					
j	jal	beqz	bnez	addi	subi
andi	ori	xori	slli	nop	srli
snei	slei	sgei	lw	sw	sll
srl	add	sub	and	or	xor
sne	sle	sge			

Table 2.4: List of mandatory instructions.

Additional Instruction					
jr	jalr	lb	lh	lbu	lhu
sb	sh	sra	seq	slt	sgt
sltu	sgtu	sleu	sgeu	srai	seqi
slti	sgti	sltui	sgtui	sleui	sgeui
addu	subu	addui	subui		

Table 2.5: List of additional instructions.

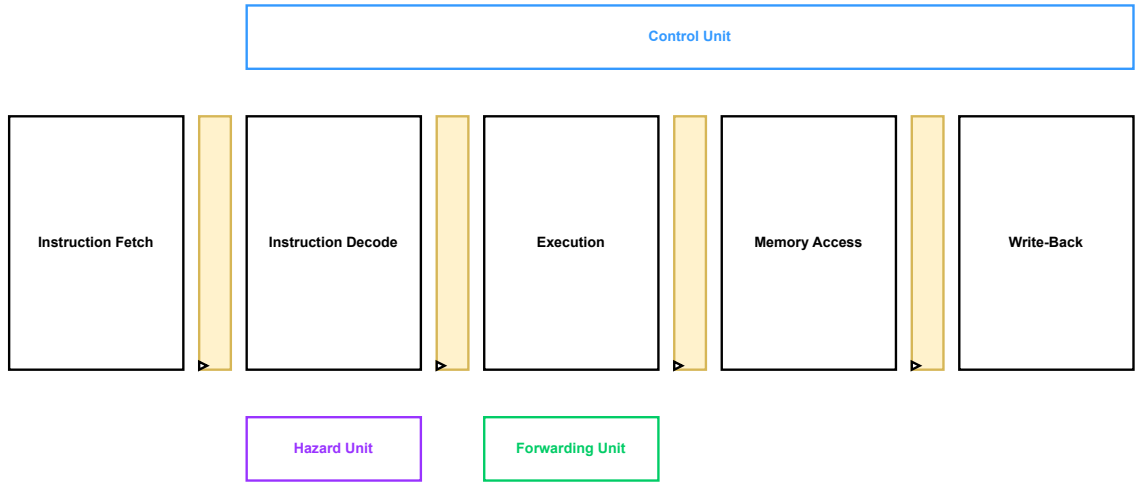

Figure 2.2: Overview of DLX Core Main Blocks

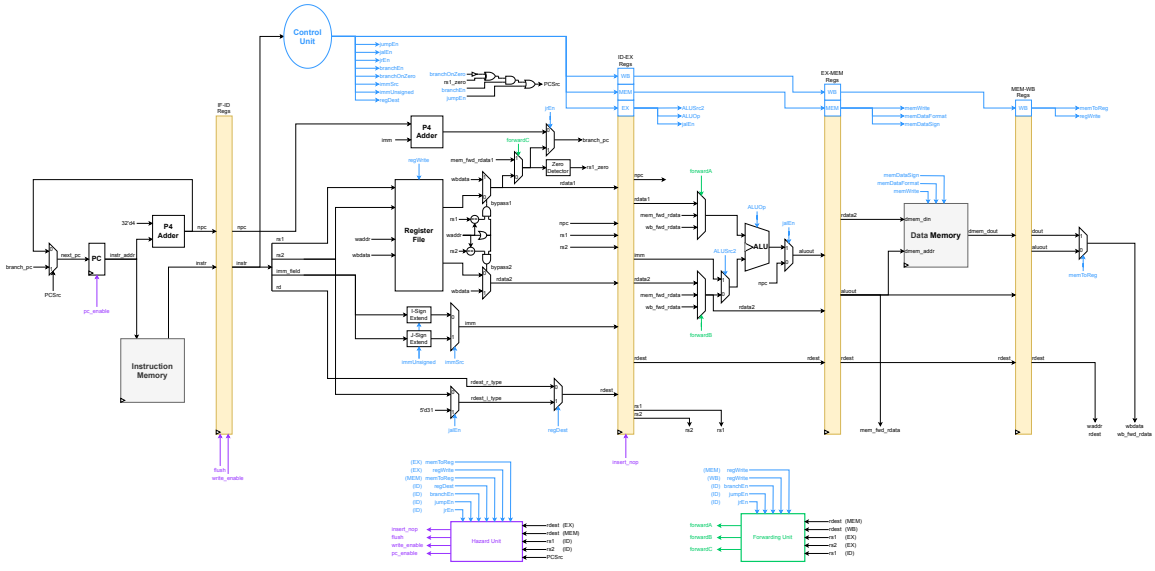

Figure 2.3: DLX Core Full Micro-Architecture

CHAPTER 3

Control Unit

3.1 Overview

The Control Unit, instantiated in the Instruction Decode stage, elaborates the instruction and produces a set of control signals to manage the datapath. This unit is hardwired, with each control word corresponding to a Look-Up Table (LUT) entry that supplies the set of signals for the instruction. These signals are buffered through the pipeline registers alongside the datapath registers.

The schematic of the Control Unit is shown in Figure 3.1.

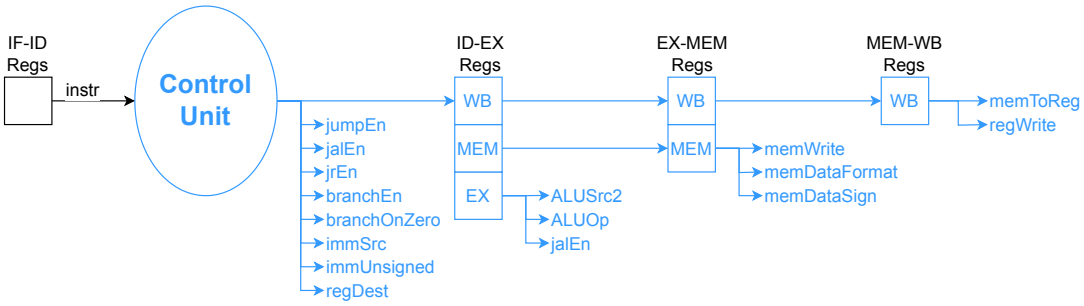

Figure 3.1: Control Unit Diagram

3.2 Control Signals

The control signals are:

- `immSrc`: selects immediate value source (0: I-type, 1: J-type)
- `immUnsigned`: indicates if immediate should be sign extended or zero padded
- `regDest`: selects destination register (0: R-type field, 1: I-type field)
- `regWrite`: enables writing to register file
- `jumpEn`: enables jump operation
- `jrEn`: enables jump register operation
- `branchEn`: enables branch operation

- `branchOnZero`: branch condition (0: branch if not zero, 1: branch if zero)
- `jalEn`: enables jump and link to register 31 operation
- `ALUSrc2`: selects second ALU operand (0: register, 1: immediate)
- `ALUOp`: specifies ALU operation type
- `memWrite`: enables memory write operation
- `memDataFormat[1:0]`: specifies memory access size (byte, halfword, word)
- `memDataSign`: indicates whether memory data should be sign extended
- `memToReg`: selects between ALU result and memory data for register write-back operation

CHAPTER 4

Datapath

4.1 Overview

4.2 Instruction Fetch

The Instruction Fetch (IF) stage is responsible for supplying the pipeline with the next instruction to execute. Figure 4.1 shows the detailed micro-architecture of this stage.

The Program Counter (PC) provides the address of the instruction to the **Instruction Memory**, which returns the corresponding instruction value. In this clock cycle, the PC value is also incremented by 4 bytes, producing the `npc` (next program counter), which points to the following sequential instruction.

A multiplexer selects the next PC. The `PCSrc` signal chooses between the incremented PC (for sequential execution) and a target address (`branch_pc`) for a taken branch or jump instruction, which is calculated in a later pipeline stage.

The selected value is written into the PC on every clock cycle where `pc_enable` is asserted. The `pc_enable` signal is unset during a stall. Both the fetched instruction (`instr`) and the incremented PC (`npc`) are passed to the IF-ID pipeline register.

A `flush` signal can be asserted to clear the IF-ID pipeline register, which is required when a control hazard is detected. Additionally, during a stall, the `write_enable` signal is deasserted to preserve the register's contents.

4.3 Instruction Decode

The Instruction Decode (ID) stage is responsible for interpreting the fetched instruction and preparing all necessary operands and control signals for the next stages. Figure 4.2 shows the detailed micro-architecture of this stage.
The instruction from the IF-ID pipeline register is decoded into:

- Source register addresses (`rs1`, `rs2`)
- Destination register address (`rd`)
- Immediate field (`imm_field`)

The source register addresses are used to access the Register File (RF), which outputs the contents of the two source registers (`rdata1` and `rdata2`) corresponding to `rs1` and `rs2`.

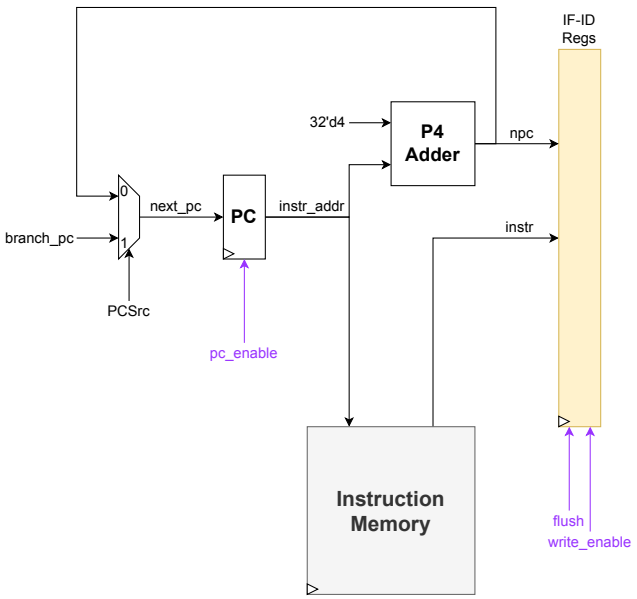

Figure 4.1: Instruction Fetch Stage Architecture

Immediate values are processed through sign-extension units. As previously seen in Chapter 1, there are two different encodings for immediates in I-type and J-type instruction formats. Additionally an immediate field might be either sign-extended or zero padded depending on the operation performed. Because of these two reasons, two control signals are needed to produce the final immediate value `imm` on 32 bits:

- `immUnsigned` indicates the correct extension of the field to 32 bits.
- `immSrc` selects between the I-type and the J-type immediate.

Finally the `imm` value is forwarded to the Execution stage through the ID-EX pipeline register.

In order to avoid stalls due to data hazards, bypass and forwarding paths are implemented into this stage. These paths allow instructions to get source data before it is written back to the RF. This can happen when a previous instruction has already produced the value but has not completed the Write-Back stage yet. More details about this mechanism will be given in Chapter 5.

This data can come from either the Memory Access stage or the Write-Back stage, respectively from the signals `mem_fwd_rdata1` and `wbdata`.

The ID stage is also in charge of handling branches and jumps. In particular:

- It identifies if the instruction is a branch or a jump.
- It calculates the target address for branches and jumps by adding the sign-extended immediate value to the incremented PC.
- For Jump(-and-Link) Register instructions, it selects the value directly from a register, bypassing the PC-relative calculation.
- It computes the conditional branch decision, based on a **Zero Detector** that checks whether `rdata1` equals zero.

- Given the previous steps, it produces the value of the `PCSrc` signal computed as:

$$\text{PCSrc} = (\text{branchEn} \wedge (\text{rs1\_zero} \oplus (\neg\text{branchOnZero}))) \vee \text{jumpEn}$$

Finally, an `insert_nop` signal can be issued by the Hazard Unit to insert a `NOP` instruction into the ID-EX pipeline register to solve data hazards (see Chapter 5).

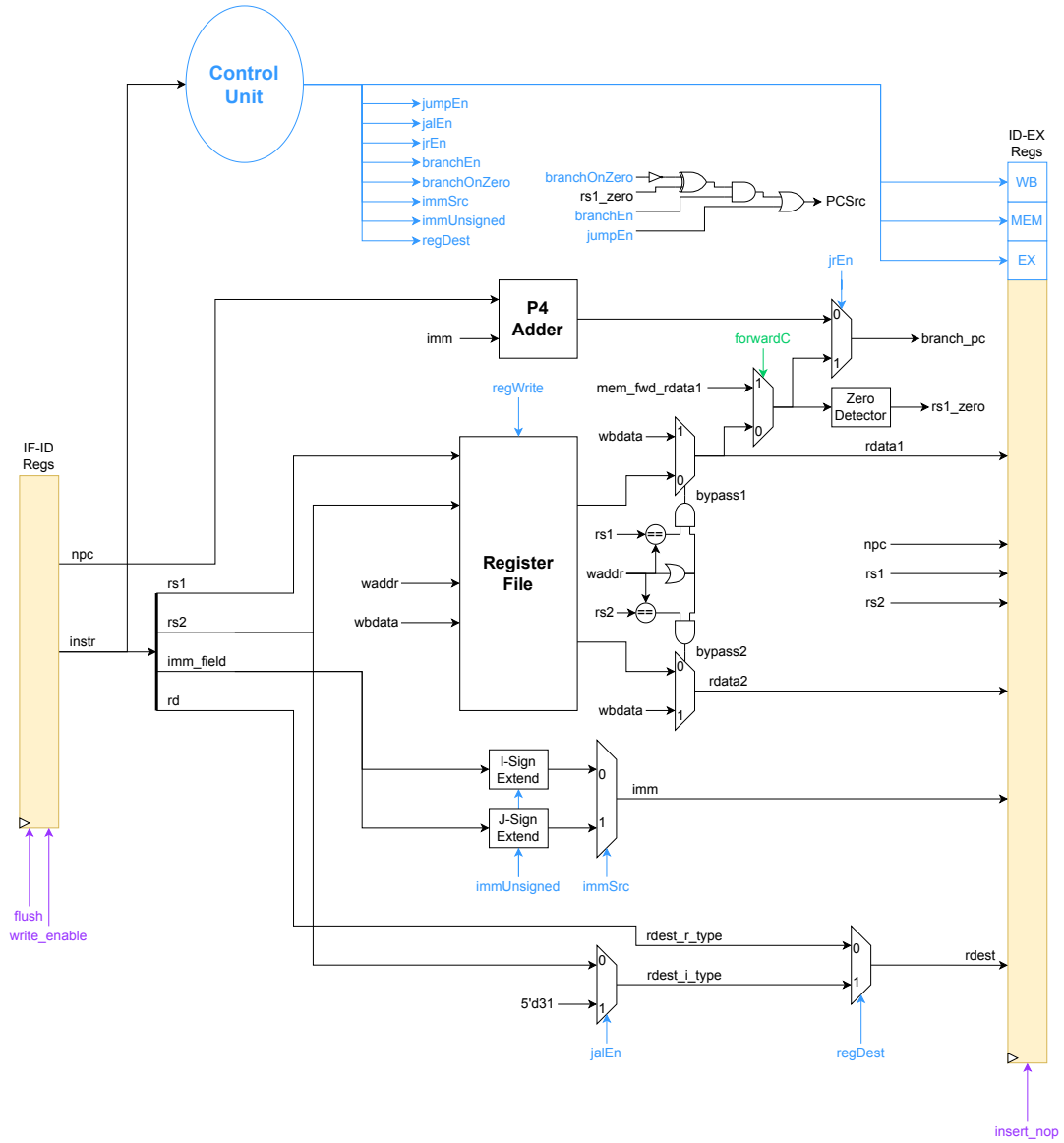

Figure 4.2: Instruction Decode Stage Architecture

4.4 Execution

The Execution (EX) stage is responsible for arithmetic and logical operations. Figure 4.3 shows the datapath of this stage.

The stage primarily consists of the ALU, along with several multiplexers that select the appropriate data sources for the ALU or bypass it entirely.

Each ALU input can receive data from three register sources:

- Directly from the Register File.
- Forwarded from the Memory Access stage.
- Forwarded from the Write-Back stage.

Additionally, the second ALU input is selected between a register value and an immediate value.

After the ALU computes the result, a final multiplexer selects between that result and the incremented PC (`npc`). This selection is necessary for `jal(r)` instructions, which must save the incremented PC to register `R31`.

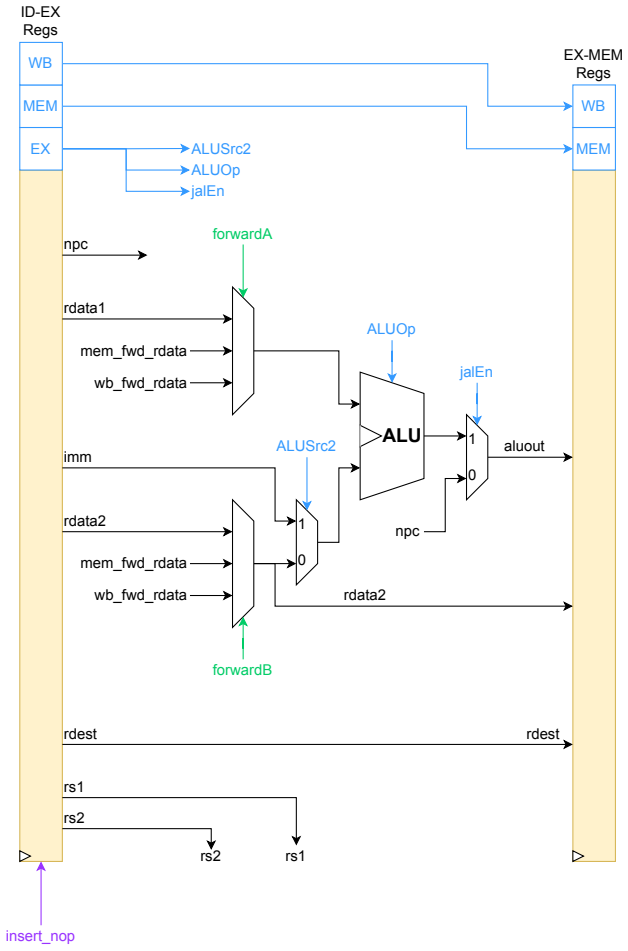

Figure 4.3: Execution Stage Architecture

4.5 Memory Access

The Memory Access (MEM) stage is responsible for handling memory operations with the external **Data Memory**. As shown in Figure 4.4, based on the corresponding control signals, the data memory can either store a data value (`rdata2`) at the computed address (`aluout`) or return the content (`dmem_out`) of that address.

The memory output is passed to the next pipeline register along with the `aluout` and `rdest` values.

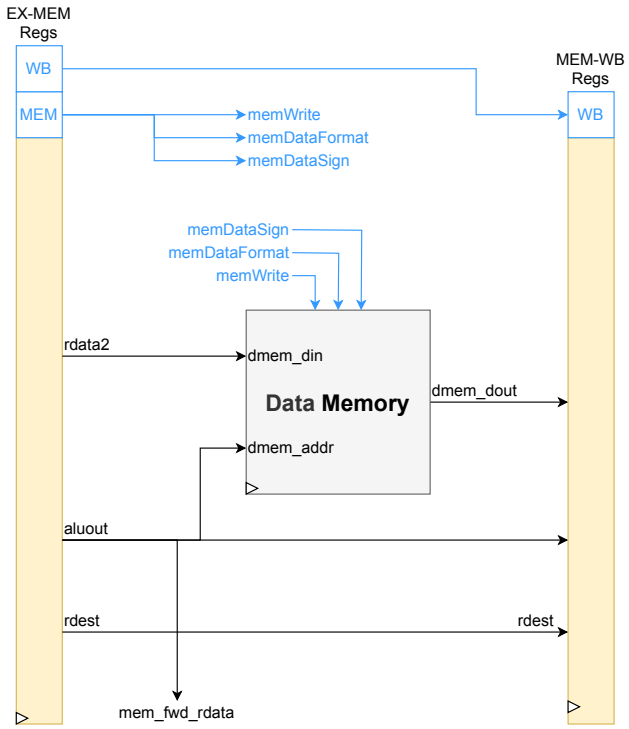

Figure 4.4: Memory Access Stage Architecture

4.6 Write-Back

The Write-Back (WB) stage is responsible for writing a computed value into the RF. This value originates from either the EX stage or the Data Memory. This final pipeline stage is illustrated in Figure 4.5.

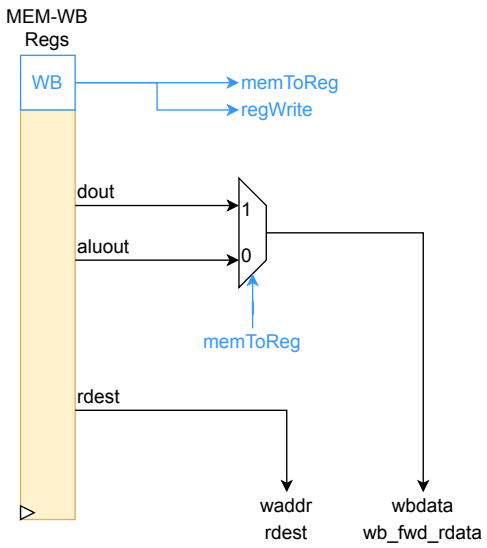

Figure 4.5: Write-Back Stage Architecture

4.7 Components

This DLX processor implementation integrates several computational units inspired by commercial components explored during the course lectures. This section illustrates them.

4.7.1 ALU

The ALU is the component that performs arithmetical and logical computations based on the `ALUOp` control signal. It supports the following set of operations:

- Addition
- Subtraction
- Bitwise operations (`AND`, `OR` and `XOR`)
- Shift operations
- Comparison operations

The `ALUOp` signal, provided by the Control Unit, determines which operation the ALU executes. Figure 4.6 illustrates a block diagram of this component, which takes two 32-bit operands and produces a 32-bit `aluout` result. Its internal structure consists of the following sub-units:

- UltraSPARC T2 Logic Unit.
- UltraSPARC T2 Shifter.
- Intel Pentium 4 Adder.
- Comparator Logic.

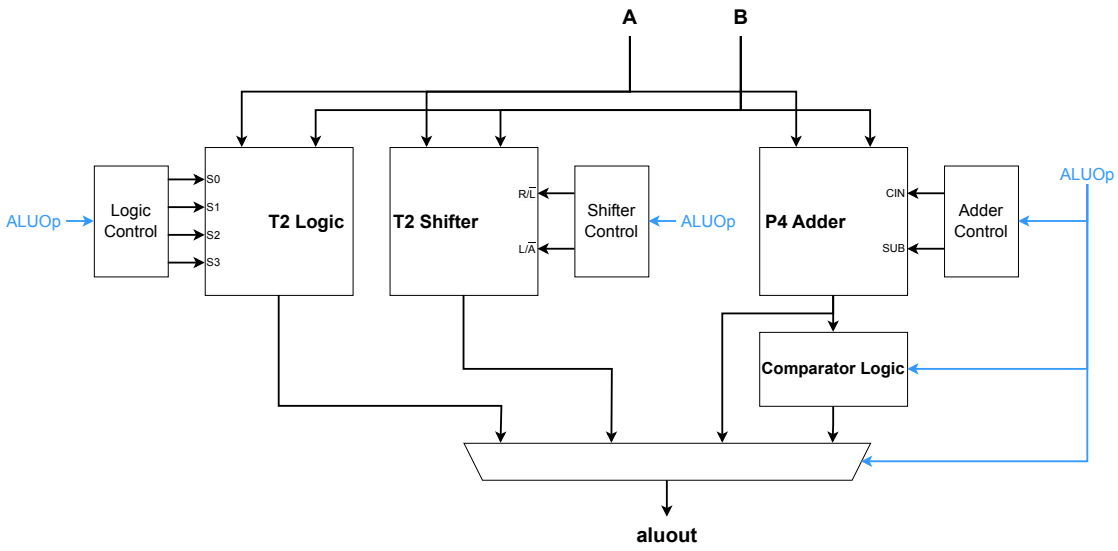

Figure 4.6: ALU Block Diagram

4.7.2 UltraSPARC T2 Logic Unit

A logic unit inspired by the UltraSPARC T2 was implemented to compute all operations required by the DLX architecture: `AND`, `OR`, and `XOR`. The complete architecture of this unit is shown in Figure 4.7.

The module is composed of a single repeated basic building block, illustrated in Figure 4.8. Each block takes as input one bit from each of the two operands, `A` and `B`, and a 4-bit control signal `S`. Four `NAND3` gates and one `NAND4` gate are sufficient to implement the required logic functions. The control signal `S` is set according to the values listed in Table 4.1.

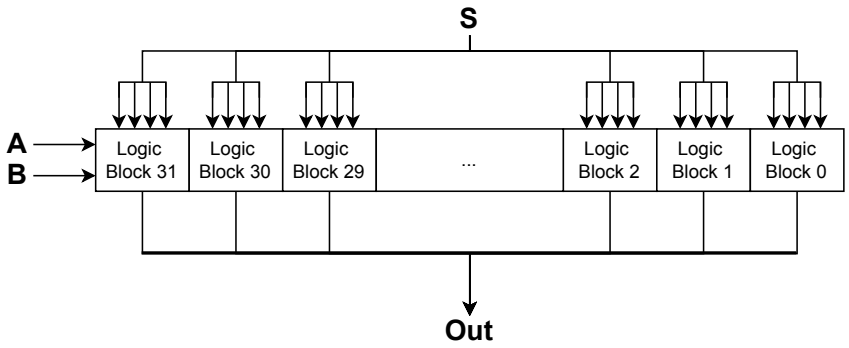

Figure 4.7: UltraSPARC T2 Logic Block Diagram

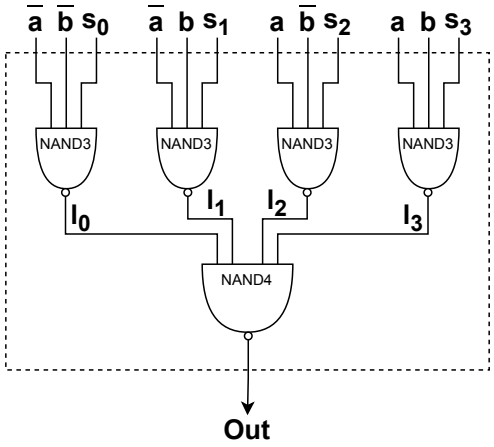

Figure 4.8: UltraSPARC T2 Logic Basic Blocks

4.7.3 UltraSPARC T2 Shifter

This module implements a 32-bit shifter using a two-stage coarse-fine architecture inspired by the UltraSPARC T2 processor. The design supports both logical and arithmetic shifts.

The shifter employs mask-based selection in both stages. The coarse shifter handles byte-aligned shifts using four masks (0, 8, 16, 24), while the fine shifter handles bit-level adjustments using eight

s3	s2	s1	s0	Operation
1	0	0	0	AND
1	1	1	0	OR
0	1	1	0	XOR

Table 4.1: T2 logic unit's Control LUT

masks (0-7). This hierarchical approach minimizes critical path delay and reduces multiplexer complexity compared to a single-stage design.

The block diagram of this module is shown in Figure 4.9

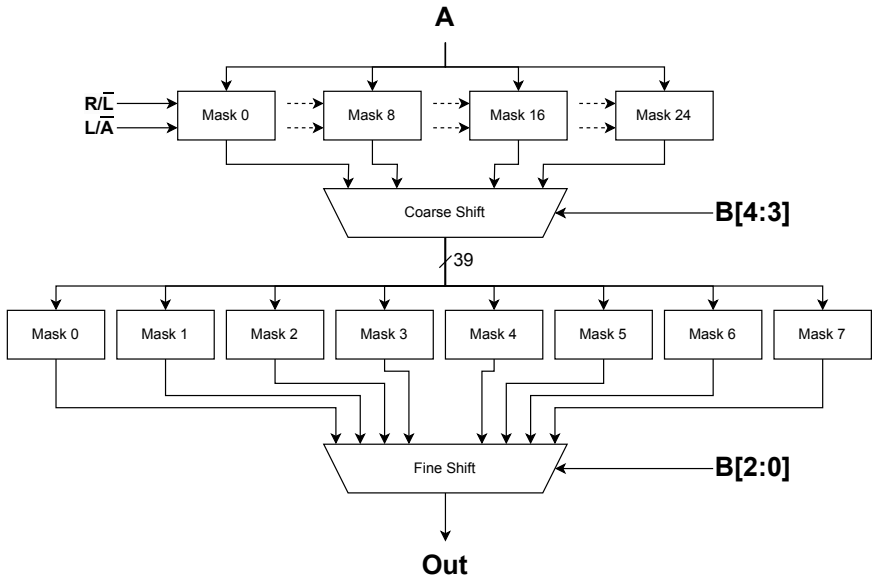

Figure 4.9: UltraSPARC T2 Shifter Block Diagram

4.7.4 Intel Pentium 4 Adder

Additions and subtractions are performed by a component inspired by Intel's Pentium 4 Adder. The block diagram of the implemented adder is shown in Figure 4.10.

The XOR layer is required to selectively invert the second operand, enabling the computation of its two's complement during subtraction operations. This layer is controlled by the `SUB` control signal. At the same time, the adder's carry-in must be set appropriately to complete the two's complement operation of operand `B`.

Following the XOR layer, a fast carry generation network is implemented. Its building blocks are fully detailed in Figure 4.11.

First, the two inputs, `A` and $\underline{\texttt{B}}$, are converted bitwise into *propagate* and *generate* signals. These signals are then combined within a tree of PG-blocks, which computes the carry values with logarithmic delay complexity. The network uses a granularity of 4, generating carries for bits 3, 7, 11, and so on.

This architecture is illustrated in detail in Figure 4.12.

Following the carry generator, the sum generator block calculates the final result using the inputs `A`, $\underline{\texttt{B}}$, and the previously computed carries. The sum is computed using multiple 4-bit Carry-Select

Adders (CSAs) operating in parallel to achieve the full 32-bit width. This architecture allows each CSA to precompute two possible results, one for a '0' carry-in and another for a '1' carry-in, while the actual carry-in is being generated. The final carry-in value, once available, is then used to select the correct result via an output multiplexer.

The structure of a single 4-bit CSA is shown in Figure 4.13, and the complete 32-bit sum generator is illustrated in Figure 4.14.

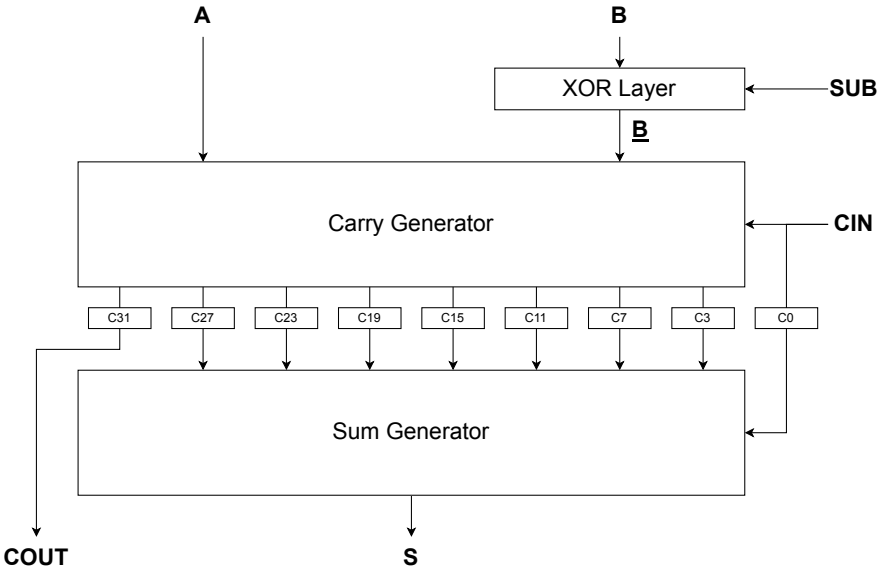

Figure 4.10: Pentium 4 Adder Block Diagram

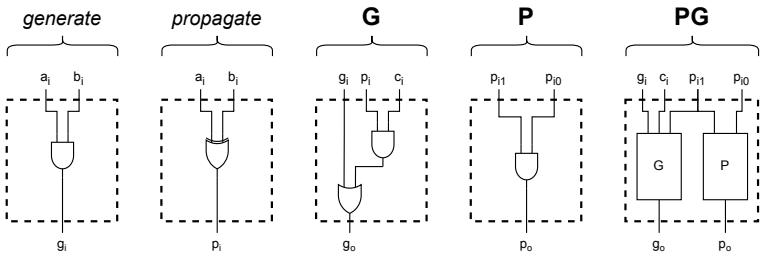

Figure 4.11: Carry Generator Basic Blocks

4.7.5 Comparator Logic

The comparator unit evaluates conditions between two operands. A block diagram of the architecture is shown in Figure 4.15.
The logic uses ARM-like status flags to compute the output. They are generated by performing a subtraction between the two operands and are calculated as follows:

- **N (Negative)**: `result[31]`
- **Z (Zero)**: `result[31:0] == 0`
- **C (Carry)**: `c_out`

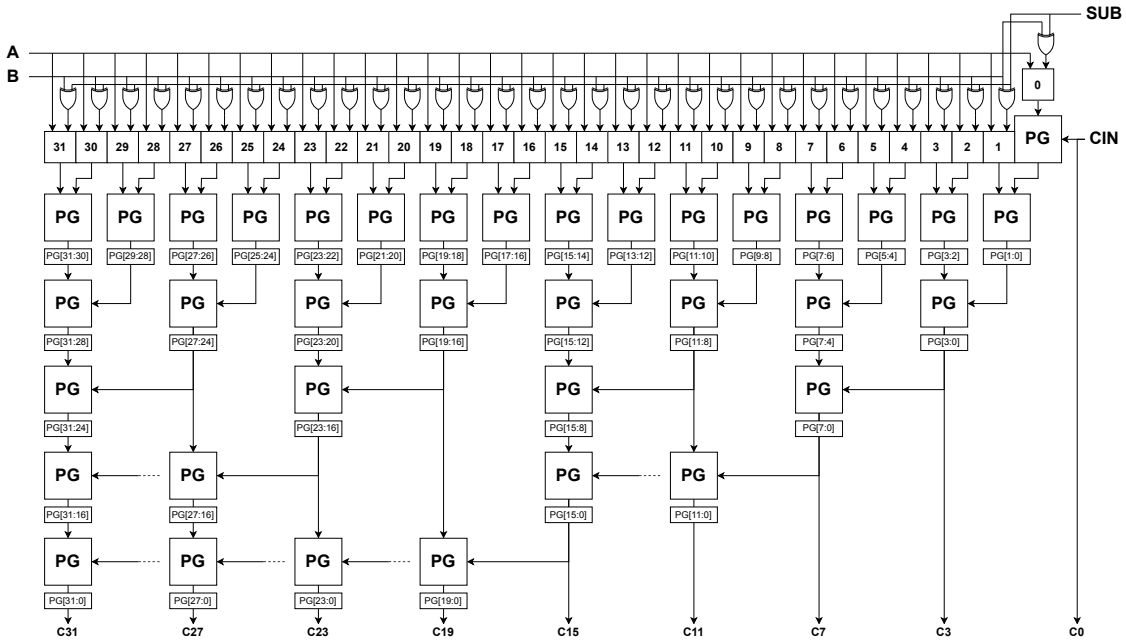

Figure 4.12: Carry Generator Detailed Architecture

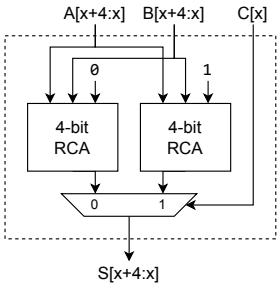

Figure 4.13: Carry Select Adder

- **V (Overflow)**: `(A[31] & ~B[31] & ~result[31]) | (~A[31] & B[31] & result[31] )`

These flags are then combined to compute the comparison result, and all cases are shown in Table 4.2.

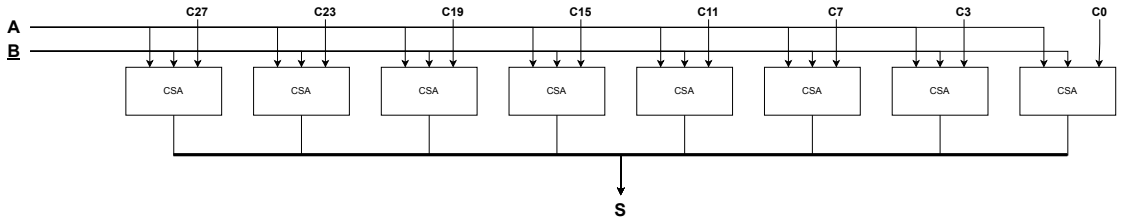

Figure 4.14: Sum Generator Block Diagram

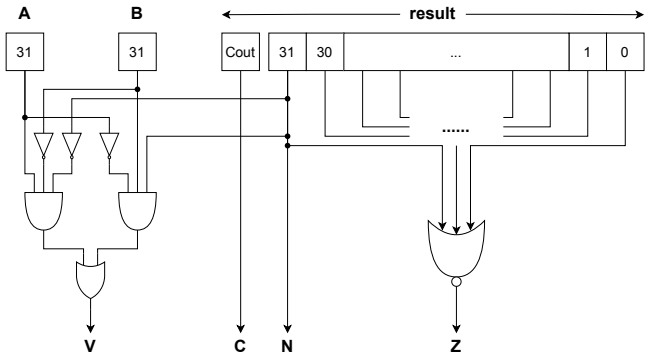

Figure 4.15: Comparator Logic Block Diagram

Condition	**Description**	**Logic**
seq	equal to	`Z`
sne	not equal to	`NOT(Z)`
slt	signed less than	`N XOR V`
sgt	signed greater than	`NOT(Z) AND (N XNOR V)`
sle	signed less than or equal to	`Z OR (N XOR V)`
sge	signed greater than or equal to	`N XNOR V`
sltu	unsigned less than	`NOT(C)`
sgtu	unsigned greater than	`C AND NOT(Z)`
sleu	unsigned less than or equal to	`Z OR NOT(C)`
sgeu	unsigned greater than or equal to	`C`

Table 4.2: Comparator Logic Outputs

CHAPTER 5

Hazard Handling

This chapter analyzes data, control, and structural hazards in a simple in-order microprocessor.

The analysis begins with a hypothetical baseline micro-architecture that lacks hardware-based hazard resolution, relying entirely on compiler scheduling to ensure correct execution. This approach is used to understand the performance penalties inherent to each hazard type. Subsequently, the principles of compiler-based hazard prevention are translated into hardware mechanisms, integrating hazard handling directly into the microprocessor's logic. Finally, optimizations are implemented to minimize the performance impact of these hazards.

5.1 The Baseline In-Order Micro-architecture

The hypothetical baseline micro-architecture implements a simple in-order 5-stage pipeline processor. It has no hardware mechanisms to resolve hazards: instruction scheduling to avoid hazards is entirely the compiler's responsibility. The simplest technique is inserting `NOPs` between dependent instructions and after any control-flow instruction.

5.2 Analysis of Pipeline Hazards

5.2.1 Data Hazards

A data hazard occurs when an instruction cannot execute in its intended clock cycle because the data it requires is not yet available. This particular case is called a Read-After-Write hazard. By design, an in-order processor is not affected by hazards caused by anti-dependencies, namely Write-After-Read (WAR) and Write-After-Write (WAW), so these cases are not discussed further.

In this micro-architecture, source operands can only be read from the RF. If an instruction depends on the result of a preceding instruction, it must be delayed until the previous instruction completes execution.

For example, consider an addition whose result, stored in `R1`, is used by the immediately following AND instruction:

```
ADD R1, R2, R3
AND R4, R5, R1
```

The result of `ADD` is written to the RF during the WB stage. Therefore, the `AND` instruction can only read the correct value from the RF in the following clock cycle. To synchronize correctly,

the compiler must schedule the `AND` so that its ID stage occurs one cycle after the `ADD`'s WB stage, requiring exactly three `NOPs`. The pipeline behavior is shown in Figure 5.1.

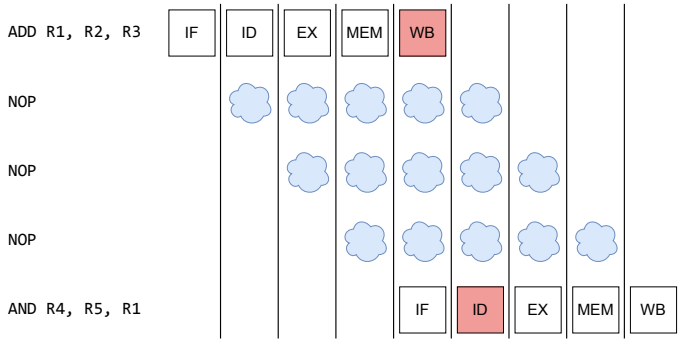

Figure 5.1: Pipeline view of a Read-After-Write Hazard handled by inserting `NOPs`

This example demonstrates that in this micro-architecture, the worst-case RAW hazard incurs a penalty of three clock cycles, during which some pipeline stages remain unused, degrading performance.

5.2.2 Control Hazards

A control hazard is a problem in a pipelined processor where the CPU cannot know the address of the next instruction to fetch because it is executing a conditional branch or jump.

This naive micro-architecture lacks a dedicated adder for calculating program counter offsets in control-flow instructions. Instead, jump and branch targets are computed using the main ALU in the EX stage, and the PC is updated in the MEM stage. This design creates a three-cycle delay between fetching a control-flow instruction and resolving the next PC value. During this delay, the pipeline continues fetching sequential instructions. Since the pipeline registers cannot be flushed, the compiler handles this hazard by inserting `NOP` instructions after each jump or branch. These `NOP` instructions prevent incorrect execution by occupying pipeline slots until the correct program counter value is available.

This approach results in a fixed three-cycle penalty for all control-flow instructions, regardless of whether the branch is taken or not. Figure 5.2 illustrates this pipeline behavior.

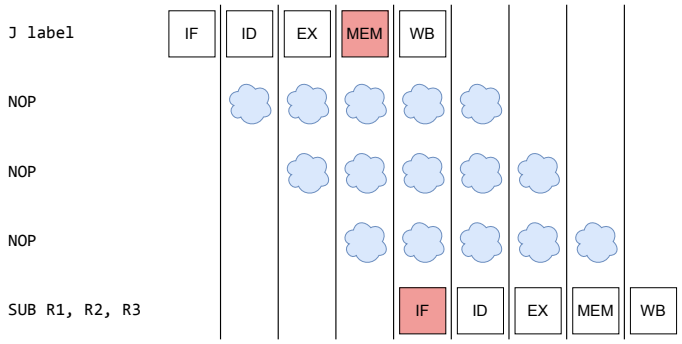

Figure 5.2: Pipeline view of a Control Hazard handled by inserting `NOPs`

5.3 Hardware-Based Hazard Resolution

As seen earlier, this naive micro-architecture can avoid hazards through compiler scheduling alone. However, hardware hazard detection is implemented to demonstrate how this complexity can be shifted from software to hardware, illustrating how processors manage dependencies at runtime even when it is not strictly necessary. Alongside this implementation, additional performance optimizations are introduced to either eliminate certain stalls entirely or significantly reduce their penalty.

5.3.1 Data Forwarding Logic

The first hardware enhancement is the introduction of forwarding paths in the design. Consider again the previous example of a RAW hazard:

```
ADD R1, R2, R3
AND R4, R5, R1
```

The result of the first instruction, although not yet written back to the RF, becomes available after the EX stage. An additional path can therefore be introduced to forward this result directly to the subsequent instruction, avoiding the need to wait for it to be read from the RF. From a pipeline perspective, this behavior is illustrated in Figure 5.3.

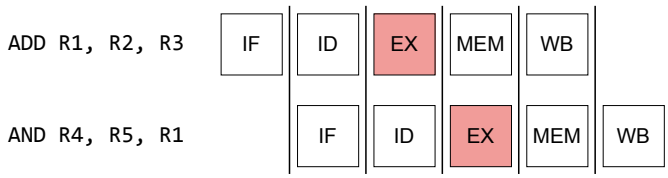

Figure 5.3: Pipeline view of a data hazard resolved through forwarding

This mechanism completely eliminates the hazard in this case. However, if the first instruction is a memory read, the situation differs. The result of a load instruction becomes available only after the MEM stage. Therefore, a `NOP` is still required when the subsequent instruction depends on this result, as in the following example:

```
LW R3, Imm(R0)
SUB R1, R2, R3
```

This delay, known as the load-use delay, is illustrated from a pipeline perspective in Figure 5.4. It also implies the presence of an additional forwarding path that connects the output of the MEM stage to the EX stage, allowing load results to be forwarded as soon as they become available.

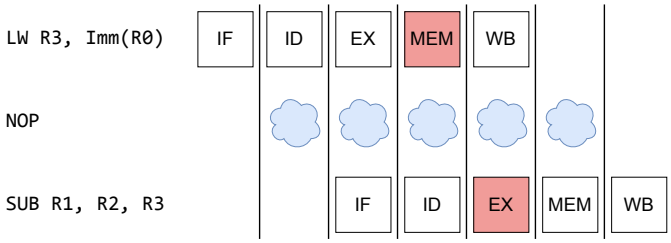

Figure 5.4: Pipeline view of the load-use delay

5.3.2 Hazard Unit

With the forwarding mechanism in place, most data hazards are resolved. In this simplified design, the hazard unit only intervenes when detecting an instruction that depends on the result of an immediately preceding load instruction. In such cases, the unit identifies the dependency and automatically inserts a `NOP` between the two instructions.

The second unresolved source of hazards is control hazards. The hazard unit must detect when a control-flow instruction results in a taken branch. If the branch is taken, all preceding pipeline stages are flushed, effectively executing `NOPs`. If the branch is not taken, no action is required since the correct sequential instructions are already present in the pipeline.

This dynamic detection removes roughly half of the control hazards previously handled by compiler scheduling. Earlier, `NOPs` had to be inserted statically, regardless of whether the branch was actually taken. With hardware detection, `NOPs` are introduced only when a real control-flow change occurs.

Assuming control-flow instructions are still completed in the MEM stage, three `NOPs` are still required. This means that the worst-case control hazard penalty remains three clock cycles, the same as in the baseline processor without a hazard unit.

5.3.3 Branch/Jump Penalty Optimization

This section addresses the control hazard penalty by introducing a dedicated branch handling unit. In the baseline design, branch and jump targets are computed in the EX stage and completed in the MEM stage, which requires flushing multiple pipeline stages and results in a worst-case penalty of three clock cycles.

This optimization moves the computation of branch and jump targets to the ID stage. This is achieved by introducing a dedicated adder specifically for control flow instructions, which calculates the target address as soon as the instruction reaches the decode stage. As a result, the pipeline only needs to flush the instruction currently in the IF stage on a jump or when a branch is taken.

With this dedicated hardware, the worst-case control hazard penalty is reduced from three clock cycles to a single clock cycle, significantly improving performance without affecting the in-order execution of other instructions.

On the bad side, moving this computation to the ID stage introduces new kinds of data hazards and new forwarding paths to the ID stage. These new hazard cases are shown in Figures 5.5 and 5.6 and will be further discussed in the following section.

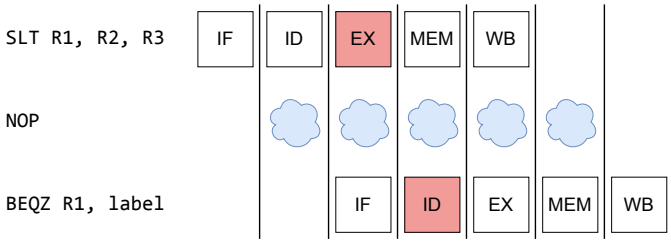

Figure 5.5: Pipeline view of the ALU-Branch hazard

5.4 Hazard Handling Implementation

Having analyzed hazards and explored the implications of hardware-based resolution, building from a naive implementation towards a fully hardware-based resolution with early branch execution, we now turn to the concrete implementation details. This section presents the actual design of the Hazard

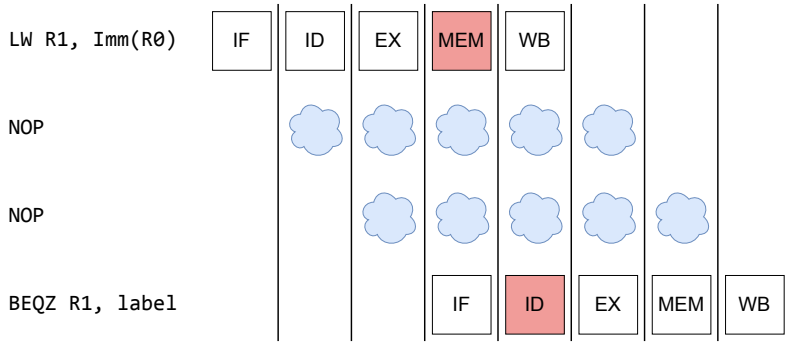

Figure 5.6: Pipeline view of the Load-Branch hazard

and Forwarding Units, highlighting how the principles discussed previously are realized in the specific case of the implemented DLX core.

5.4.1 Hazard Unit

The developed Hazard Unit executes exactly two operations:

- Stall with `NOP` insertion.
- Flush of the IF stage.

These two operations are performed through four output signals, as shown in the interface in Figure 5.7:

- `insert_nop`: When set, the input control signals to the ID-EX register bank are masked to perform a `NOP`.
- `flush`: When set, the contents of the IF-ID register bank are cleared.
- `write_enable`: When unset, the IF-ID register bank is not updated with new values, keeping the old ones.
- `pc_enable`: When unset, the PC is prevented from updating with a new value, keeping the old one.

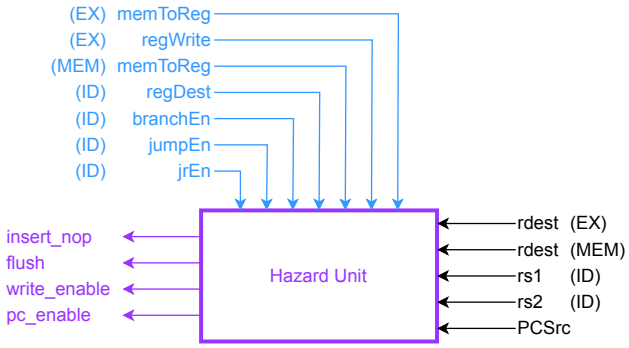

Figure 5.7: Hazard Unit Interface

Signal	Stall (NOP insertion)	Flush	Normal
insert_nop	1	0	0
flush	0	1	0
write_enable	0	1	1
pc_enable	0	1	1

Table 5.1: Hazard Unit output signals in different operating conditions

Table 5.1 shows how these signals are set during normal execution, as well as during stall and flush operations.

A pipeline stall is triggered whenever a data or control dependency prevents the instruction in the decode stage from executing correctly. The specific stall conditions implemented are the following.

Load–Use Data Hazard The instruction in the EX stage is a load and its destination register matches one of the source registers (RS1 or RS2) of the instruction currently in the ID stage. The dependent instruction must wait for the load result to become available in the MEM stage.

ALU–Branch or ALU–JR Hazard The instruction in the EX stage writes to a register, and this destination register matches the source register RS1 used by a branch or JR instruction currently in the ID stage. The branch or JR depends on a value still being computed in the EX stage.

Load–Branch or Load–JR Hazard The instruction in the MEM stage is a load whose destination register matches the source register RS1 used by a branch or JR instruction in the ID stage. The branch or JR depends on a value being loaded from memory, which is not yet available.

Finally, a flush is performed only when no stall is required and a valid change in control flow is detected. This occurs when the PCSrc signal is asserted, selecting the branch/jump target address instead of the sequentially incremented PC.

5.4.2 Forwarding Unit

The Forwarding Unit implements data forwarding to minimize stalls caused by RAW hazards. It enables the results of previous instructions to be used as soon as they are available, without waiting for them to be written back to the RF. The unit generates three output signals, summarized in Figure 5.8.

- forwardA: 2-bit control signal selecting the source for the first ALU operand (RS1).
- forwardB: 2-bit control signal selecting the source for the second ALU operand (RS2).
- forwardC: 1-bit control signal enabling forwarding for branch and JR instructions in the ID stage.

Each control signal selects the appropriate forwarding path depending on data availability in the pipeline, according to the rules summarized in Table 5.2.

The forwardC signal does not include a configuration to forward from the WB stage, since data from this stage is already directly available to the ID stage. For this reason, the corresponding logic was implemented directly within the ID stage rather than in the Forwarding Unit itself. This mechanism, referred to as the *Bypass*, is shown in Figure 4.2, to the right of the RF.

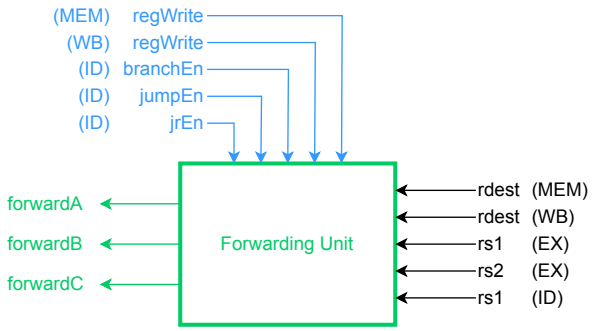

Figure 5.8: Forwarding Unit Interface

Signal	From MEM	From WB	No Forwarding
`forwardA`	10	01	00
`forwardB`	10	01	00
`forwardC`	1	–	0

Table 5.2: Forwarding Unit output encoding

A forwarding operation is triggered whenever the destination register of a preceding instruction matches a source register of the instruction currently in the EX or ID stages. The specific cases handled are the following.

MEM → EX Forwarding When the instruction in the MEM stage writes to a register and this register matches either RS1 or RS2 of the instruction in the EX stage, the result is forwarded directly from the EX-MEM pipeline register.

WB → EX Forwarding When the instruction in the WB stage writes to a register and this register matches either RS1 or RS2 of the instruction in the EX stage, the result is forwarded directly from the MEM-WB pipeline register.

MEM → ID Forwarding Branch and `JR` instructions in the ID stage may depend on a value in the MEM stage. In this case, the forwarding path controlled by `forwardC` provides the correct operand for the branch comparison or target computation.

CHAPTER 6

Flow Automation

The development of the DLX CPU core required a digital design flow spanning multiple steps from RTL simulation to physical implementation. To manage this complexity and ensure reproducibility, an automated build system was developed using GNU Make to control the entire design process.

6.1 Makefile Architecture

The automation system is built around a centralized Makefile that provides a unified interface to all design tools and workflows. This approach offers several key advantages:

- **Consistency**: All team members use identical tool configurations and procedures
- **Reproducibility**: Build results are consistent across different executions
- **Efficiency**: Automated dependency tracking eliminates manual intervention
- **Documentation**: The Makefile serves as executable documentation of the design flow

The Makefile organizes the project into logical directories with clear separation of goals:

- `src/` for VHDL source code and packages
- `scripts/` for automation scripts and utilities
- `asm/` for test programs and assembly code
- `sim/`, `syn/`, `pnr/` for build artifacts from different stages

6.2 Automated Instruction Package Generation

A critical automation feature is the dynamic generation of the instruction package from JSON definitions. Before any simulation target, the Makefile automatically invokes a Python script that processes the JSON instruction specification and generates the `instructions_pkg.vhd` file. This ensures that instruction definitions remain synchronized between the JSON specification and VHDL implementation, eliminating manual updates to instruction encodings.

6.3 Integrated Simulation Flow

The simulation automation provides a hierarchical testing approach. For the most critical blocks, component-level testing is performed to ensure the correct behavior of the block in isolation. Finally, an integration test is performed with the full CPU.

6.3.1 Component-Level Testing

Individual components like the T2 shifter and P4 adder can be simulated in isolation:

```
make t2-logic-tb
make t2-shifter-tb
make p4-adder-tb
make alu-tb
```

These testbenches are fully self-checking, incorporating internal behavioral models that automatically verify that the structural RTL implementation of the Device-Under-Test (DUT) produces the correct outputs.

6.3.2 Full CPU Simulation

The complete CPU testbench workflow executes the following steps:

1. Automatically assembles the specified ASM file into instruction memory format
2. Executes the Python-based DLX emulator to generate reference outputs
3. Compiles all VHDL sources including the generated instruction package
4. Runs QuestaSim simulation with the test program

DLX Emulator

To support the development and verification of the DLX processor, a Python-based reference emulator was implemented. This single-cycle emulator serves as a golden model for validating the correctness of both the assembly programs and the RTL implementation.

The choice of a single-cycle architecture for the emulator was strategic. While the actual HDL implementation employs pipelining and other microarchitectural optimizations, the emulator operates at a higher level of abstraction with several key advantages:

- **Simplicity and Development Speed**: A single-cycle model is the most straightforward implementation of the ISA, allowing for rapid development and easier debugging. The emulator was developed in significantly less time than a cycle-accurate pipelined model would have required.
- **Reduced Complexity**: By avoiding pipeline hazards, data forwarding, and stall logic, the emulator focuses exclusively on architectural correctness rather than microarchitectural details. This abstraction level makes it ideal as a reference model.
- **Minimized Bug Correlation**: The implementation approach differs fundamentally from the RTL design. This diversity significantly reduces the risk of identical bugs appearing in both implementations.
- **Deterministic Behavior**: Without pipeline effects, the emulator provides predictable, instruction-by-instruction execution that is easier to reason about during initial program development and debugging.

The emulator reads the same hexadecimal memory image generated by the assembler as the HDL simulation. It models the DLX instruction set architecture, including:

- All arithmetic and logical operations supported by the RTL
- Control-flow instructions
- Memory access operations
- Signed and unsigned data handling with proper sign extension

Key features of the emulator include:

- **Debug Mode**: When enabled with the `--debug` flag, the emulator prints the state of all 32 registers after each execution cycle
- **Final State Reporting**: After program completion, the emulator displays the final register file contents and the complete data memory state
- **Memory Alignment Checking**: Properly detects and reports misaligned memory accesses
- **Safety Limits**: Includes a configurable maximum cycle count to prevent infinite loops during testing

The verification workflow is straightforward: the same assembly program is executed on both the Python emulator and the RTL simulation. The final register states and memory contents from both executions are compared to validate functional correctness. Any discrepancies indicate either an error in the HDL implementation or an inconsistency in the architectural understanding.

This integrated approach ensures that any assembly program can be immediately tested on both the reference emulator and HDL implementation, significantly accelerating the development and debugging cycle while providing high confidence in the correctness of the processor implementation.

6.4 Configurable Synthesis Flow

The synthesis automation supports multiple optimization strategies through a configurable flattening parameter:

```
make synthesis FLATTEN=all    # Full flattening (default)
make synthesis FLATTEN=auto   # Automatic flattening for delay optimization
make synthesis FLATTEN=none   # No flattening
```

Each configuration stores results in separate directories (`syn/results_all/`, `syn/results_auto/`, etc.), enabling comparative analysis of different optimization strategies.

6.5 Post-Synthesis Verification

A critical quality assurance step is the post-synthesis simulation, which verifies the gate-level netlist maintains functional correctness:

```
make post-syn-cpu-tb FLATTEN=auto
```

This target automatically uses the synthesized netlist from the specified flattening configuration and runs the same testbench used for RTL simulation, ensuring equivalence between pre- and post-synthesis behavior.

6.6 Physical Implementation Automation

The Place-and-Route (PnR) flow extends the configurability to physical implementation:

```
make pnr FLATTEN=auto
```

This invokes Innovus in batch mode with the appropriate synthesized netlist, maintaining consistency between synthesis and PnR configurations.

6.7 Maintenance and Cleanliness

The automation system includes cleanup targets that maintain a clean working environment:

- `clean_sim`: Removes simulation artifacts
- `clean_syn` – `purge_syn`: Cleans synthesis results
- `clean_pnr` – `purge_pnr`: Cleans PnR results
- `clean`: Complete cleanup of all generated files

These targets ensure that builds can be reproduced from scratch.

6.8 Execution Workflow

The typical development workflow using the automation system proceeds as follows:

1. Navigate to project root directory
2. Develop and test individual components with targeted simulations
3. Verify complete CPU functionality with custom assembly programs
4. Perform synthesis with appropriate optimization strategy
5. Validate post-synthesis netlist functionality
6. Execute Place-and-Route for physical implementation
7. Clean build artifacts between major changes

CHAPTER 7

Synthesis

This chapter explores the synthesis process which transforms the RTL description into a gate-level netlist optimized for timing, area, and power constraints. The digital design was synthesized using *Synopsys Design Compiler*.

7.1 Constraints Definition

7.1.1 Timing Constraints

The primary timing constraint was applied to the main clock signal:

- **Clock Period**: Varied based on flattening strategy:
  - Full flattening: 1.50 ns
  - Automatic ungrouping: 1.85 ns
  - No flattening: 2.01 ns
- **Clock Uncertainty**: 70 ps to account for clock skew and jitter

7.1.2 Input/Output Constraints

The following I/O constraints were applied to ensure proper interface timing:

- **Input Delay**: 500 ps maximum for all inputs (excluding clock)
- **Output Delay**: 500 ps maximum for all outputs
- **Output Load**: Equivalent to `BUF_X4` input capacitance from Nangate library

A maximum path delay constraint of one clock period was set between all input and output pairs to ensure combinational paths meet timing requirements.

7.2 Optimization Strategy

7.2.1 Flattening Approaches

Flattening is an optimization technique in logic synthesis where the hierarchical structure of a design is removed, creating a single, flat netlist without module boundaries. Three different optimization strategies were evaluated based on this:

Full Flattening Complete hierarchy removal for maximum optimization across module boundaries. This approach achieved the most aggressive timing with a 1.50 ns clock period.

Automatic Ungrouping Compilation with `-auto_ungroup delay` option, allowing the tool to selectively flatten hierarchy based on timing critical paths. This approach reached a 1.85 ns clock period.

Hierarchical Preservation Maintained full design hierarchy during compilation. This configuration reaches a 2.01 ns clock period.

The *Hierarchical Preservation* approach is kept despite its lower performance for two reasons:

1. It serves as a baseline for the other two flattening options.
2. It provides useful diagnostic insights. Since the hierarchy is fully preserved, the synthesis tool can attribute cells to their original modules, report area by module, and trace the critical path through specific hierarchical blocks. This allows for precise correlation between gate-level and RTL, making it easier to identify bottlenecks and potential optimization points in the design.

7.2.2 Compilation Effort

The synthesis flow uses different compilation commands depending on the selected flattening option:

- `compile_ultra`: Performs the most aggressive optimization. It enables advanced transformations such as register retiming, logic restructuring, and high-effort mapping to achieve the best possible timing and area results.
- `compile -map_effort high -auto_ungroup delay`: Runs a high-effort compile with timing-driven optimizations. The tool focuses on improving delay on critical paths while keeping compilation runtime reasonable.
- `compile -map_effort high`: Uses a standard high-effort compile, applying timing and area optimizations within each module.

CHAPTER 8

Physical Implementation

The final stage of the digital design flow involved the **Place-and-Route** of the synthesized core, carried out using *Cadence Innovus*. This process transforms the synthesized gate-level netlist into a complete physical layout that satisfies both electrical and geometric constraints. The implementation followed the standard flow provided in the laboratory documentation, and it was automated through the `place_and_route.tcl` script.

8.1 Flow Overview

The main steps of the physical implementation are detailed below.

1. **Importing the Design**

   The first step consisted of importing the synthesized gate-level netlist and the corresponding technology libraries. This was accomplished by editing and sourcing the `scripts/design.globals` file, which defines:

   - The top-level module name.
   - The path to the synthesized Verilog netlist.
   - References to the `Nangate 45 nm` standard-cell technology libraries.

   The Multi-Mode Multi-Corner (MMMC) configuration file, `scripts/mmm_design.tcl`, was also loaded to establish the timing views for setup and hold analysis across process, voltage, and temperature corners. This ensured that the Innovus environment was correctly initialized with consistent technology, constraint, and timing information.

2. **Floorplanning**

   The floorplan defines the physical dimensions and placement regions for the design. The following parameters were applied:

   $$\begin{aligned} \text{Aspect ratio} &= 1:1 \\ \text{Core utilization} &= 60\% \\ \text{Core margins} &= 5\ \mu\text{m on all sides} \end{aligned}$$

   The resulting die area is automatically determined by the tool based on the estimated total cell area and the specified utilization factor.

3. **Power Planning**

   Power rings were added around the core boundary to distribute the `VDD` and `VSS` supplies. Each ring was defined with a metal width and spacing of 0.8 μm. Horizontal and vertical power stripes were inserted to connect the standard-cell rows, to provide uniform power distribution within the core.

4. **Placement and Clock Tree Synthesis**

   Standard-cell placement was performed automatically, followed by optimization to minimize wirelength and improve timing. Input/Output (I/O) pins were positioned along the periphery according to the floorplan constraints. After placement, Clock Tree Synthesis (CTS) was executed to create a balanced clock distribution network.

   The CTS phase was configured with:

   $$\text{Maximum transition time} = 0.08 \text{ ns}$$
   $$\text{Target clock skew} = 0.5 \text{ ns}$$

   Both pre-CTS and post-CTS optimizations were run to correct setup timing issues.

5. **Signal Routing**

   After CTS, the detailed routing step connected all signal nets using the available metal layers defined in the technology file. Routing congestion and timing violations were resolved through incremental optimization. Filler cells were automatically inserted to ensure continuous well and diffusion regions ($n+$ and $p+$) along each standard-cell row.

6. **Timing and Design Analysis**

   A parasitic extraction was performed on the final routed design to obtain accurate resistance and capacitance (RC) data. Post-route Static Timing Analysis (STA) was then carried out for both setup and hold conditions across all MMMC views. Finally, the design was verified through:

   - **Connectivity checks** to confirm proper net connections.
   - **Design Rule Check (DRC)** to ensure compliance with the foundry technology constraints.

The Innovus physical implementation produced a fully placed, routed, and verified design, ready for GDSII generation.

CHAPTER 9

Results

9.1 Post-Synthesis Analysis

Three synthesis configurations were evaluated: `all`, `auto`, and `none`. The `all` configuration applies full flattening for maximum performance, `auto` selectively ungroups timing-critical hierarchies for delay optimization, and `none` preserves full hierarchy for debugging and readability. Results are presented in parallel for all three.

9.1.1 Post-Synthesis Reports

Analysis reports were generated post-synthesis using *Synopsys Design Compiler*. Each configuration includes detailed power, area, and timing information.

Power Report Summary

Table 9.1 summarizes the power consumption for the three configurations: `all`, `auto`, and `none`. The observed differences correlate directly with the timing performance achieved by each configuration.

The `all` optimization achieves the fastest clock period and, consequently, shows the highest total power consumption of 6.1453 mW. This increase is mainly due to a higher *Internal Power* component, as dynamic and internal power scale with operating frequency and cell activity. Although `all` uses optimization techniques such as cell upsizing and logic restructuring, it maintains relatively low *Leakage Power*, probably indicating a very good selection of low-leakage cells.

The `auto` configuration, operating at a more moderate clock period, shows intermediate power behavior. Its total power is slightly lower than that of `all`, but it has the highest *Switching Power*.

Finally, the `none` configuration, with the slowest timing, results in the lowest overall power consumption. The reduction comes primarily from decreased internal activity and lower operating frequency, leading to reduced *Internal Power* and *Switching Power*.

Metric	**all**	**auto**	**none**
Internal Power (mW)	5.3577	4.9190	4.5454
Switching Power (μW)	528.285	604.876	533.876
Leakage Power (μW)	259.26	317.86	316.89
Total Power (mW)	6.1453	5.8418	5.3961

Table 9.1: Post-Synthesis Power Report Summary

Table 9.2 shows how combinational and sequential blocks impact the total power. Across all three configurations, the power drawn by combinational blocks is roughly one-fourth that of sequential blocks, with sequential blocks dominating the total power consumption. This imbalance is even more pronounced in the all configuration, where sequential blocks account for an even larger fraction of the total power. The reasons could be multiple:

- Full flattening removed many combinational cells, reducing the overall impact of their power group.
- Retiming techniques increased the proportion of flip-flops, further moving the balance toward sequential power.

Power Group	**all**	**auto**	**none**
Register	85.41 %	81.08 %	81.55 %
Combinational	14.59 %	18.92 %	18.45 %

Table 9.2: Percentage Contribution of Power Groups Post-Synthesis

Area Report Summary

Table 9.3 summarizes the post-synthesis area metrics for the three configurations. The data show that the `all` configuration results in the smallest total cell area and the lowest number of cells, compared to `auto` and `none`. This reduction is primarily due to a smaller combinational area and a significantly reduced buffer/inverter area.

In contrast, the `auto` and `none` configurations have comparable total areas and cell counts, with larger contributions from both combinational and buffer/inverter areas. The non-combinational (sequential) area remains relatively consistent across all configurations, suggesting that the area differences are mostly driven by optimizations in the combinational logic and associated buffering.

Overall, the `all` configuration has the most area-efficient implementation, likely as a result of full flattening and optimization during synthesis, which reduces combinational complexity and buffer overhead without significantly affecting sequential elements.

Metric	**all**	**auto**	**none**
Comb. Area (μm^2)	6,966.01	8,555.62	8,505.08
Buf/Inv Area (μm^2)	346.864	1,260.57	1,250.20
Noncomb. Area (μm^2)	6,572.33	6,750.02	6,748.42
Total Cell Area (μm^2)	13,538.3	15,305.6	15,253.50
#Cells	7,299	9,892	9,813

Table 9.3: Post-Synthesis Area Report Summary

9.1.2 Critical Path Analysis

The critical path was extracted from the Design Compiler timing report for each configuration. The startpoint and endpoint of the critical path of each configuration are reported in Table 9.4.
In the `all` configuration, full flattening results in the lack of hierarchy visibility, making the timing path almost impossible to trace back to functional blocks, without other hints.
The `auto` configuration partially preserves hierarchy, easing readability while still optimizing delay-critical modules.

Configuration	Startpoint	Endpoint
all	mem_wb_regs_inst/memToReg_o_reg	ex_mem_regs_inst/aluout_o_reg[0]
auto	ex_mem_regs_inst/rdest_o_reg[4]	ex_mem_regs_inst/aluout_o_reg[0]
none	ex_mem_regs_inst/rdest_o_reg[4]	ex_mem_regs_inst/aluout_o_reg[0]

Table 9.4: Critical path start and end points for each configuration.

The none configuration retains complete hierarchy, producing a clear and readable critical path.

For these reasons the none configuration will be used to better analyze the critical path.

Critical path of the none configuration

The critical path traverses the following modules in order:

1. **MEM-WB Pipeline Registers** (mem_wb_regs_inst)
   Source flip-flop.
   Path delay : 0.00 ns → 0.15 ns.

2. **Forwarding Unit** (forwarding_unit_inst)
   Small combinational logic producing forwardB_o[0].
   Path delay : 0.15 ns → 0.37 ns.

3. **Execution stage** (execute_inst)
   Signal forwardB_i[0] enters mux_rdata2, the selected data then goes through mux_rdata2_imm selecting the ALU operand.
   Path delay : 0.37 ns → 0.74 ns.

4. **ALU block** (alu_inst)
   Data enters the p4_adder sub-block: XOR layer, carry generator, and sum generator.
   Path delay : 0.74 ns → 1.32 ns.

5. **ALU comparator and final logic**
   Data goes through the comparator logic and the final ALU multiplexer.
   Path delay : 1.32 ns → 1.82 ns.

6. **Final multiplexer and output register input**
   Data coming from the ALU is selected against the npc (mux_npc_alu), the output is sampled by the EX-MEM Pipeline Register.
   Path delay : 1.82 ns → 1.90 ns.

In summary, the critical path starts from a register in the EX-MEM pipeline register, passes through the forwarding unit and EX stage operand multiplexers, then traverses the ALU's arithmetic (adder/-carry/sum) and comparator logic, before reaching the EX-MEM pipeline register. The largest incremental delays occur inside the ALU sub-blocks (XOR layer, carry generator) and the comparator logic, which contain many multi-input gates.

For configurations all and auto, although the startpoint differs, the paths quickly converge to the EX stage, ultimately sharing the same endpoint.

This convergence is still visible in the auto configuration. The two traces converge almost immediately at the Forwarding Unit output (forwardB_o[0]) and then traverse the same exact path through front-end multiplexers, ALU and final multiplexer into the EX-MEM pipeline register. This

optimized synthesis reduces several intermediate cell delays (hence slightly smaller cumulative numbers), but it does not fundamentally change the logical chain: the critical path remains essentially the same sequence of modules and gates.

In the fully flattened netlist, the hierarchy has been completely removed, making it difficult to trace the exact sequence of cells that the critical path traverses. The timing report only lists low-level standard cells, without any reference to the original RTL module boundaries.
Nevertheless, knowing the startpoint and endpoint, it is possible to reasonably infer that the path still passes through the same logical regions as in the previous builds. In particular, the two traces would likely converge at the front-end multiplexers within the EX stage and from there follow the same route into the EX-MEM pipeline register.

Although cell-level delay optimizations and buffer insertions slightly alter local timing characteristics, the overall critical path structure remains consistent with the hierarchical and partially flattened configurations.

9.2 Post-PnR Results

Post-PnR reports were generated using *Cadence Innovus*. They include updated timing and area results that account for physical layout effects such as wire delays and congestion.

9.2.1 Post-PnR Timing Results

Tables 9.5 and 9.6 show the post-PnR timing results for *setup* and *hold* time respectively. All timing constraints are met with positive slack across all scenarios, indicating a successful implementation.

The physical implementation revealed superior timing performance compared to synthesis estimates. The clock period was initially constrained based on synthesis timing estimates, but the physical implementation revealed additional optimization opportunities. Using the available slack, the theoretical maximum achievable clock periods are calculated as:

- `all` **configuration**: Theoretical minimum period = 1.36 ns (1.50 - 0.142)
- `auto` **configuration**: Theoretical minimum period = 1.71 ns (1.85 - 0.140)
- `none` **configuration**: Theoretical minimum period = 1.75 ns (2.01 - 0.262)

Setup Time	**all**	**auto**	**none**
WNS (ns)	0.142	0.140	0.262
TNS (ns)	0.000	0.000	0.000
Violating Paths	0	0	0

Table 9.5: Setup Time Results

Hold Time	**all**	**auto**	**none**
WNS (ns)	0.108	0.133	0.123
TNS (ns)	0.000	0.000	0.000
Violating Paths	0	0	0

Table 9.6: Hold Time Results

9.2.2 Floorplan Analysis

The final floorplans illustrate the structural impact of each synthesis mode. Each configuration has two views: one showing cell placement and routing, and another showing module boundaries.

The post-synthesis area results correlate very well with the post-PnR results, not showing any significant difference. The post-PnR area results are shown in Table 9.7.

Metric	**all**	**auto**	**none**
Area (μm^2)	13,577.7	14,358.4	14,355.2
Cells	7,265	8,068	8,052

Table 9.7: Post-PnR Area Results

The final floorplans for the three configurations are shown in Figures 9.1, 9.2, and 9.3, corresponding to the `all`, `auto`, and `none` configurations, respectively.

The analysis begins with the `none` configuration, which preserves the module hierarchy. From this layout, the most area-intensive modules can be identified.

Table 9.8 lists the Level 1 module areas. The design area is dominated by the Instruction Decode stage due to the presence of the RF, which alone occupies 8616.5 μm^2. This is expected, as flip-flops contribute significantly more to area than combinational cells, and the RF essentially consists of a large array of flip-flops. The Register File is clearly visible in all floorplans, including the fully flattened configuration, appearing as a large blob that occupies about 50% of the total cell area.

The second largest contributor is the Execution stage, primarily containing the ALU, which occupies 1497.0 μm^2. The remaining smaller modules contribute relatively little area and are difficult to distinguish in the floorplan.

Module	**Area** (μm^2)	**Percent of top (%)**
decode_inst	9460.8	65.91
execute_inst	1841.5	12.83
id_ex_regs_inst	847.7	5.91
fetch_inst	643.2	4.48
if_id_regs_inst	465.2	3.24
ex_mem_regs_inst	400.3	2.79
mem_wb_regs_inst	379.6	2.64
control_unit_inst	131.1	0.91

Table 9.8: Level 1 Module Areas in `none` configuration

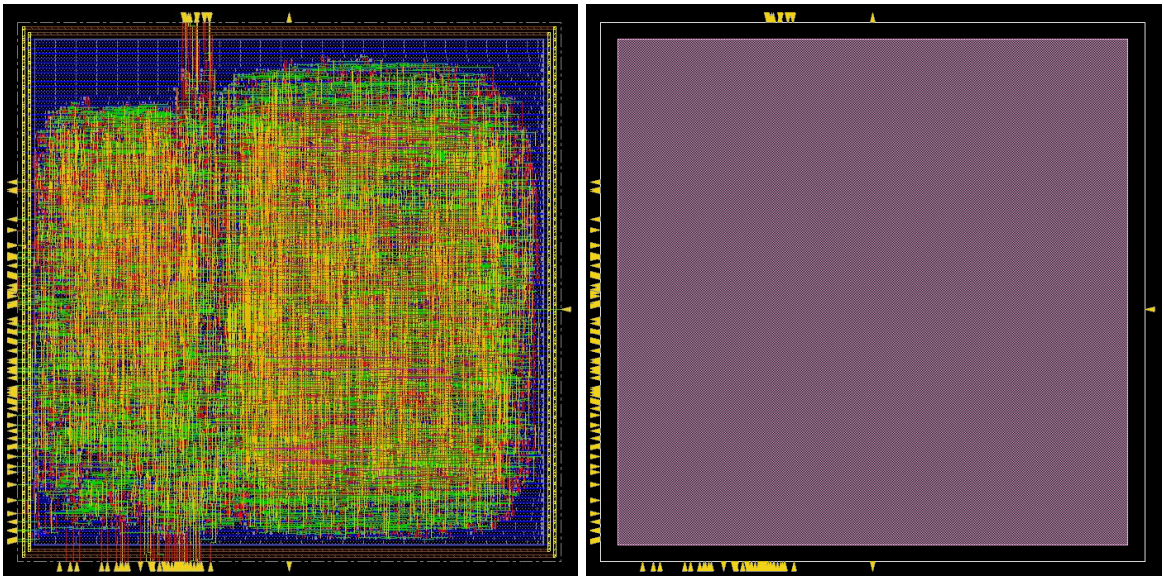

Figure 9.1: Final floorplan for `all` configuration. No visible module boundaries due to full flattening.

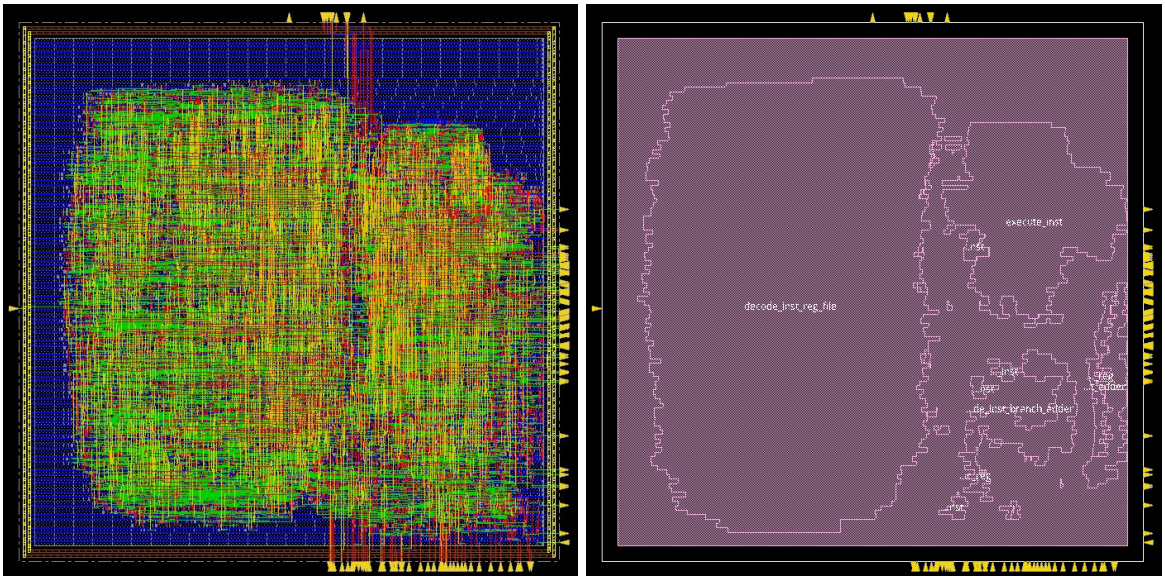

Figure 9.2: Final floorplan for `auto` configuration. Partial hierarchy visible.

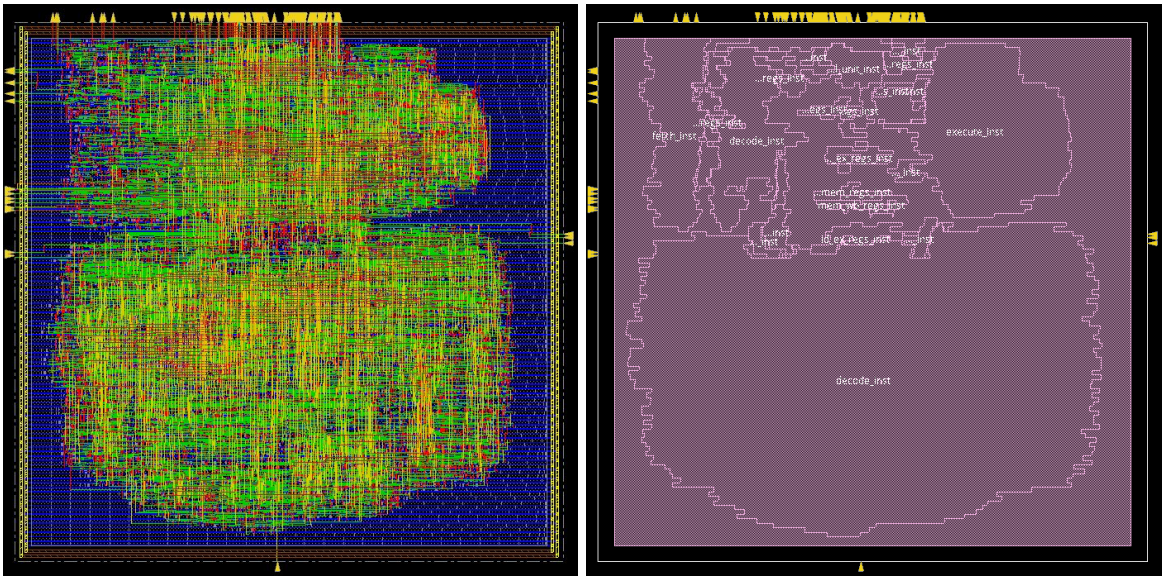

Figure 9.3: Final floorplan for `none` configuration. Full hierarchy boundaries visible.

CHAPTER 10

Conclusion

This project successfully accomplished the complete design and implementation of a DLX microprocessor, spanning the entire digital design flow from a high-level RTL description down to the physical implementation.

Key achievements of the project include:

- The development of a fully pipelined, five-stage DLX-Pro processor core, extending the basic DLX architecture with an expanded instruction set, full hazard handling via forwarding logic and a dedicated hazard unit, and early branch resolution to minimize control hazards.
- An automated design flow implemented using GNU Make, which allowed for consistency and reproducibility across all stages, from VHDL simulation with *Synopsys ModelSim*, through synthesis with *Synopsys Design Compiler* to physical implementation with *Cadence Innovus*.
- An in-depth exploration of synthesis strategies, comparing the effects of full flattening, automatic ungrouping, and hierarchical preservation on timing, area, and power.
- Successful physical design closure, with post-Place-and-Route timing analysis confirming that all setup and hold constraints were met with positive slack. The final layout validated the timing integrity of the design.
- The creation of a Python-based emulator that served as a golden reference model, significantly accelerating the verification process during the development cycle.

In conclusion, this project provided valuable hands-on experience in digital system design, successfully applying theoretical concepts to implement a functional DLX microprocessor.

Acronyms