Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Master degree in Electronics Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group_16

Battilana Matteo, La Greca Salvatore Gabriele, Pollo Giovanni

July 16, 2021

Feature

Frequency	400 MHz
Slack	0 (MET)
Area	35280.37
Total Dynamic power	10.02 mW
Cell Leakage power	657.45 μW

- 32 bit RISC CPU
- 400 MHz operating frequency
- Up to 4 GB of Addressable Memories
- Five stages pipeline
- Full Hazard Detection & Control
- System Tick Timer
- Enhanced T2 3-level Shifter
- Hardwired Control Unit
- Expanded instruction set
- Fast Subroutine call with Windowing RF
- Word, Half and Byte addressable Memory
- Signed/Unsigned multipurpose comparator
- Advanced Datapath with reduced delay
- Efficient CLA-Carry Select Adders
- Enhanced Booth's Multiplier

Contents

Listings

CHAPTER 1

Introduction

1.1 Abstract

The goal of this project is to build from scratch a working implementation of a DLX. To achieve the goal, some known blocks, created during laboratories, were used.

The second step was the design of the Datapath, done in the best possible way to obtain an high optimization and performance level. Some optimization examples, that will be explained more in-depth in the document, are the use of the P4 adder and the Booth multiplier inside the ALU, the comparator and many others.

The third step was the design of the Control Unit. Again, the choice fell on the microprogrammed version that guaranteed the pipeline implementation. To simplify the maintainability of the Control Unit, some struct-like constructs were used in the VHDL code.

In the fourth step, very exhaustive testing has been executed. All the proposed asm codes were verified, but some well-known algorithms (bubble sort, Fibonacci and factorial) were written and tested.

Last but not least, the DLX has been synthesized using Synopsis, and a post-synthesis simulation has been executed.

Thanks to the datapath optimization and the synthesis optimization, the microprocessor proposed in this paper reached a peak speed of 400MHz.

1.2 Workflow

As many tools we used to automate the working process, they are explained in this section.

The first and most important tool was versioning control. The choice fell on GitHub. Thanks to this, the team managed all versions of the code and stepped back if any problem occurs. In addition to that, team communication and issue management were very straightforward, thanks to the possibilities offered by the tool. Another handy feature was the milestone, which allows the team to be on time and respect deadlines.

The used programming technique was pair programming, which allows writing code and checking its correctness at the same time. This technique was beneficial under challenging parts of the projects. To exploit pair programming the extension used was Live Share for Visual Studio Code (`https://github.com/MicrosoftDocs/live-share`). Indeed, for the more manageable steps, the use of the branches and pull request gave the possibility of parallel working and drastically reduced the presence of conflicts.

The last thing to point out was the intensive use of scripting to compile VHDL code, simulate it, add waves, and then synthesize. All scripts are reported in the appendix.

CHAPTER 2

Hardware Architecture

2.1 Overview

This DLX is a 32-bit RISC processor with a five-stage pipeline. The external interface is made mainly for memories connection (IRAM, DRAM and a DRAM for the Register File) and the Clock and Reset signals. Inside it's possible to find the following blocks:

- Control Unit: it receives the fetched instruction from the IR register and starts to output the correct control signals towards all the pipeline stages. Moreover, it receives status signals from all other units about their working status like the comparator result (for branch decision), the status about possible hazards in the pipeline, Register File's Push & Pop operations under execution, and all the memories readiness. It's in charge of controlling the entire pipeline and stop it in case of hazards or other situations that require a stall.
- Decode Unit: part of the decode stage, it is in charge of keeping the status about all registers under usage (for further hazard controls), computation of the new Program Counter (given a Jump or not), data comparison (for branches) and, the most important thing, the operation decode with the dispatch of all the operands towards the right ports of the DataPath.
- DataPath: the computational core of the processor. Made of 4 pipeline stages (Instruction Decode, Execution, Memory, Write Back) contains all the units capable of computation. In particular, there is the Register File (that manages all the core registers), the Arithmetic Logic Unit, the Load-Store Unit for data memory management, and other units useful for the correct operation of everything.
- IR and PC: two registers that compose the Instruction Fetch stage of the pipeline are in charge of keeping in memory the current instruction under execution and the address for the next instruction to execute, respectively.

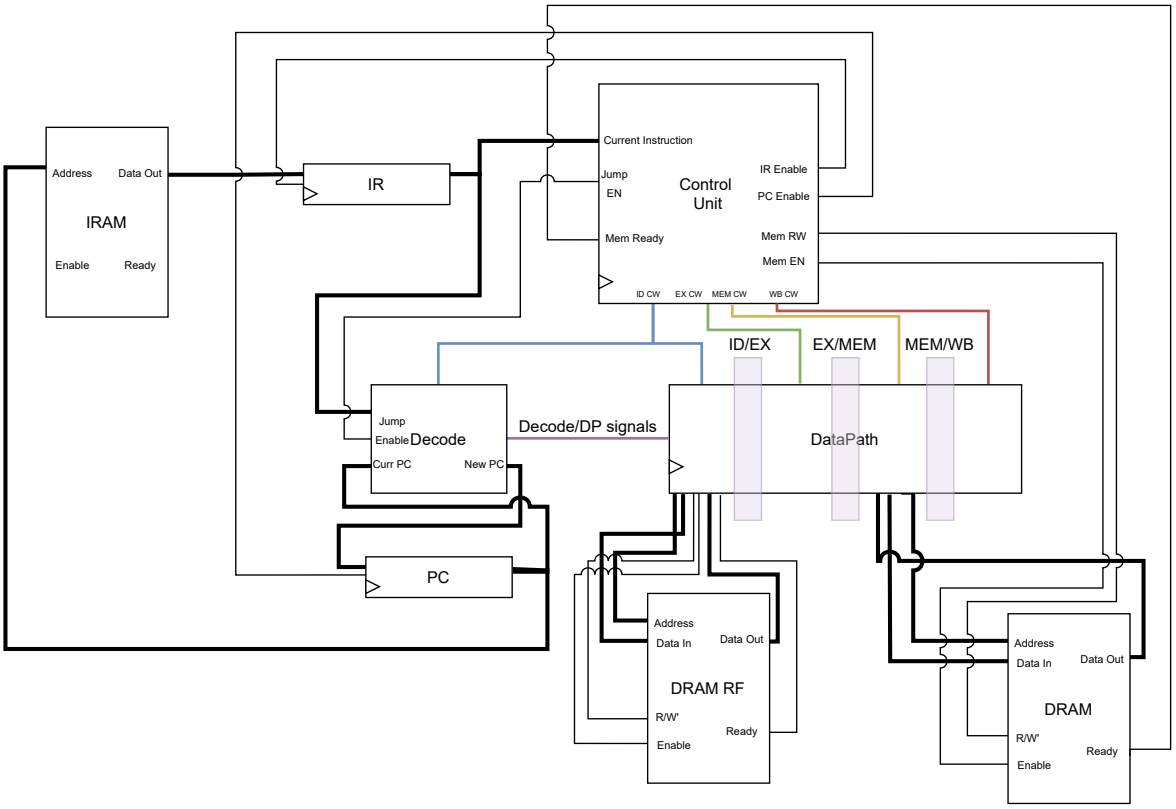


Figure 2.1: Schematic of the DLX

2.2 Pipeline Stages

To exploit the performance of the DLX processor, a five pipeline stage is implemented. This is done to reduce the overall delay, and different instructions can be processed in various stages simultaneously.

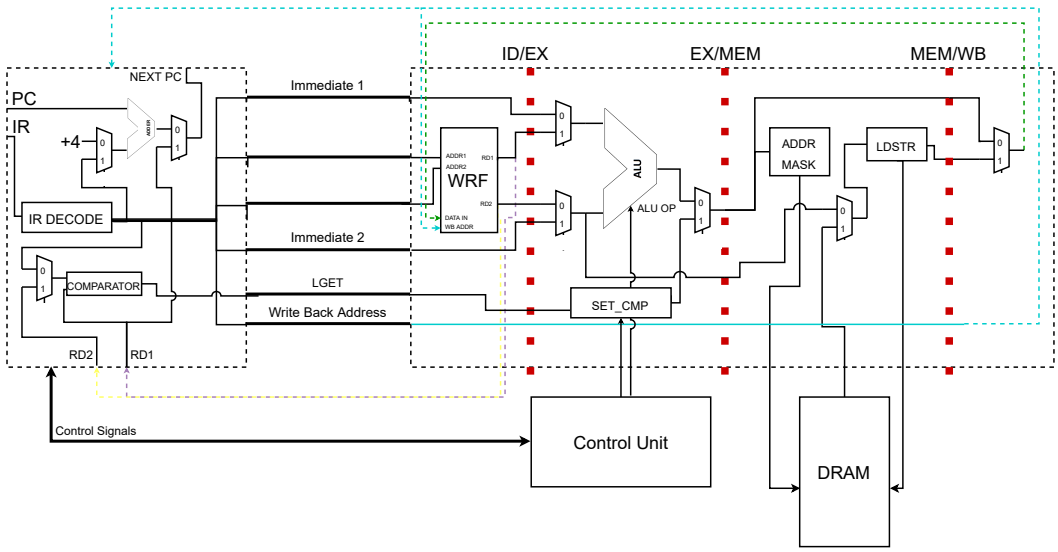


Figure 2.2: Schematic of the DLX Pipeline Structure

As shown in figure 2.2, the five pipeline stages are:

1. *Fetch*: not shown in the figure, it keeps track of the current instruction (IR) and the address of the next instruction to be fetched (PC).
2. *Decode*: it has the role of taking as input the current instruction (IR) and decoding it by extracting all the fields and dispatch them to the correct ports (Immediate 1, RS1, RS2, WB ADDR, Immediate 2). Furthermore, it's in charge of comparing values (from the Register File or an immediate) and computing the next PC given the actual PC. Last but not least, it keeps track of all the instructions in execution inside the pipeline and signals possible Data Hazard to the Control unit. By design choice, the Branch/Jump instructions are executed here instead of the Execute Stage, to lose only one clock cycle after a taken jump ideally (because of the wrong fetched instruction).
3. *Execute*: it takes the data from the Decode stage. Then, it does some computation on them, like adding them, multiply them, compute a shift and so on (refer to Table 5.1 for available ALU operations), or outputs to the next stage the result of the comparison made in the Decode stage.
4. *Memory*: given the Address computed by the ALU during the previous stage and an optional data, it's capable of doing a Load from the DRAM or a Store towards the DRAM. However, not all operations use this stage, so there is a bypass connection from the output of the EX stage towards the WB stage.
5. *Write Back*: the last stage of the pipeline selects the output from the ALU or the Memory (after a load operation) and gives it input to the Register File. Meanwhile, the destination address has been propagated through each stage and is now ready to be used by the Register File to store the final result. The Hazard Control Unit inside the Decode Stage intercepts this Write Back operation and knows that new instructions requiring this Data can now be executed.

2.3 Control Unit

The Control Unit is one of the essential parts of the DLX processor. Its role is to orchestrate all the jobs through the processor's pipeline and manage dangerous situations.

Its interface is made of input signals (status signals from other components) and of control signals (towards other components), as described in Table 2.1 and Table 2.2.

Signal Name	**Description**
IR_IN	Fetched Instruction, output from the IR Register
HAZARD_SIG	The Decode Stage is signaling a Hazard situation
BUSY_WINDOW	The Current RF Window has some registers in use
SPILL	The Register File has started a push operations towards the memory
FILL	The Register File has started a pop operations from the memory
IRAM_READY	The Instruction Memory has a data ready as output
LGET	Comparison status from the comparator inside the Decode Unit
DRAM_READY	Indicates if the DRAM is ready or not

Table 2.1: Input signals towards the Control Unit

Signal Name	Sign	**Description**
PIPLIN_IF_EN	F	IF Stage enable: enable of IR register
IF_STALL	S	IF Stage stall: a NOP is insterted in the IR register
PC_EN	P	Enable of the PC register
JUMP_EN	J	JUMP must occur: Decode Stage will compute the new PC
CALL	L	The Register File starts the context switch in a new window
RET	E	The Register File must restore the previous window
SEL_CMPB	P	Reg or Imm field as comparator input in the Decode Stage
UNSIGNED_ID	U	All arithmetic units must consider operands as unsigned
NPC_SEL	N	Selects new PC (Adder or REG content)
HAZARD_TABLE_WR1	H	Enables log of the current instruction in the hazard control
RF_RD1_EN	1	Enables the Read Port 1 in the RF
RF_RD2_EN	2	Enables the Read Port 2 in the RF
PIPLIN_ID_EN	D	ID Stage enable: enables all the ID pipeline REGs
PIPLIN_EX_EN	X	EX Stage enable: enables all the EX pipeline REGs
MUXA_SEL	A	Input A of ALU: Immediate 1 or RD1
MUXB_SEL	B	Input B of ALU: Immediate 2 or RD2
SEL_ALU_SETCMP	T	ALU or SET Comparator as EX stage output
ALU_OPCODE	`ALU`	ALU operation to be done
SEL_LGET	`SEL`	SET Comparator operation to be done
DRAM_WE	W	Enables the Write Operation in the DRAM
DRAM_RE	R	Enables the Read Operation in the DRAM
DRAM_ME		Enable signal for the DRAM
DATA_SIZE	DS	Indicates the data of transfer towards/from the DRAM
UNSIG_SIGN_N	U	Same as *UNSIGNED_ID* but 2 clock cycles delayed
PIPLIN_MEM_EN	M	MEM Stage enable: enables all the MEM pipeline REGs
WB_MUX_SEL	C	Selects MEM output or ALU out as content to write back
PIPLIN_WB_EN	K	WB Stage enable: enables the Write signal in the RF

Table 2.2: Output signals towards the Control Unit

2.3.1 Control Words

Going deeper into the Control Unit, it's possible to look at the generated Control Words for each Instruction. Each bit is referred to a Control Signal as indicated in Table 2.2, *Sign* column, while a blank bit means that it's dynamically assigned.

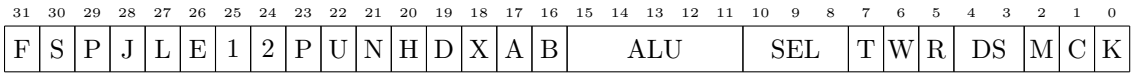


Figure 2.3: Control Word bits organization

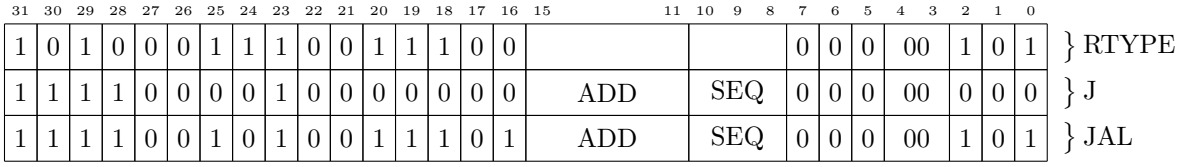


Figure 2.4: Control Word for each instruction

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1		0	0	1	1	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} BEQZ
1	0	1		0	0	1	1	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} BNEZ
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SEQ			0	0	0	00		1	0	1	} ADDI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SEQ			0	0	0	00		1	0	1	} ADDUI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	SUB		SEQ			0	0	0	00		1	0	1	} SUBI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	SUB		SEQ			0	0	0	00		1	0	1	} SUBUI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	AND		SEQ			0	0	0	00		1	0	1	} ANDI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	OR		SEQ			0	0	0	00		1	0	1	} ORI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	XOR		SEQ			0	0	0	00		1	0	1	} XORI
1	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	SLL		SEQ			0	0	0	00		1	0	1	} LHI
1	1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} JR
1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	ADD		SEQ			0	0	0	00		1	0	1	} JALR
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	SLL		SEQ			0	0	0	00		1	0	1	} SLLI
1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} NOP
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	SRL		SEQ			0	0	0	00		1	0	1	} SRLI
1	0	1	0	0	0	1	0	1	1	0	1	1	1	0	1	SRA		SEQ			0	0	0	00		1	0	1	} SRAI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SEQ			1	0	0	00		1	0	1	} SEQI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SNE			1	0	0	00		1	0	1	} SNEI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SLT			1	0	0	00		1	0	1	} SLTI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SGT			1	0	0	00		1	0	1	} SGTI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SLE			1	0	0	00		1	0	1	} SLEI
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SGE			1	0	0	00		1	0	1	} SGEI
1	1	1	1	0	0	1	0	1	0	0	1	1	1	0	1	ADD		SEQ			0	0	0	00		1	0	1	} CALL
1	1	1	1	0	1	1	0	0	0	1	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} RET
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SEQ			0	0	1	10		1	1	1	} LB
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SEQ			0	0	1	01		1	1	1	} LH
1	0	1	0	0	0	1	0	0	0	0	1	1	1	0	1	ADD		SEQ			0	0	1	00		1	1	1	} LW
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SEQ			0	0	1	10		1	1	1	} LBU
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SEQ			0	0	1	01		1	1	1	} LHU
1	0	1	0	0	0	1	1	0	0	0	0	1	1	0	1	ADD		SEQ			0	1	0	10		1	0	0	} SB
1	0	1	0	0	0	1	1	0	0	0	0	1	1	0	1	ADD		SEQ			0	1	0	01		1	0	0	} SH
1	0	1	0	0	0	1	1	0	0	0	0	1	1	0	1	ADD		SEQ			0	1	0	00		1	0	0	} SW
1	0	1	0	0	0	1	0	1	0	0	1	1	1	0	1	ADD		SEQ			0	0	0	00		1	0	1	} TKTM
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SLT			1	0	0	00		1	0	1	} SLTUI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SGT			1	0	0	00		1	0	1	} SGTUI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SLE			1	0	0	00		1	0	1	} SLEUI
1	0	1	0	0	0	1	0	0	1	0	1	1	1	0	1	ADD		SGE			1	0	0	00		1	0	1	} SGEUI
1	0	1		0	0	1	1	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} BGT
1	0	1		0	0	1	1	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} BGE
1	0	1		0	0	1	1	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} BLT
1	0	1		0	0	1	1	1	0	0	0	0	0	0	0	ADD		SEQ			0	0	0	00		0	0	0	} BLE

Figure 2.5: Control Word for each instruction

2.3.2 Internal organization of the Control Unit

The Control Unit internally is organized mainly with a stage that converts the actual Instruction into a Control Word. Then this control word is propagated to support all the pipeline stages, as shown in Figure 2.6.

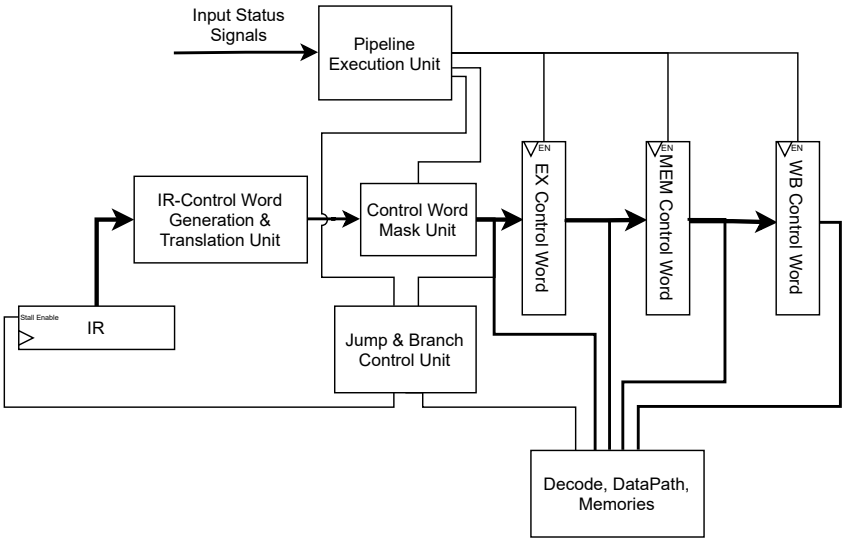


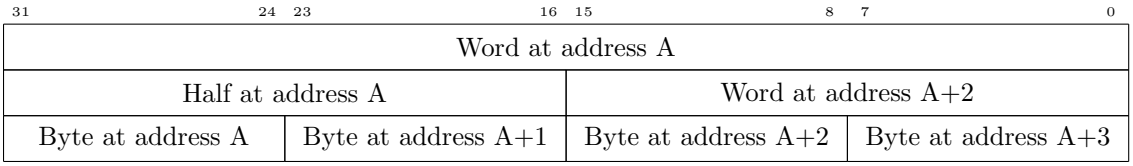
Figure 2.6: Schematic of the DLX Control Unit

It is made mainly by the following components:

- IR-Control Word Generation & Translation Unit: its role is to take the instruction given as input and outputs a valid Control Word related to that instruction.
- Pipeline Execution Unit: it's in charge of controlling the Pipeline Flow and stop it in case of needs and/or masking the actual Control Word to prevent its propagation through the chain in case of Hazards.
- Control Word Mask Unit: when the *Pipeline Execution Unit* signals to mask the control word, this unit will do it by disabling all the bits in the CW that refer to the stage enabling signals.
- Jump & Branch Control unit: given the Control Word and other internal control signals, its job is to manage (by enabling or disabling them) signals like *CALL*, *RET*, *JUMP_EN* and *IF_STALL*.

2.4 Memory Interface

The DLX processor has a Harvard architecture, with two 32-bit data buses carrying instructions and data respectively. Only load and store instructions can access data from Data Memory. The data are stored in memory in a Big-Endian format.



The third RAM required, the one for the register file, is not under the user's direct access. It's managed as a Stack memory by the Register File for automatic subroutines management, and it has a dedicated interface. It can be merged with the Data Memory with external logic.

All the subsequent paragraphs are referred in the same way to all the three memory types.

2.4.1 Signals and Timing

The signals in the DLX processor bus interface can be grouped into tree categories:

- Address class signals
- Memory Request signals
- Data class signals

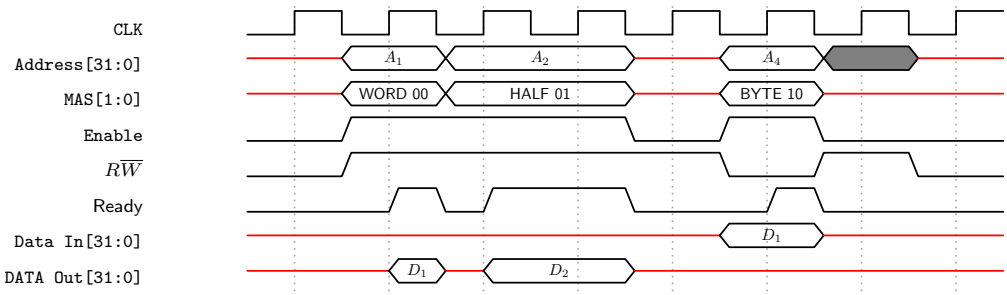
The *Address class signals* are:

- A[31:0]
- DATA_SIZE[1:0] *or MAS[1:0]*

The *Memory Request signals* are:

- Enable
- $R\overline{W}$
- Ready

Moreover, all the memories connected must agree on a specific protocol both for writing and reading operations. The most important thing to take under consideration is the *Ready* signal: it must be high only when the operation is completed. For example, after a data read, the *Ready* stays at 1 only when the data is valid. If, meanwhile, the address changes, the *Ready* signal must go off.



If the Memory in use can't accomplish this timing, an external *Memory Control Unit* must be placed between the CPU and the Memory.

2.4.2 Memory Addressing

A[31:0] is the 32-bit address bus that specifies the address for the transfer. All addresses are byte addresses, so a burst of word accesses results in the address bus incrementing by four for each cycle.

The address bus provides 4GB of linear addressing space, and this can be used externally in different manners, like in an SoC with memory-mapped peripherals that share the same address space.

When word access is signaled, the memory system ignores the bottom two bits, A[1:0], and when halfword access is signaled, the memory system ignores the bottom bit, A[0]. However, the core already masks the two LSBs when needed.

2.4.3 Memory Data Size: the MAS[1:0] signal

The *MAS[1:0]* bus encodes the size of the transfer. The DLX processor can transfer word, halfword, and byte quantities and the processor indicates the size of the transfer through this signal.

The *MAS[1:0]* signal is supposed to be used in combination with *A[1:0]* to know precisely which byte is taken under consideration during an operation.

2.4.4 Memory Read

When an half-word or byte read is performed, a 32-bit memory system can return the complete 32-bit word for simplicity, and the processor extracts the valid half-word or byte field from it as shown in Table 2.3. Other byte lanes will be ignored. For 8 and 16 bit memories, the data must be placed on the right byte lanes in the data bus.

DATA_SIZE[1:0]	A[1:0]	D[31:0]	DLX Register
00 WORD	00	`0xAABBCCDD`	`0xAABBCCDD`
01 HALF	00	`0xAABB----`	`0x0000AABB`
01 HALF	10	`0x----CCDD`	`0x0000CCDD`
10 BYTE	00	`0xAA------`	`0x000000AA`
10 BYTE	01	`0x--BB----`	`0x000000BB`
10 BYTE	10	`0x----CC--`	`0x000000CC`
10 BYTE	11	`0x------DD`	`0x000000DD`

Table 2.3: How data are read by the CPU in different MAS configurations

2.4.5 Memory Write

Instead, when the DLX processor performs a byte or half-word write, the data being written is replicated across the data bus, as shown in Figure 2.7. The memory system can use the most convenient copy of the data.

A writable memory system must be capable of performing a write to any single byte in the memory system. This is required for the correct working of the DLX.

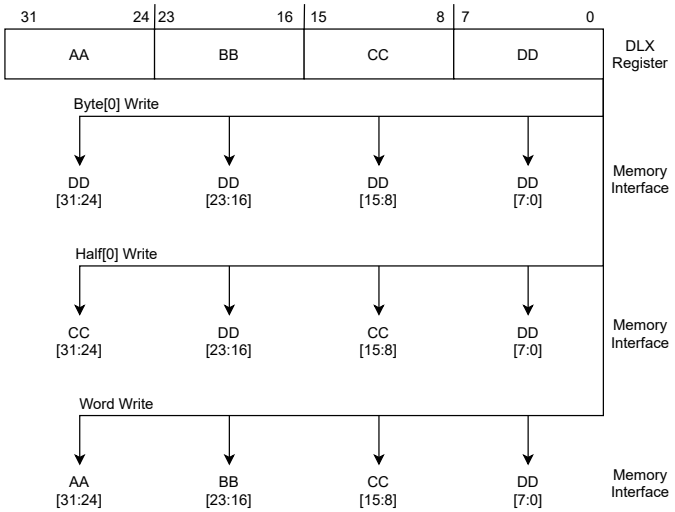


Figure 2.7: Data Write Replication

2.5 Instruction Set

The Instruction Set supported by the DLX is made of different instructions, in particular regarding Integer operations. Instructions are on 32 bit and are grouped in 3 different types:

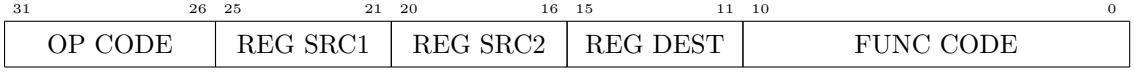


Figure 2.8: R-Type

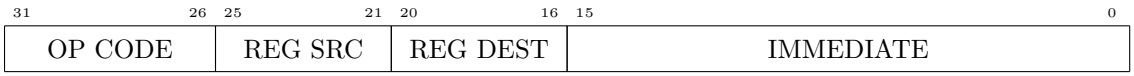


Figure 2.9: I-Type

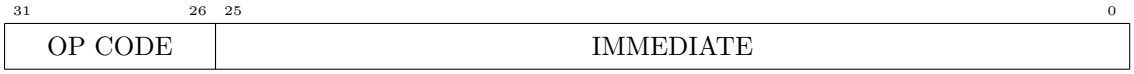


Figure 2.10: J-Type

In particular, all of them have in common the *OP CODE* field, used to identify the instruction.

2.5.1 R-Type Instructions

R-Type Instructions are called in this way because they operate only between registers. They all have in common the *OP CODE* equal to `0b00000` and each instruction is differentiated from another one thanks to the *FUNC CODE* field, as shown in Table 2.4 and Table 2.5.

MNEMONIC	FUNC CODE	EXAMPLE	OPERATION	DESCRIPTION
SLL	`0x04`	`SLL R3, R2, R1`	$R3 \leftarrow R2 \ll R1$	Logical shift left
SRL	`0x06`	`SRL R3, R2, R1`	$R3 \leftarrow R2 \gg R1$	Logical shift right
SRA	`0x07`	`SRA R3, R2, R1`	$R3 \leftarrow R2 \gg R1$	Arithmetic shift right
ROR	`0x08`	`ROR R3, R2, R1`	$R3 \leftarrow R2 \circlearrowright R1$	Right rotation
ROL	`0x09`	`ROL R3, R2, R1`	$R3 \leftarrow R2 \circlearrowleft R1$	Left rotation
MUL	`0x0E`	`MUL R3, R2, R1`	$R3 \leftarrow R2 * R1$	Integer multiplication
ADD	`0x20`	`ADD R3, R2, R1`	$R3 \leftarrow R2 + R1$	Integer Add
ADDU	`0x21`	`ADDU R3, R2, R1`	$R3 \leftarrow R2 + R1$	Integer Add
SUB	`0x22`	`SUB R3, R2, R1`	$R3 \leftarrow R2 - R1$	Integer subtraction
SUBU	`0x23`	`SUBU R3, R2, R1`	$R3 \leftarrow R2 - R1$	Integer subtraction
AND	`0x24`	`AND R3, R2, R1`	$R3 \leftarrow R2 \& R1$	Bitwise AND
OR	`0x25`	`OR R3, R2, R1`	$R3 \leftarrow R2 \mid R1$	Bitwise OR
XOR	`0x26`	`XOR R3, R2, R1`	$R3 \leftarrow R2 \oplus R1$	Bitwise XOR

Table 2.4: R-TYPE Instructions: logical and arithmetic

MNEMONIC	FUNC CODE	EXAMPLE	OPERATION	DESCRIPTION
SEQ	0x28	SEQ R3, R2, R1	$R3 \leftarrow R2 == R1$	1 if R2 equal R1
SNE	0x29	SNE R3, R2, R1	$R3 \leftarrow R2! = R1$	1 if R2 not equal R1
SLT	0x30	SLT R3, R2, R1	$R3 \leftarrow R2 < R1$	1 if R2 less R1
SGT	0x31	SGT R3, R2, R1	$R3 \leftarrow R2 > R1$	1 if R2 great R1
SLE	0x32	SLE R3, R2, R1	$R3 \leftarrow R2 \leq R1$	1 if R2 less or equal R1
SGE	0x33	SGE R3, R2, R1	$R3 \leftarrow R2 \geq R1$	1 if R2 great or equal R1
SLTU	0x3A	SLTU R3, R2, R1	$R3 \leftarrow R2 < R1$	1 if R2 unsigned less R1
SGTU	0x3B	SGTU R3, R2, R1	$R3 \leftarrow R2 > R1$	1 if R2 uns great R1
SLEU	0x3C	SLEU R3, R2, R1	$R3 \leftarrow R2 \leq R1$	1 if R2 uns less/eq R1
SGEU	0x3D	SGEU R3, R2, R1	$R3 \leftarrow R2 \geq R1$	1 if R2 uns great/eq R1

Table 2.5: R-TYPE Instructions: test instructions

2.5.2 Pseudo R-Type Instructions

These are the R-Type like operations: they work only with registers but have their OPCODE.

MNEMONIC	OP CODE	EXAMPLE	OPERATION	DESCRIPTION
JR	0x12	JR R1	$PC \leftarrow PC + R1$	PC += R31
JALR	0x13	JALR R1	$R31 \leftarrow PC$ $PC \leftarrow PC + R1$	R31 = actual PC PC += R1
RET	0x1F	RET	$PC \leftarrow R31$	PC = R31 Context Switch begins
NOP	0x15	NOP		Does exactly nothing
TICKTMR	0x34	TICKTMR R16	$R16 \leftarrow TICKTIMER$	Get Tick Timer value

Table 2.6: Pseudo R-TYPE Instructions: branch, jump and mix

2.5.3 J-Type Instructions

J-Type Instructions are groups of instructions made for code jump. Thanks to the structure of these instructions, long jumps can be executed (i.e. the relative address can be very big).

MNEMONIC	OP CODE	EXAMPLE	OPERATION	DESCRIPTION
J	0x02	J LABEL	$PC \leftarrow PC + LABEL$	PC += rel label
JAL	0x03	JAL LABEL	$R31 \leftarrow PC$ $PC \leftarrow PC + LABEL$	R31 = actual PC PC += rel label
CALL	0x1E	CALL LABEL	$R31 \leftarrow PC$ $PC \leftarrow PC + LABEL$	R31 = actual PC PC += rel label Context Switch begins

Table 2.7: J-TYPE Instructions

2.5.4 I-Type Instructions

I-Type instructions support immediate operands, as shown in Table 2.8.

MNEMONIC	OP CODE	EXAMPLE	OPERATION	DESCRIPTION
ADDI	0x08	`ADDI R3, R0, #2`	$R3 \leftarrow R0 + 2$	Integer Add
ADDUI	0x09	`ADDUI R3, R2, #2`	$R3 \leftarrow R2 + 2$	Integer Add
SUBI	0x0A	`SUBI R3, R2, R1`	$R3 \leftarrow R2 - R1$	Integer subtraction
SUBUI	0x0B	`SUBUI R3, R2, R1`	$R3 \leftarrow R2 - R1$	Integer subtraction
ANDI	0x0C	`ANDI R3, R2, R1`	$R3 \leftarrow R2 \& R1$	Bitwise AND
ORI	0x0D	`ORI R3, R2, R1`	$R3 \leftarrow R2 \mid R1$	Bitwise OR
XORI	0x0E	`XORI R3, R2, R1`	$R3 \leftarrow R2 \oplus R1$	Bitwise XOR
LHI	0x0F	`LHI R3, #0xABCD`	$R3 \leftarrow$ 0xABCD0000	Load Upper Half
SLLI	0x14	`SLLI R3, R2, #1`	$R3 \leftarrow R2 \ll 1$	Logical shift left
SRLI	0x16	`SRLI R3, R2, #1`	$R3 \leftarrow R2 \gg 1$	Logical shift right
SRAI	0x17	`SRAI R3, R2, #1`	$R3 \leftarrow R2 \gg 1$	Arithmetic shift right

Table 2.8: I-TYPE Instructions: logical and arithmetic

MNEMONIC	OP CODE	EXAMPLE	OPERATION	DESCRIPTION
SEQI	0x18	`SEQI R3, R2, #1`	$R3 \leftarrow R2 == 1$	1 if R2 equal 1
SNEI	0x19	`SNEI R3, R2, #1`	$R3 \leftarrow R2! = 1$	1 if R2 not equal 1
SLTI	0x1A	`SLTI R3, R2, #1`	$R3 \leftarrow R2 < 1$	1 if R2 less 1
SGTI	0x1B	`SGTI R3, R2, #1`	$R3 \leftarrow R2 > 1$	1 if R2 great 1
SLEI	0x1C	`SLEI R3, R2, #1`	$R3 \leftarrow R2 \leq 1$	1 if R2 less or equal 1
SGEI	0x1D	`SGEI R3, R2, #1`	$R3 \leftarrow R2 \geq 1$	1 if R2 great or equal 1
SLTUI	0x3A	`SLTUI R3, R2, #1`	$R3 \leftarrow R2 < 1$	1 if R2 unsigned less 1
SGTUI	0x3B	`SGTUI R3, R2, #1`	$R3 \leftarrow R2 > 1$	1 if R2 uns great 1
SLEUI	0x3C	`SLEUI R3, R2, #1`	$R3 \leftarrow R2 \leq 1$	1 if R2 uns less/eq 1
SGEUI	0x3D	`SGEUI R3, R2, #1`	$R3 \leftarrow R2 \geq 1$	1 if R2 uns great/eq 1

Table 2.9: I-TYPE Instructions: test instructions

MNEMONIC	OP CODE	EXAMPLE	OPERATION	DESCRIPTION
LB	0x20	`LB R1, 40(R2)`	$R1 \leftarrow MEM[R2 + 40]$	Load Byte / Sign Ext
LH	0x21	`LH R1, 40(R2)`	$R1 \leftarrow MEM[R2 + 40]$	Load Half / Sign Ext
LW	0x23	`LW R1, 40(R2)`	$R1 \leftarrow MEM[R2 + 40]$	Load Word / Sign Ext
LBU	0x24	`LBU R1, 40(R2)`	$R1 \leftarrow MEM[R2 + 40]$	Load Byte
LHU	0x25	`LHU R1, 40(R2)`	$R1 \leftarrow MEM[R2 + 40]$	Load Half
SB	0x28	`SB 40(R2), R1`	$MEM[R2 + 40] \leftarrow R1$	Store Byte
SH	0x29	`SH 40(R2), R1`	$MEM[R2 + 40] \leftarrow R1$	Store Half
SW	0x2B	`SW 40(R2), R1`	$MEM[R2 + 40] \leftarrow R1$	Store Word

Table 2.10: I-TYPE Instructions: Load & Store

MNEMONIC	OP CODE	EXAMPLE	OPERATION	DESCRIPTION
BEQZ	0x04	BEQZ R1, LABEL	$PC \leftarrow PC + LABEL$	$R1 = 0$? PC += label
BNEZ	0x05	BNEZ R1, LABEL	$PC \leftarrow PC + LABEL$	$R1 \neq 0$? PC += label
BGT	0x30	BGT R1, R2, LABEL	$PC \leftarrow PC + LABEL$	$R1 > R2$? PC += label
BGE	0x31	BGE R1, R2, LABEL	$PC \leftarrow PC + LABEL$	$R1 \geq R2$? PC += label
BLT	0x32	BLT R1, R2, LABEL	$PC \leftarrow PC + LABEL$	$R1 < R2$? PC += label
BLE	0x33	BLE R1, R2, LABEL	$PC \leftarrow PC + LABEL$	$R1 \leq R2$? PC += label

Table 2.11: I-TYPE Instructions: Conditional Branch Instructions

CHAPTER 3

Fetch Stage

The first stage of the DLX pipeline is the Fetch Stage, which has been included directly inside the `DLX`. It is used to manage the current Program Counter `PC` and send it out to the memory to fetch the instruction from it and save the instruction into the Instruction Register `IR`. At each clock cycle, the new `PC` is computed by summing 4, since the instructions are on 32 bits and the memory is organized in words of 1 byte. This can be summarized in the following way:

$$IR \leftarrow MEM[PC]$$
$$NPC \leftarrow PC + 4$$

As will be explained in the Jump and Branch Management section (refer to 3.3), to manage jumps and conditional branches, some additional hardware is necessary in order to compute the new address.

3.1 Instruction Register

The Instruction Register (`IR`) is, in this case, a 32 bits register that is used to store the instruction (that is in fact on 32 bits) that comes from the Instruction RAM.

During the normal operation, this register is updated with the new instruction coming from the IRAM, but this is not always true. There are two cases in which the register is not updated:

1. When `i_IR_LATCH_EN` is at 0, in fact, this signal is used to avoid updating the value inside the `IR`. During the instruction execution, it could be that the DRAM is not ready, a signal `i_DRAM_NOTREADY` is checked inside the Control Unit and in case of '1' the `i_IR_LATCH_EN` is put at 0. In this way, everything is stalled so the CW in the pipeline and the Instruction Register will remain the same until the DRAM will become ready again.

2. When the reset signal is asserted or when the `i_IR_STALL` is at one (this signal is at one when the pipeline must continue its execution but a NOP is executed for resolving a hazard), a "bubble" is added to the pipeline. So, the Instruction Register will contain the `x"54000000"` value. This is used in order to manage the jump and branch instructions. A more detailed description is available at section 3.3.

3.2 Program Counter

As anticipated before, the Program Counter (`PC`) is the address used to fetch the instruction from the IRAM and it is on 32 bits. Thus it can address up to 4GB.

At each clock cycle, the current Program Counter is updated with the NPC computed in the Decode stage in order to point to the next instruction. This allows managing both the standard execution, by adding 4 to the current PC, subtracting/adding an offset, or set it to an immediate. Only in case of reset, it is set to 0 otherwise, it will be set to the value of the new Program Counter, computed in the Decode stage. In a specular way, from what is done for the Instruction Register, the PC is updated only when the i_PC_LATCH_EN is '1'.

3.3 Jump and Branch Management

Jumps and branches are crucial instructions to build a program, but their management from the point of view of the pipeline is not straightforward. Therefore, the Fetch Stage must be able to manage three additional conditions:

1. i_PC_LATCH_EN: this signal is used to avoid updating the PC with the new one coming from the Decode Stage. The update is inhibited when the instruction is a jump or when i_IR_LATCH_EN is '0'.

2. i_IR_LATCH_EN: it is used to avoid updating the Instruction Register. This happens when:

   - A Data Hazard occurs
   - A CALL instruction is executed and the CPU has to wait for a spill operation to be completed
   - A RET instruction is executed and the CPU has to wait for a fill operation to be completed

   It's important to say that, when the IR is not updated, the PC is not updated too. In all the three cases, all the pipeline registers are disabled (PC, IR, ID, EX, MEM, WB) for the instruction; so, the stalled instruction is waiting in the Decode stage of the pipeline.

3. i_IR_STALL: a NOP is inserted in the pipeline in order to manage the jump and the branch instructions. This is due to the control hazards; when a branch is executed, it may or may not change the PC to something other than its current value plus 4. As it will be described more precisely in the Decode Stage section, the computation of the new PC, both for branch and jump, is done there. For this reason, it is updated only at the end of the Decode Stage, and, if not blocked, a wrong instruction is fetched.

   The solution this DLX implements is to add a NOP not to execute a wrong instruction.

CHAPTER 4

Decode Stage

The Decode Stage is the first DataPath stage, and logically, it's divided between the DataPath itself and a separated unit called "Decode". The Decode Stage is mainly used to perform the Instruction Decode identification of instruction type given its *OP_CODE* and then the dispatch of the operands contained in the instruction towards the DataPath. Meanwhile, other side operations, like computation of the new PC (given a jump or not), data hazard detection and comparisons, are performed.

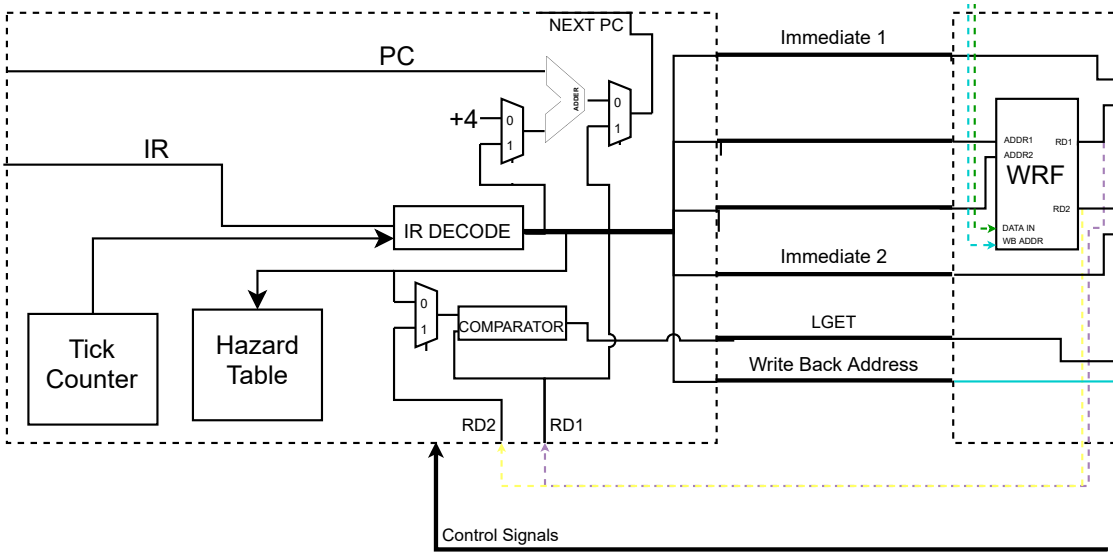


Figure 4.1: Simplified schematic of the decode

4.1 Instruction Decode

The most important operation is the Instruction Decode. Here, after the identification of the instruction type, its fields are extracted according to the general instruction structure, as described in Figure 2.8, Figure 2.9 and Figure 2.10.

The Decode is based on the logic shown in Table 4.1. The table describes only the signals that go towards the DataPath and not the internal ones.

Instruction	INP1	INP2	RS1	RS2	WS	Note
RTYPE	0	0	IR[25:21]	IR[20:16]	IR[15:11]	
J	0	0	R0	R0	R0	
JAL (CALL)	0	PC	R0	R0	R31	
JALR	0	PC	IR[25:21]	R0	R31	
TICKTMR	0	*i_TICKTMR*	R0	R0	IR[25:21]	Destination reg on [25:21]
ITYPE	0	IR[15:0]	IR[25:21]	IR[20:16]	IR[20:16]	INP2 Sign Extension iff *UNSIGNED_ID*
LHI	IR[15:0]	16	IR[25:21]	IR[20:16]	IR[20:16]	No Sign Ext

Table 4.1: Instruction Decoding

For Jump or Branch instruction, the immediate field doesn't go beyond the decode stage itself. The immediate is a relative value to be summed to the actual PC in case of a taken jump. For Jump-like instructions (J, JAL, CALL) the field is 26 bits long (far jumps), while for other branch instructions the field is 16 bits long (near jumps). In both cases, the value is extended to 32 bit with its sign because the relative value can be both positive (forward jumps) and negative (backward jumps).

4.2 Register File and Windowing

The general structure of a register file is based on a decoder that takes the selection input (so the address of the desired register) and enables it (also using the enable signal). An input signal will contain the value to be written. On the other hand, a read signal is used to select among all the registers.

The DLX presented in this document has been enhanced in order to be able to manage subroutine transparently from the point of view of the user. For this reason, the DLX must be able to handle subroutines, so the context switching, which consists of saving the content of the registers to be restored once the procedure has been completed. The straightforward solution is to save into the memory all registers, but this is not feasible in terms of delay since, for 32 registers, 32 clock cycles are needed. In a pipeline, this corresponds to a long stall each time a procedure is called.

A windowed register allows reducing the overhead due to the context switch. The basic idea is to split the available registers in the physical register file into blocks, called *windows*. There are a limited amount of physical registers in the register file; for this reason, a finite number of windows are defined. Each window is assigned to a subroutine so that the procedure can write only on those registers. This is transparent from the point of view of the CPU, which sees all registers available. Thus, the physical register file has a wrapper around it with a logic and a Register Management Logic (RML) that allows performing a translation between the CPU requests and the corresponding window for the running procedure.

When the number of called procedures is larger than the number of available windows, the main memory is involved, and the oldest allocated window is swapped in. This happens only when there are no free windows in the register file so that the new one can be allocated. Once all the recursion chain has been unrolled, the swapped windows in the memory must be restored into the register file.

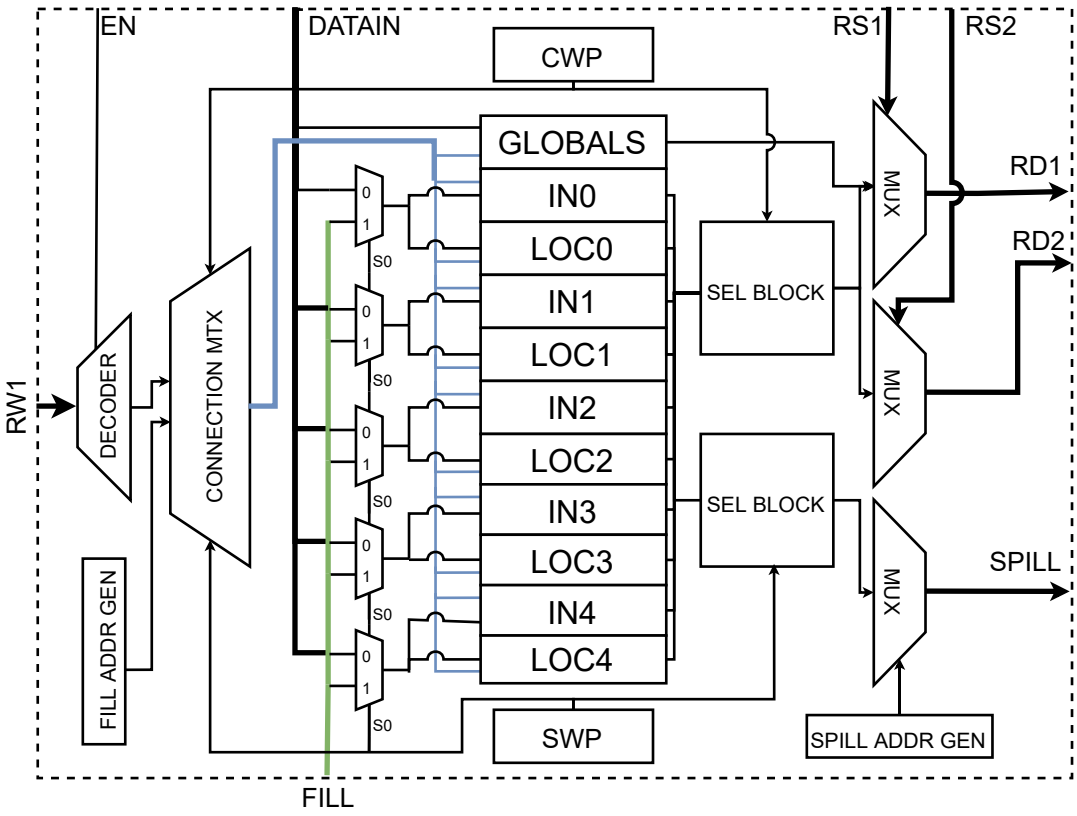


Figure 4.2: Schematic of the Windowing Register File

All windows, so each procedure, is made of 4 blocks of 8 registers each one:

1. **IN**: the first block is dedicated to the data inherited from the parent routine (previous OUT);
2. **LOCAL**: contains the registers that are dedicated only to the procedure;
3. **OUT**: is dedicated to the variables to be passed to the child routine, that is the IN of the next sub-procedure
4. **GLOBAL**: the last block is common to every window.

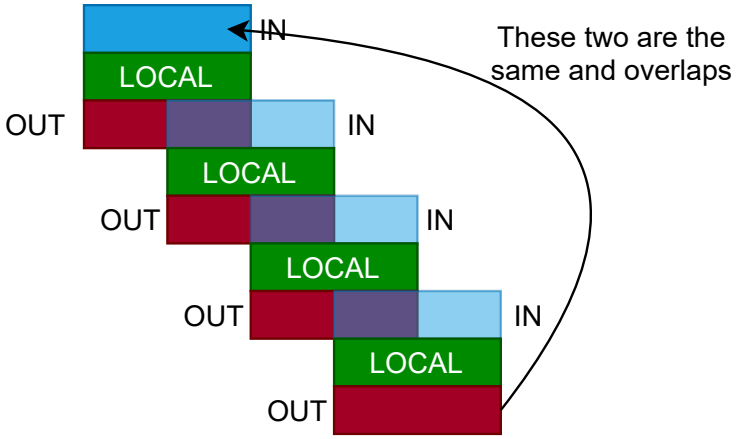


Figure 4.3: Logical organization of the Windowing Register File

When a procedure is called, the first LOCALS and OUT blocks are allocated from the physical file register and assigned to it (because IN is the OUT of the previous one).

By calling many nested procedures, at some point, there will be no free windows; for this reason, the oldest is de-allocated from the physical FR and swapped to the main memory. This operation is called **SPILL**. This is accomplished by using a support pointer, called **Saved Window Pointer SWP** that stores the pointer of the spilled data, exactly the end of the LOCALS block (only IN and LOCAL are spilled, the OUT block is not spilled because it is the IN of the next sub-procedure). In practice, it defines the position of the last free cell. Notice that this operation cannot be executed in one clock cycle: each register is spilled once at a clock cycle.

On the other hand, when the last procedure in the chain is finished, the others are unrolled; if some of them have been spilled, a **FILL** must be executed before the actual execution.

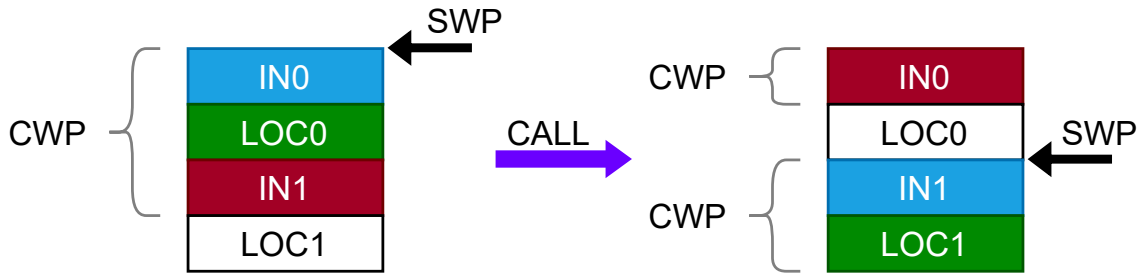


Figure 4.4: Logical Flow of CWP and SWP

It's important to notice that the implementation of the entire register file has been done in a complete structural way. It is composed of several components:

- **Decoder**: it is used to generate a single enable signal from an address on *NBIT_ADD* bits given by the CPU; in this way, a register is selected to perform a write.
- **Connection matrix**: this component allows to "highlight" the active window, the block IN, LOCAL and OUT will be the default destination when writing and reading;
- **Register file**: this block corresponds to the physical registers, composed by rows of registers;
- **Select block**: this block is used for reading: it is connected to all the registers and selects, using the current window pointer, all the registers that belong to the pointed window. A multiplexer then is used to read from a single register;
- **Address generator**: this block is only used when performing a FILL or a SPILL. It generates the address for the registers to be moved from/to the memory. The latter, in this situation, works exactly like a stack;
- **WRF Control Unit**: A FSM is used to manage and generate the memory addresses when performing a SPILL or a FILL.

Additional but less complex hardware has been used in order to manage the CWP and SWP. The complete diagram is available at Figure E.1.

4.2.1 Decoder

This block receives as input the *write address* on *NBIT_ADD* bits and outputs $2^{NBIT\_ADD} - 1$ bits. It has the utility of converting the address of the register at which the CPU needs to write into the relative register enable signal.

The idea is that if the input is $0b00010$ the output has to be $0b00000000000000000000000000000100$. If the input is decimal 2, it means that the CPU needs to write the third register of the *GLOBAL* block. In terms of enable signals, it translates into having the bit with index 2 at one.

The output is divided, as shown in Figure 4.5, in order to represent the group of bits. In particular:

$$\text{DECODED ADDRESS: } 0b \underbrace{00000000}_{\text{OUT}}\ \underbrace{00000000}_{\text{LOCAL}}\ \underbrace{00000000}_{\text{IN}}\ \underbrace{00000100}_{\text{GLOBAL}}$$

On the top of the schematic Figure 4.5 an AND logic gate between an *ENABLE* signal and the *WR* signal is used. If both *ENABLE* and *WR* are at 1, it means that the register is enabled.

The generated enable signal is called *Logical Enable Signal*. To generate the *Physical Enable Signal*, a translation is needed and it's performed by the *connection matrix*.

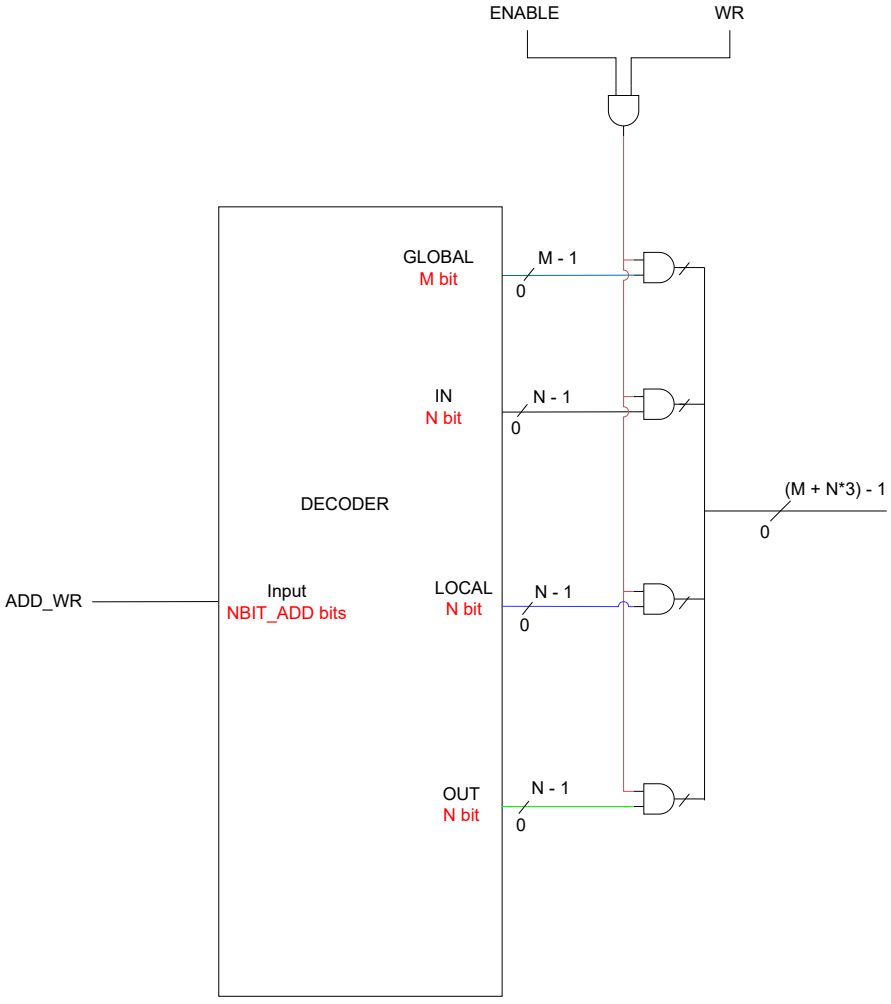


Figure 4.5: Schematic of the Decoder

4.2.2 Connection Matrix

With the previous block, all the logical enable signals are generated. They are related to a single window, so a block that puts together the required enabled register and the actual window is needed.

This block receives as inputs the signals coming from the decoder and the current window. The output is an enable signal that is a translation from the "logic" enable (the one referred to a virtual register, the ones that are seen by the CPU) and the physical one (within the actual window). This is related to the logic placed on the left in Figure 4.6.

There is a specific structure for each group of logical enable signals:

- GLOBAL ones: the global is the simplest because it is always the same despite the selected window.
- IN: for this group of registers, one of them is activated when "an IN enable signal is high AND the related window is active OR an OUT enable signal is high AND the previous related window is active".
- OUT: same reason as before: "an OUT enable signal is high AND the related window is active OR an IN enable signal is active and the next related window is active".
- LOCAL: for this block, a simple AND is used with the LOCAL enable coming from the decoder and the related window.

However, the connection matrix is used during a FILL operation too. It addresses the group of registers to be restored. For this purpose, it receives the SWP and FILL signal and the same reasoning is applied (group of logic in the middle in Figure 4.6).

For addressing during the FILL operation, an internal address signal is used, called addr_pop. This group of signals is generated by an address_generator unit that produces the addresses from the last to the first one. This happens because the memory is used like a STACK.

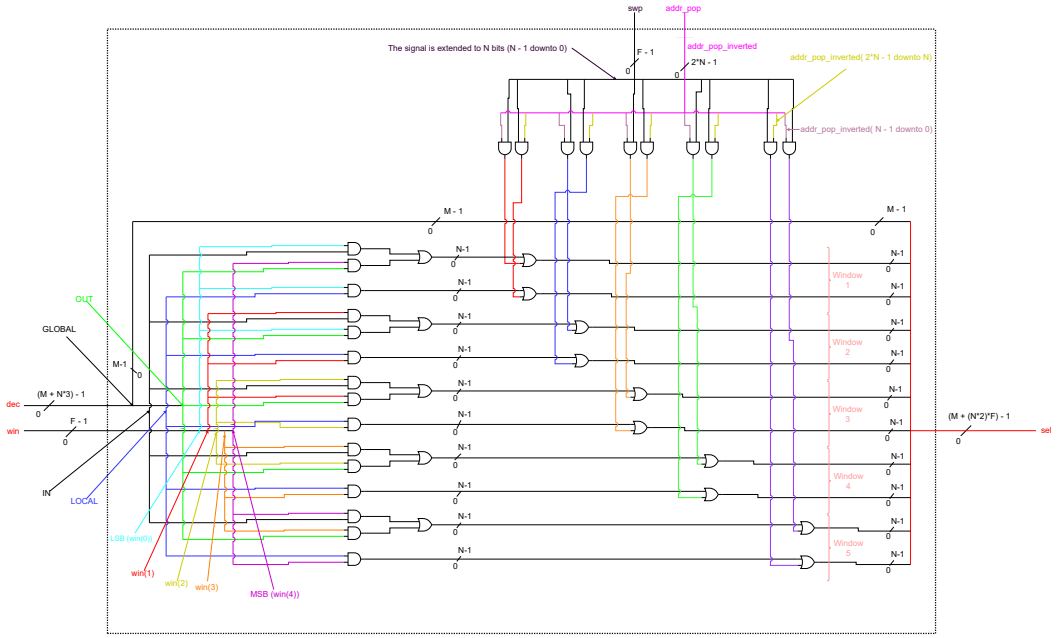


Figure 4.6: Connection matrix

4.2.3 Physical Register File

The next block is the Physical Register File, which is a sequence of registers. For writing operations, there are two sources of Data: the ones coming from the CPU that wants to write in a register and the ones coming from the memory during a FILL operation. In order to support this kind of access, a multiplexer is used for each window, so for each group of IN-LOCAL.

The multiplexer selection signal is the $SWP - 1$ because the SWP always points to the window that eventually needs to be spilled so the previous one is the one that has been spilled and so it's the right one to restore.

4.2.4 Select Block

The purpose of this unit is very simple and straightforward. It receives as input the current window and the output of all the registers, and it outputs the content of the IN, LOCAL and OUT of the current window. Its interface is shown in Figure 4.7.

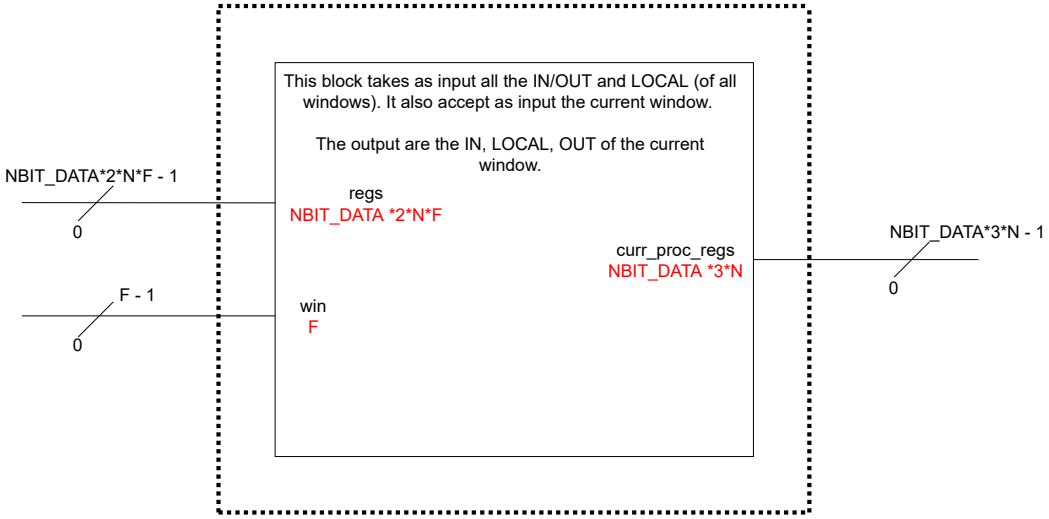


Figure 4.7: Interface of the select block

4.2.5 Output Selection Stage

This is the stage that decides the outputs RD1 and RD2 of the Register File. The design is shown in Figure 4.8.

This stage receives the IN, LOCAL, OUT of the current window, thanks to the select block, and the GLOBAL. The two addresses, ADD_RS1 and ADD_RS2, select the single register output through the multiplexer.

A latch is placed between the multiplexer output and the final output of the register file. When one of the two read port is disabled, the latch is in memory state so the output will remain stable.

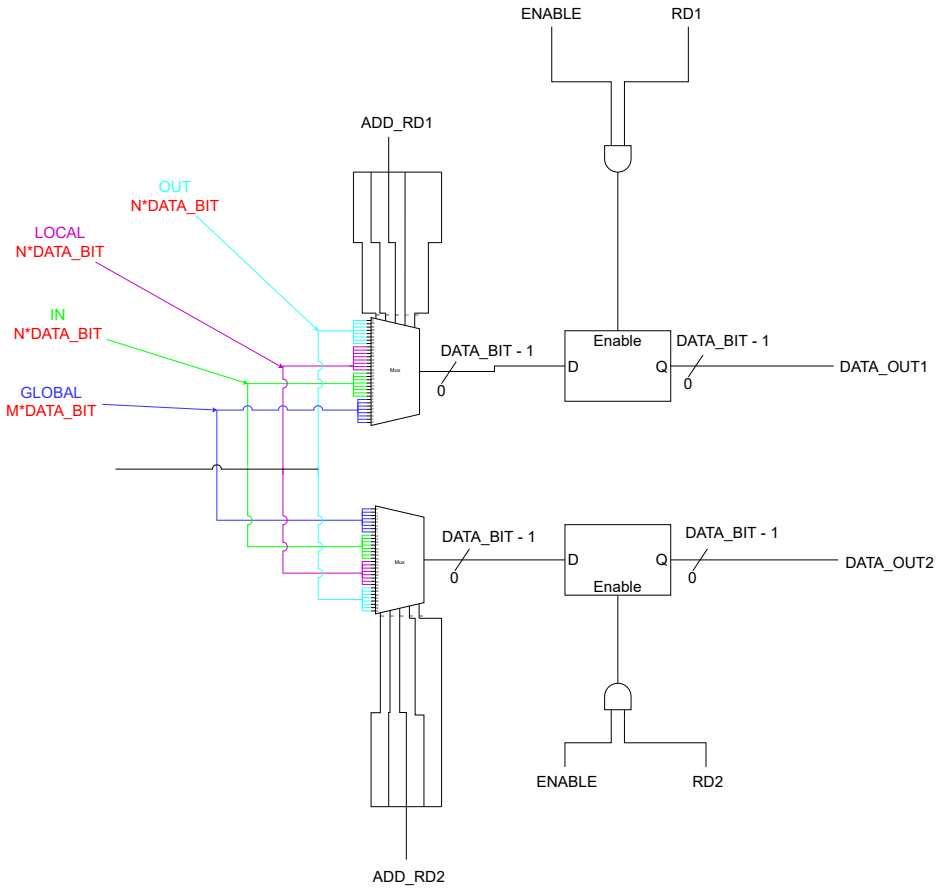


Figure 4.8: Design of the output selection

4.2.6 Next Window Calculator

This block is used to compute the next window, both for the current window and the saved window. The schematic is shown in Figure 4.9.

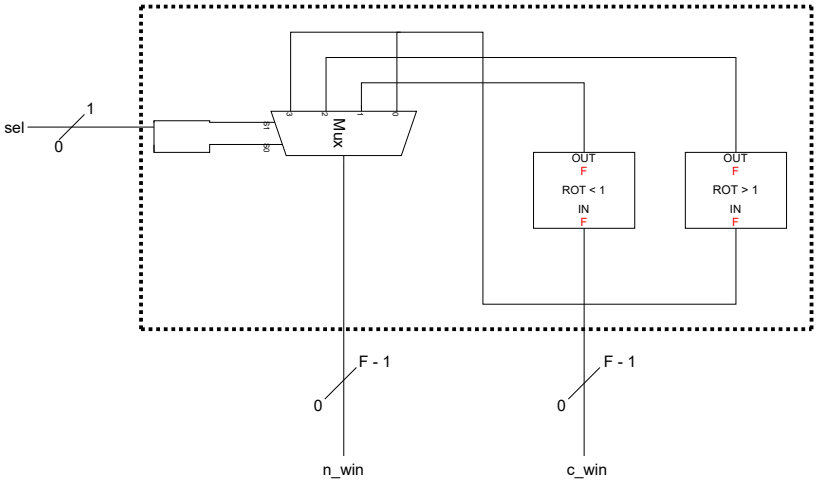


Figure 4.9: Design of the next window calculator

Inside this block there is a logic able to rotate right or left and a multiplexer that allows selecting the correct output based on what the circuit needs.

4.2.7 WRF Control Unit

An additional element is necessary in order to be able to manage the addresses for the SPILL and FILL procedure to and from memory. It has been implemented using a Moore FSM and consists of different phases; a starting one that set the memory address to 0 and when the FILL or SPILL input is '1' and the ram is ready, the address is decreased or increased by 4, accordingly to the operation. Refer to figure 4.10. It's important to say that, if the RAM is not ready the address is kept untouched, so that when the RAM returns in a ready state, the procedure can continue.

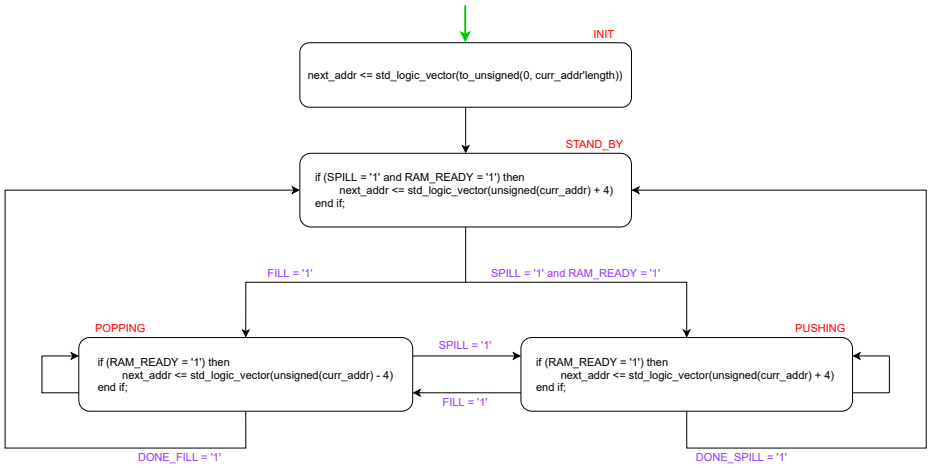


Figure 4.10: Moore FSM implementation for WRF CU

4.3 Hazard Control

During the execution of a pipeline, different problems or hazard situations may occur. Regarding Data Hazards, the most dangerous situation is the *Read After Write (RAW)*. So a solution is needed in order to detect these problems. Actually exists a unit inside the Decode Stage that is dedicated to this job called *Hazard Table*.

RAW Hazards occurs when an instruction wants to read from a register that is the destination register of a previous instruction still inside the pipeline so not completed yet. This means that the second instruction has to be stopped (i.e. a stall or a bubble must be inserted into the pipeline) until the previous instruction has been completed.

```
1       add r1, r2, r3
2       add r4, r1, r2 ; RAW HAZARD
```

Listing 4.1: Example of DLX ASM code for RAW Hazard

The *Hazard Table* is made of a table with 32 entries (one for each register of the DLX). Each entry is made of 3 bits. When an instruction exits the Decode Stage and enters the Execute Stage, the related destination address entry in the table is increased by one. When the instruction reaches the Write Back Stage, the corresponding entry is decreased by one.

When a new instruction is decoded, its source registers are checked inside the *Hazard Table*: if one or both the registers have an entry greater than 0 means that one of them is not available (i.e. a previous instruction is still writing on that register) and the Decode Stage raises a *Hazard Signal* towards the Control Unit that will insert stalls inside the pipeline until the hazard is solved. When the hazard is over, the instruction will be processed inside the pipeline.

4.4 Comparator

The straightforward way to implement a comparator, allows only to check if two operands, `A` and `B`, are equals. The solution is sketched at figure 4.11, which is based on N XNOR, where N is the number of bits of the operands and an AND gate with N inputs.

Even if this solution is extremely compact, it allows to perform only the equality comparison; since this DLX implementation has the ability to perform complex conditional branch instructions (refer to the Instruction section 2.5) and conditional set instructions (refer to the Set-Like Operations Unit 5.2) a more complex solution is requested. For example, if a conditional branch, based on the condition `A` $>$ `B` (strictly greater) is executed, the exact check of this precise condition has to be performed.

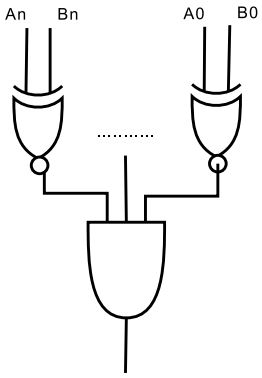


Figure 4.11: Design of the basic comparator

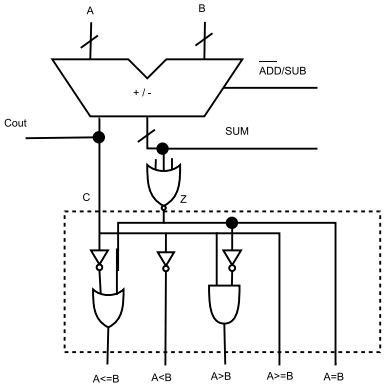


Figure 4.12: Design of the advanced comparator

The advance comparator exploits the comparison by performing a subtraction between `A` and `B` and then checking the result. This DLX implementation is based on a P4 adder that is able to perform subtraction and then a set of checks, the same that are in the 4.12 are performed in order to generate the comparison outputs. The comparisons can be performed using the following boolean equations, where C is the carry-out and Z is the zero check (all bits of the results are zeros):

$$A > B \rightarrow C \cdot \overline{Z}$$
$$A \geq B \rightarrow C$$
$$A < B \rightarrow \overline{C}$$
$$A \leq B \rightarrow \overline{C} + Z$$
$$A = B \rightarrow Z$$
$$A \neq B \rightarrow \overline{Z}$$

In order to avoid to propagate six different signals, the outcomes of the comparisons are encoded into a signal `LGET` on two bits. The encoded value are the ones in the Table 4.2.

LGET	Case
01	$A < B$
00	$A \leq B$
11	$A > B$
10	$A \geq B$

Table 4.2: LGET encoding

```
1  LGET <= "01" when (a_l_b = '1') else
2    "00" when (a_le_b = '1') else
3    "11" when (a_g_b = '1') else
4    "10" when (a_ge_b = '1') else
5    "00";
6
```

Listing 4.2: VHDL code for the encodig

The ordering of the comparisons in the `when` statement is not casual nor follows the normal patterns. The strictly lower comparison is made before the lower equals one, because, if the latter one is true, it means that also `A` lower than `B` is true, but not vice-versa. Using this encoding, the CPU can check the second bit in order to understand if `A` $\leq$ `B` or `A` $\geq$ `B`; instead, if the CPU wants to check only $<$ or $>$ comparisons the CPU has to check also the first bit.

A further improvement has been done with the respect to the advanced comparator in order to manage comparison between both signed and unsigned numbers. The carry value works like this:

- Carry = 1: if $A > B$ in unsigned
- Carry = 0: if $A \leq B$ in unsigned

The advanced comparator works with unsigned numbers only. So it simply needs to be adapted for cases in which signed comparison and unsigned comparison are different. They are shown in the Table 4.3 highlighted in red. It is easy to notice that in the red lines, A and B always have different signs. Therefore, the logic must work when the `UNSIG_SIGN_N` bit is 0, which means that the circuit is dealing with a signed number. In this case, the carry bit must be complemented. Knowing this, it's easy to derive the following logic:

$$Cout\_masked \leftarrow Cout \oplus (\overline{UNSIG\_SIGN\_N} \cdot (A_{msb} \oplus B_{msb}))$$

A	B	Carry out	Signed comparison	Unsigned comparison
2	3	0	Less	Less
4	3	1	Greater	Greater
3	3	1	Equal	Equal
-3	3	1	Less	Greater
-2	3	1	Less	Greater
-5	3	1	Greater	Greater
3	2	1	Greater	Greater
3	4	0	Less	Less
3	3	1	Equal	Equal
3	-3	0	Greater	Less
3	-2	0	Greater	Less
3	-5	0	Greater	Less
2	-3	0	Greater	Less
4	-3	1	Greater	Less
3	-3	1	Greater	Less
-3	-3	1	Equal	Equal
-2	-3	1	Greater	Greater
-5	-3	1	Less	Less
-3	2	1	Less	Greater
-3	4	0	Less	Greater
-3	3	1	Less	Greater
-3	-3	0	Equal	Equal
-3	-2	0	Less	Less
-3	-5	0	Greater	Greater

Table 4.3: All cases of possible comparison

So, the final unit corresponds to the one at figure 4.12 with an additional encoder before the output that encodes the six conditions into a signal on two bits. The resulting schema is the one in figure 4.13.

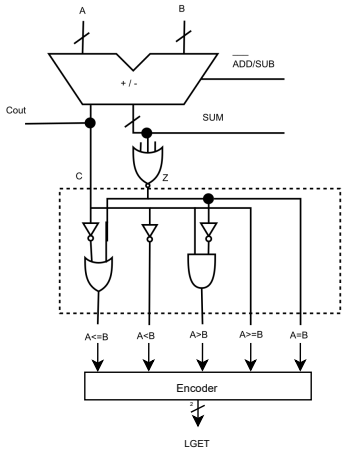


Figure 4.13: Final implementation of the comparator

It is important to mention the utility of the multiplexer before the comparator in Figure 4.14. There are two cases in which the comparator is used:

- Comparison between the content of two registers: in this case the multiplexer will select, with a 1, `i_RD2` and so the comparator will compare `i_RD1` and `i_RD2`. For example operations like BGT, BGE fall within this subcase.
- Comparison between the content of a register and an immediate: in this case the multiplexer will select, with a 0, `i_INP2` that is the immediate. So the comparator will compare `i_RD1` and `i_INP2`. For example, operations like SEQI, SNEI fall within this subcase.

The selection signal of the multiplexer is driven by the Control Unit.

4.5 Jump and Branch decision

When taking into consideration jump and branch decisions, there are many details to explain. First, as shown in Table 4.1, there are some instructions, like JAL and JALR, that need to save the value of the current program counter inside the register R31. For this reason, the value of WS is set to R31 in this kind of instruction.

Another important thing is that when jumping, there needs to be a change on the program counter. In order to exploit this feature, there is dedicated hardware inside the decode block that takes into account the computation of the next program counter. All the circuits and their functionalities are explained in the next section.

In the control word, there is a dedicated bit for the `JUMP_EN`. For the instructions that require a jump, such as J, JAL, JALR, CALL and RET, this bit is set to 1 by default. For other instructions that may jump, particularly the conditional branches, there is an intermediate step, the *decision*. In fact, for these instructions, the `JUMP_EN` bit is set to 0 by default. Then, thanks to the *Jump & Branch Control Unit* inside the Control Unit, as can be seen in Figure 2.6, based on the status of the comparison (i.e. the signal `LGET`), the `JUMP_EN` may be set to 1. This is very important because when jumping, there is a need to fetch the next instruction. For this reason, the first stage of the pipeline is stalled to avoid fetching the next instruction immediately.

According to this, is important to underline again that the signal *IF_STALL* is set to 1 to solve the Control Hazard caused by a taken jump, as explained in section 3.3.

The *Jump & Branch Control Unit* can be described with the following VHDL code:

```
1   if (IR_opcode = OP_BEQZ and LGET(0) = '0') then -- BEQZ
2     i_JUMP_EN <= '1';
3     IF_STALL <= '1';
4   elsif (IR_opcode = OP_BNEZ and LGET(0) = '1') then -- BNEZ
5     i_JUMP_EN <= '1';
6     IF_STALL <= '1';
7   elsif (IR_opcode = OP_BGE and (LGET(1) = '1' or LGET(0) = '0')) then -- BGE
8     i_JUMP_EN <= '1';
9     IF_STALL <= '1';
10  elsif (IR_opcode = OP_BLE and LGET(1) = '0') then -- BLE
11    i_JUMP_EN <= '1';
12    IF_STALL <= '1';
13  elsif (IR_opcode = OP_BGT and LGET = "11") then -- BGT
14    i_JUMP_EN <= '1';
```

```
15      IF_STALL <= '1';
16    elsif (IR_opcode = OP_BLT and LGET = "01") then -- BLT
17      i_JUMP_EN <= '1';
18      IF_STALL <= '1';
19    elsif (IR_opcode = OP_CALL and BUSY_WINDOW = '1') then -- CALL
20      CALL <= '0';
21      i_JUMP_EN <= '0';
22    elsif (IR_opcode = OP_CALL and BUSY_WINDOW = '0' and i_SPILL_delay = '0') then --
        CALL
23      CALL <= '1';
24    elsif (IR_opcode = OP_RET and BUSY_WINDOW = '1') then -- RET
25      RET <= '0';
26      i_JUMP_EN <= '0';
27    end if;
```

Listing 4.3: VHDL code for the conditional branch

To check if a branch must be taken or not:

- BEQZ: if the LSB of `LGET` is 0, then it means that there is equivalence with 0
- BNEZ: if the LSB of `LGET` is 1, then it means that there is don't have equivalence with 0
- BGE: if the LSB of `LGET` is 0 or if the LSB+1 of `LGET` is 1
- BLE: if the LSB+1 of `LGET` is 0, then it means that there is less equal
- BGT: if LGET = 11
- BLT: if LGET = 01

Another aspect to underline is the management of *CALL* and *RET* operations. They are like Jump Instructions, but there is something more. If there is `BUSY_WINDOW` equal to 1, it means that the current window is still in use, and so no operation on it have to be executed. In fact `JUMP_EN` is set to 0.

The last case is the one when `BUSY_WINDOW` is 0 and so the execution of the *CALL* procedure or the *RET* operation is feasible. In fact, *CALL* is set to 1.

4.6 Next Program Counter computation

The next program counter computation is included inside the decode block to manage Control Hazards created by taken jumps in a better way. In addition, to lighten the load on the ALU, a dedicated P4 adder has been included inside the decode block to compute the next program counter. The complete schematic of the decode is shown in Figure 4.14.

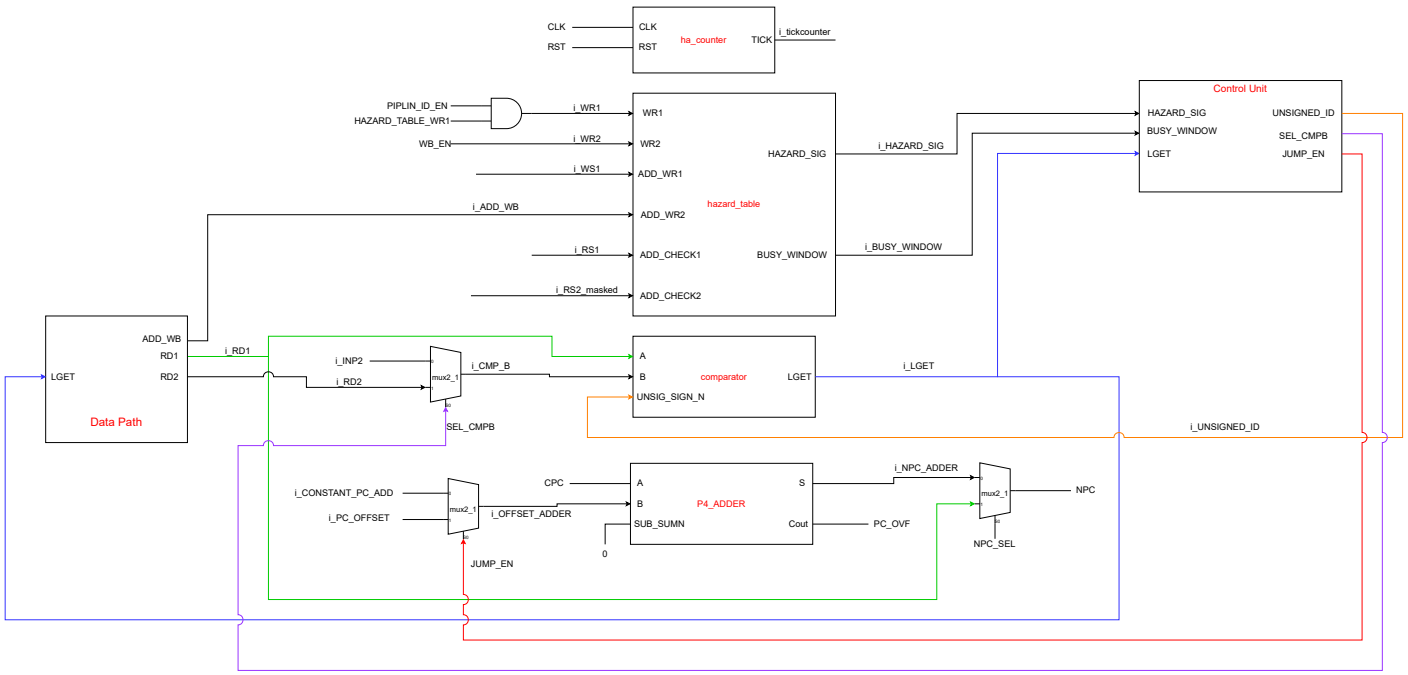


Figure 4.14: Schematic of the decode block

There are 2 possible conditions:

1. `JUMP_EN = 0`: this means that there is no jump. So the program counter should be increased by a constant value that is 4, because each instruction is 32 bit long so they are 4 bytes aligned. By comparing the explanation and the schematic, it's clear how the left multiplexer, with a 0, will select the signal `i_CONSTANT_PC_ADD`. Then the two signals enter the P4 Adder and the next program counter is computed.

2. `JUMP_EN = 1`: this means that there is a jump. So the program counter should be increased by an offset. Looking at the schematic, it's clear how the left multiplexer, with a 1, will select the signal `i_PC_OFFSET`. Then the two signals enter the P4 Adder and the next program counter is computed.

The multiplexer on the right allows the CPU to select an *Next Program Counter* the value computed by the adder or a complete different value taken from a Register. This multiplexer is controlled by a signal from the Control Unit. The are only three instructions in which the value of `NPC_SEL` is 1, and so it selects RD1, are:

- JR
- JARL
- RET

In fact, in these cases, the absolute value of the program counter to be used is the one read from the register file, in particular from `RD1`.

4.7 System Tick Timer

The DLX microprocessor offers a 32 bits System Tick timer inside it, that is very useful in considering timing behaviours. The implementation is based on a Half Adder Counter, that allows to count from 0 to $2^{32} - 1$.

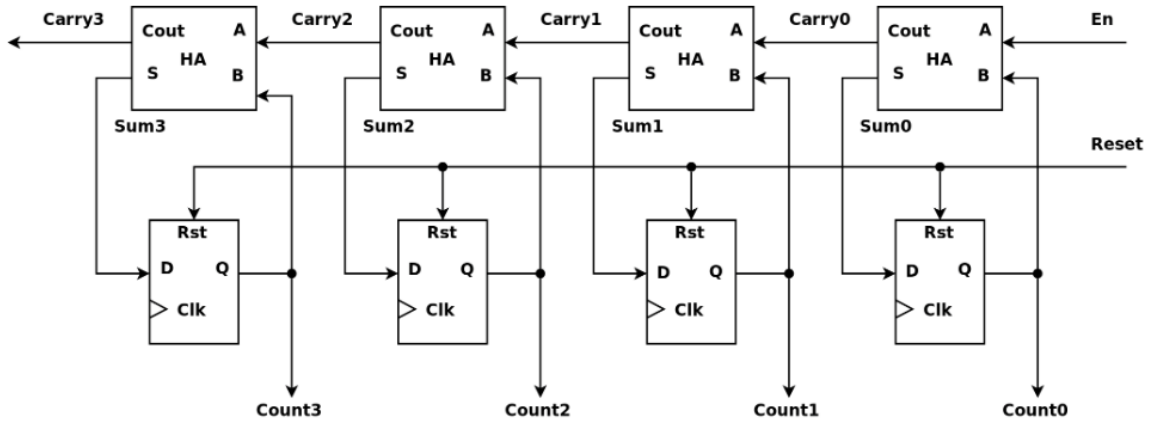


Figure 4.15: Schematic of Tick Timer implementation

By mean of this, the user can count the number of ticks and use the value to estimate the duration of a specific portion of code. It's meant to be used as a relative value so as a delta delay between a starting point and an end point.

However, it's important to underline the fact that the Tick Timer value is taken during the Decode Stage and it has to pass through all the entire pipeline until the Write Back, so for advanced applications the user should take under account this aspect together with the fact that some stalls can be inserted during the execution.

To compute the execution time given the CPU frequency and a start and an end time:

$$T_{exec} = \frac{CC_{end} - CC_{start}}{F_{CPU}}$$

CHAPTER 5

Execute Stage

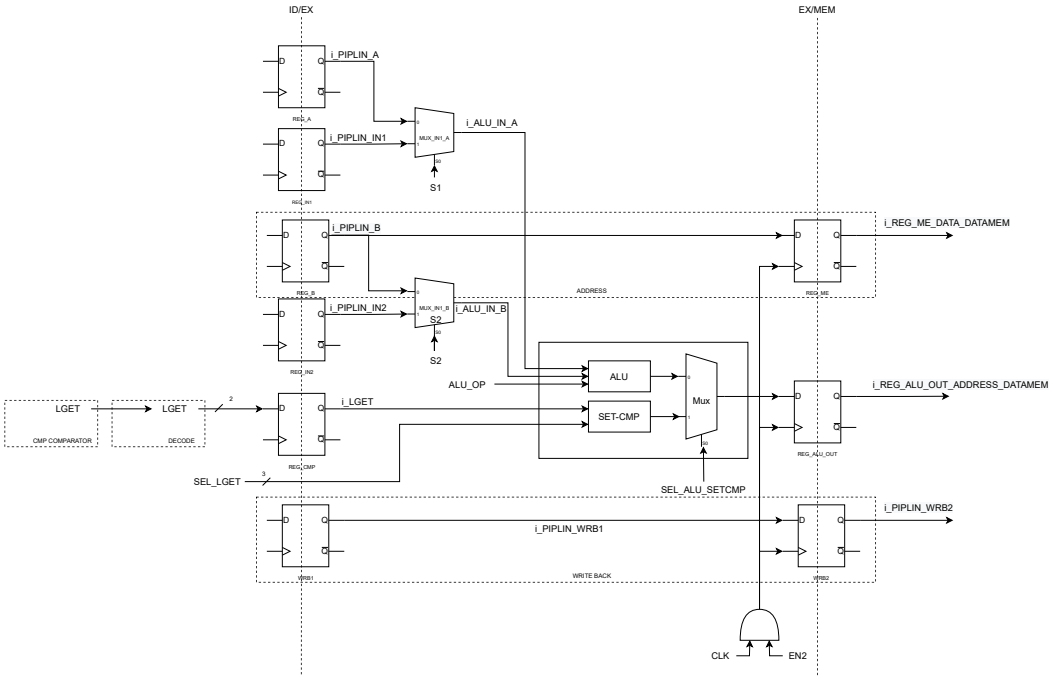


Figure 5.1: Execute stage

The Execute Stage is the second stage of the Datapath, and it is used to perform computation over data, like sum, multiplication, shift and logical operation. Finally, the Control Word coming from the Control Unit allows to correctly configure the internal signal to perform the desired function with the correct operands. In fact, in this stage, multiple signals can be configured:

- `EN1`: this signal is used in order to enable/disable the registers sampling in the DECODE/EXE junction;
- `EN2`: this signal is used in order to enable/disable the registers sampling in the EXE/MEM junction;
- `S1`: this signal is used to select between the output of `REG_A` (from register file) and `REG_IN1` (immediate). If 0, `REG_A` is selected;

- `S2`: this signal is used to select between the output of `REG_B` (from register file) and `REG_IN2` (immediate). If 0, `REG_B` is selected;
- `SEL_ALU_SETCMP`: it is used to select the input for `REG_ALU_OUT` between the SET-CMP and ALU outputs;
- `SEL_LGET`: this is a 3 bits signal, that is used to select among the `SEQ, SNE, SLE, SLT, SGE, SGT` operation for the SET-CMP unit;
- `ALU_OP`: this is a 5 bit signal, that is used to select the correct operation among the sub-unit that composes the ALU. Refer to Table 5.1 for a complete description.

Besides the main path, used to compute the output result, two addition paths (the ones in the dotted rectangles) are present in order to correctly manage the execution of all the operations in the instruction set:

- A path for the propagation of the register value to be saved into the memory in case of a load. It was not possible to pass through the addition with `B` and `0` because it's already needed in order to compute the address. So, the output from `REG_B` is directly connected to the input of `REG_ME`;
- The second path is the one used to manage the Write Back. At each step, until the MEM one, the WB must be propagated through three stages (decode, execute and memory). So, the output of the `WRB1` register goes into the input of the `WRB2` register.

5.1 ALU: Arithmetic Logic Unit

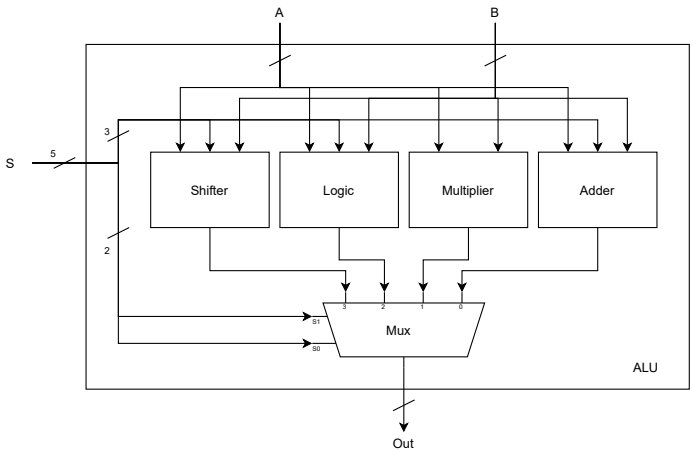


Figure 5.2: ALU unit

The Arithmetic Logic Unit can be seen as a block, that given a selection signal and the inputs is able to perform computation over the operands. The ALU implementation described in this document is based on the following block:

- Adder
- Multiplier
- Logic unit
- Shifter

Each one of these blocks will be explained in the following sections.

The base concept is that, internally, the 4 units are selected through a multiplexer that takes two out five bits from a selection signal called `OP`. Having 5 bits to describe the type of operation, the possible combinations and their relative operations are:

`OP`	**Unit**	**Operation**
000 00	ADD	ADD
001 00		SUB
000 01	MUL	MUL
000 10	LOGIC	AND
001 10		NAND
010 10		OR
011 10		NOR
100 10		XOR
101 10		XNOR
000 11	SHIFT	SHIFT RIGHT
001 11		SHIFT LEFT
010 11		ARITH SHIFT RIGHT
011 11		ARITH SHIFT LEFT
100 11		ROTATE RIGHT
101 11		ROTATE LEFT

Table 5.1: ALU operations encoding

The two LSBs are the ones used as a selection input for the multiplexer, that selects from which ALU unit taking the result. In fact, they univocally define the unit to be used. The remaining three MSBs are used as input for the units that compose the ALU in order to select the correct operation.

5.1.1 Adder

The straightforward way to implement an adder is to use the Ripple Carry Adder structure, which is composed of $N-1$ Full Adder and one Half Adder (the first), where N is the number of bits of the two operands. This solution is not optimal from a timing point of view due to the time needed to propagate the carry, which defines the critical path, that is the bottleneck.

Since the sum and the subtraction are two of the most common operations, the DLX includes an adder that is based on a CLA - Carry Look Ahead (Sparse Tree) and a Carry Select Like Adder. The complete structure can be seen in figure 5.3.

As said before, the adder is composed of two blocks:

- **Carry Select Like Adder**: The main point of the Carry Select Adder is that it doubles the complexity of the adder itself in order to obtain better performances. It is composed of two RCA to perform two sums in parallel.

  The idea is to compute both results, on 4 bits in this case, for both when the carry-in is equal to '0' or '1'. In this way, the results are computed in parallel for all the stages, even if the carry-in is '0' or '1'; then the carry-in is used to mux against the two results (on 4 bits) and the two carry-outs. Finally, the carry-out will be used as a result selection signal for the next Carry Select unit.

  The processor is paying complexity in order to reduce the sum computation time. In fact, by having a carry out that is used as carry in for the next state, there is still propagation but is lower.

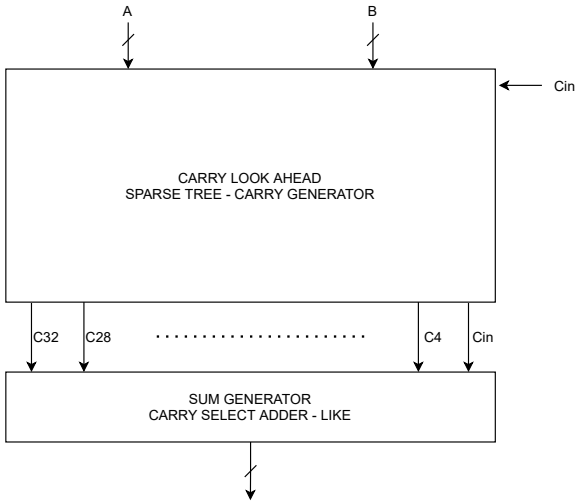


Figure 5.3: Booth's multiplier on 32 bits

The DLX implementation, instead of using a straightforward one of the Carry Select Adder, uses a modified version of it. It has been accomplished using *CLA - Sparse Tree Carry Generator* and a *Carry Select Like Adder*. The base idea is to use the CLA in order to compute a carry every n bits; these carries are then fed into the sum generator that uses them to compute the results in parallel.

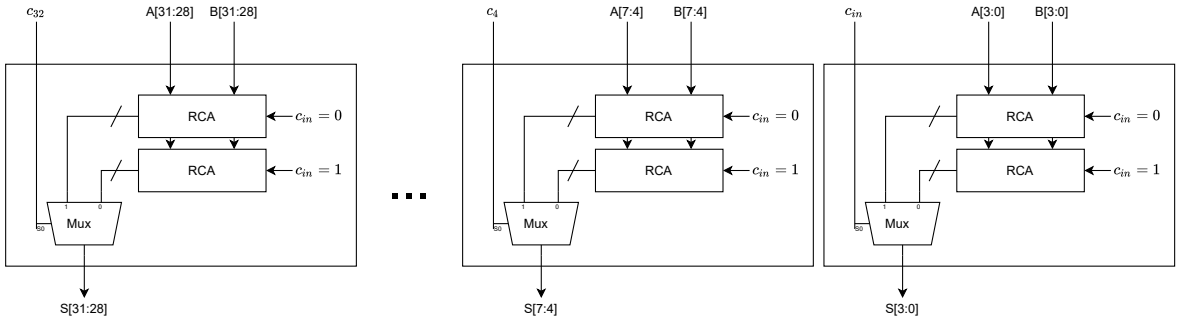


Figure 5.4: Carry Select Like Adder block for a 32 bits implementation

- **Carry Look Ahead - Sparse Tree**: this block is used to compute the carry out every 4 bits. The idea behind the CLA is to compute several carries simultaneously and to avoid waiting until the correct carry propagates from the stage of the adder in which it has been generated. This is done thanks to the *propagate (P)* (that is 1 if the carry-in is equal to the carry-out) and *generate (G)* (that is one if carry-in is 0 and carry-out is 1).

a	b	c_{in}	out	c_{out}	p	g
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	1	0	1	0
0	1	1	0	1	1	0
1	0	0	1	0	1	0
1	0	1	0	1	1	0
1	1	0	0	1	0	1
1	1	1	1	1	0	1

Table 5.2: Computation of propagate and generate bits

$$g = a \oplus b \qquad p = a \cdot b \tag{5.1}$$

The base idea is to write any s_i, that is the i-esim bit of the sum and c_i, the carry-out at i index in function of p and g. It's possible to use p and g to express the same result:

$$s_1 = a \oplus b \oplus c_{in} = p_1 \oplus c_0$$
$$c_1 = a \cdot b + a \cdot c + b \cdot c = a_1 \cdot b_1 + (a_1 + b_1) \cdot c_0 = g_1 + p_1 \cdot c_0$$

The crucial point is that it's possible to compute the carry at i position only using the initial carry-in c_{in} and p and g generate in the current and previous blocks. Tree are a family of Carry Look Ahead that differ for the carry-logic. They are based always on *propagate* and *generate*. So

$$carry = g + p \cdot c_{in}$$
$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

where

- $i \geq k > j$
- $G_{x:x} = g_x$ and $P_{x:x} = p_x$
- $g_0 = C_{in}$

The white and grey blocks in the Sparse Tree block at 5.6, that are used in the `carry_generator` block, are PG and G blocks. Normally two blocks are used, the first G generates only $G_{i:j}$ and the other PG both $G_{i:j}$ and $P_{i:j}$. The base idea is to combine their outputs and take only the

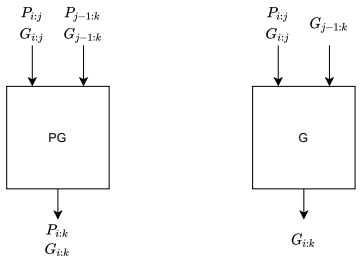


Figure 5.5: PG and P block

G as carries.

So, Carry Look Ahead - Sparse Tree needs a starting block that generates all the p and g for all the couples of bit using the 5.1 equation. The sparse tree and the PG network structure is shown in the figure 5.6, this block is called `prop_gen_generic` and is made of `prop_gen` block in order to compute p and g.

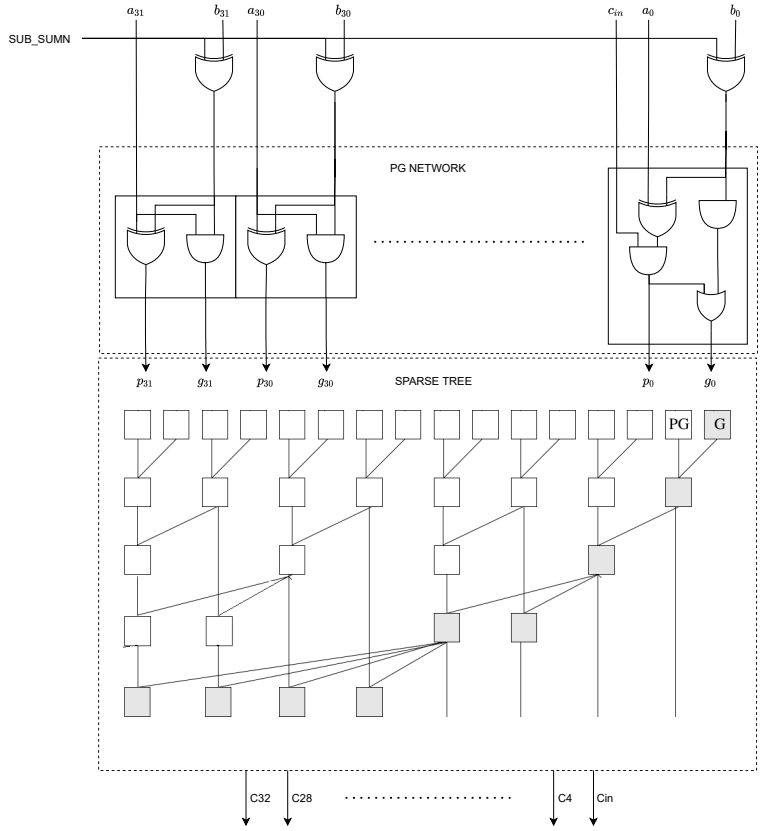


Figure 5.6: Carry Look Ahead - Carry Generator Block

To be able to perform both subtraction and sum, the first block must be modified and must include the logic to manage also the carry-in, that in case of subtraction is '1'. This is not enough to perform subtraction, in fact, an additional signal called `SUB_SUMN` is needed. The same structure can be used to implement subtraction by simply adding a XOR on each B input with the `SUB_SUMN` control signal.

The subtraction in 2's complement can be implemented as $A + \overline{B} + 1$. To implement this, the carry-in needs to be set to '1' (so `SUB_SUMN` = '1') in order to add 1 and invert the B input by using the XOR. In fact:

x	y	$y \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

This solution is good because the PG and G block has the same delay, that is driven by G and since both include it, they are the same. Many paths have the same delay and the load on components is good since an output of a block is connected to a maximum of 2 other blocks. These two factors bring a good *equilibrium* to the entire structure.

5.1.2 Multiplier

In order to overcome the limitation of the array multiplier, this DLX implementation includes a modified version of the Booth's multiplier, since the multiplexer for the partial results to be added is only on two bits instead of three. The Booth's algorithm copes with 3 bits at a time, so the number of stages is $N/2$ (this corresponds to the number of the encoders) and this allows to speed up the result computation. The Booth's algorithm is the following:

```
1   i = 0
2   P = 0
3   while i ≤ M - 2 loop
4       P = P + Vp( B_{i+1}, B_i, B_{i-1} )
5       A = A * 4
6       i = i + 2
7   end loop
```

Where `P` is the final value of the product and during the algorithm execution it will contain the partial result; `M` represents the number of bits of the multiplicand, in this case B. The algorithm takes as convention that $B_{i-1} = 0$. The `Vp` is a lookup Table (see 5.3), that return the value to add to `P`, according to the 3 bits selected. The value of A is multiplied by 4.

B_{i+1}	B_i	B_{i-1}	
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 5.3: Booth's LUT

The Booth's multiplier work with 3 main components, supposing A is the multiplicand and B the multiplier:

1. $N/2$ encoders in order to take 3 bits from the operand B; the two LSB are used as a selection signal for the multiplexers and the last one, the MSB, for the adder. In fact, as shown in Table 5.3, when the MSB is 1 the value to be added to the partial result is positive, negative otherwise. For this reason, the inputs of the multiplexers are only $\{0, A, 2A\}$. Since it's necessary to generate also negative values, like $-A, -2A$, the MSB of the three bits is used as input for the adder. This signal, called `SUB_SUMn` is used to define if the operation is a sum or a subtraction; if it is 1, a subtraction is performed;

2. $N/2$ multiplexers that select only among $\{0, A, 2A\}$, since at each stage A must be multiplied by 4, a shift by two is done starting from A of the previous multiplexer;

3. $N/2$ ripple carry adders, that allow performing the partial sums. Since the final results will be on $NBIT \cdot 2 + 1$ bits, the adders in each level have been optimized in order to work only with the minimum bits needed. In fact, the adder at i level, will generate the result on $NBIT + 2 \cdot i$ bits. As said before, a further signal called `SUB_SUMn` has been added in order to be able to perform the subtraction. The Ripple Carry Adder has been selected for its simplicity and, since multiplication is a less common instruction, it was not worth using a more sophisticated adder. This allowed reducing the total area of the multiplier itself.

The multiplexer implements the LUT and at the same time the $A = A \cdot 4$. The two values from the two multiplexers are summed together via an adder, this implements the partial sum. The overall structure can be observed at 5.7.

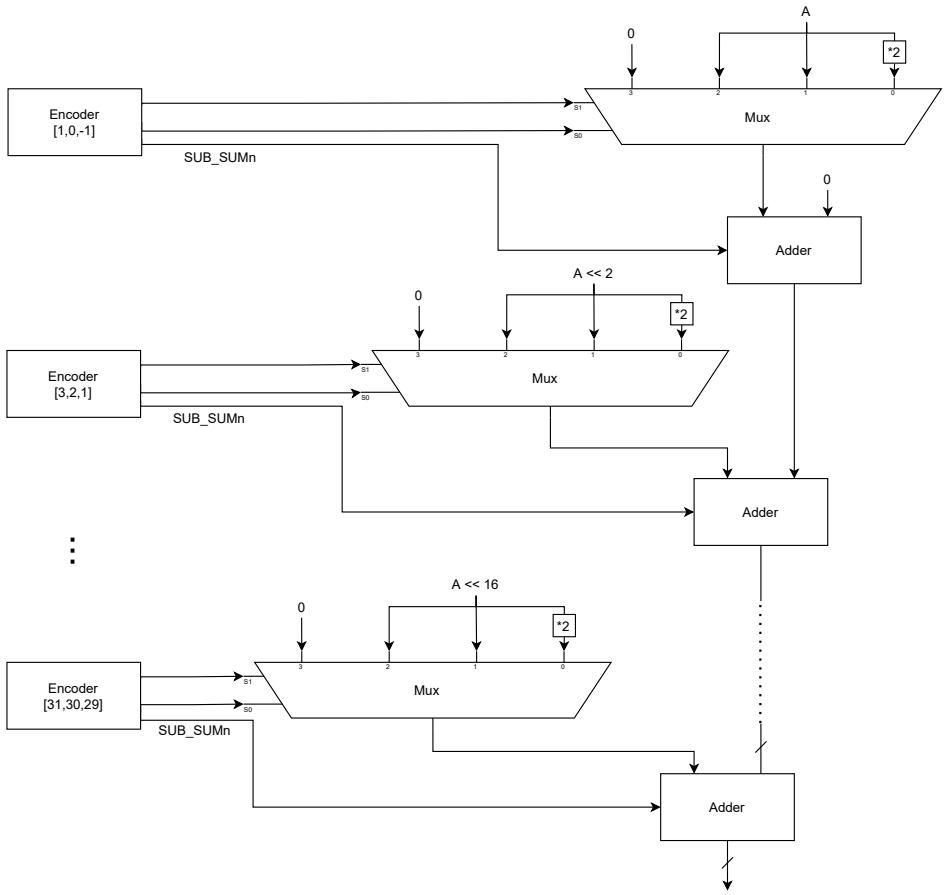


Figure 5.7: Booth's multiplier on 32 bits

5.1.3 Logic Operands

The basic and most simple implementation of a logic unit is based on single logic gates on N bits whose outputs are muxed, in order to generate the correct output. The problem with this solution is that the number of input signals of the multiplexer is extremely high; this implementation does not only suffer from a delay point of view but, since each logic function is implemented with a specific gate, also has a very high area usage.

In order to overcome the problems highlighted before, a more compact implementation has been chosen: the T2 logic unit.

This logic unit allows to perform AND, NAND, OR, NOR, XOR and XNOR using only 5 NAND gates, on two levels, and 4 selection signals. The schematic is the one in figure 5.8.

In order to compute one of the logical instructions, the select signals are properly activated as follow:

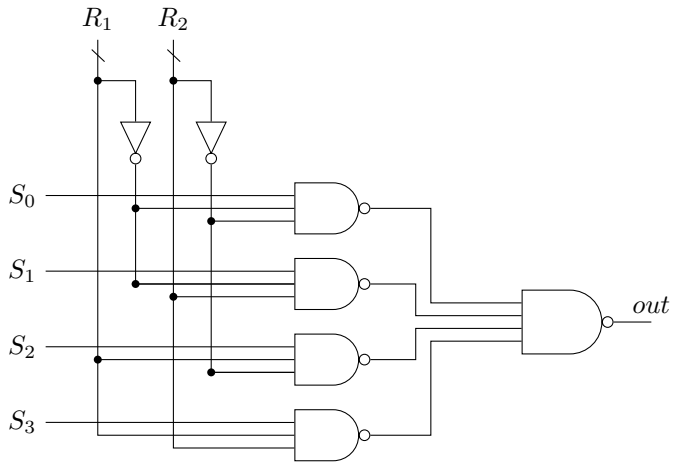


Figure 5.8: Logic unit

S_0	S_1	S_2	S_3	Operation
0	0	0	1	AND
1	1	1	0	$NAND$
0	1	1	1	OR
1	0	0	0	NOR
0	1	1	0	XOR
1	0	0	1	$NXOR$

Figure 5.9: Logic input signals with the relative operation

For example, in order to generate the AND logical operation, $S_3 = 1$, so that $out = R_1 \cdot R_2$; on the other hand, for the NAND, $S_0 = S_1 = S_2 = 1$ and $S_3 = 0$, so that $out = \overline{R_1} \cdot \overline{R_2} + \overline{R_1} \cdot R_2 + R_1 \cdot \overline{R_2} = \overline{R_1} \cdot \overline{R_2}$ that using the De Morgan law $out = \overline{R_1 \cdot R_2}$. This allows to obtain the best performances also because all paths work in parallel, compacting the area and the delay.

Since only the 3 bits are used to select among the logical operations (`S` signal), a direct correspondance is needed to generate the signal show in Table 5.9. The following table, shows the conversion:

`S`	Decoded signal
000	0001
001	1110
010	0111
011	1000
100	0110
101	1001
110	0000

Table 5.4: Conversion table, from `S` input signal on 3 bits into 4 bits

5.1.4 Shifter

The implemented shifter allows to perform shift right, logical/arithmetical shift left and left/right rotate using the full operand `A` on 32 bits and 5 bits from the second one `B` and three *control signals*. Differently from the T2 version, it uses an addition signal in order to be able to manage also the rotate instruction. The implementation takes three inputs:

- `A`: the operand to be shifted/rotated;
- `B`: only the 5 LSB [4:0] are used to select first the mask to be used and then the starting point from that mask;
- `SEL`: it encodes the operation type; the second bit is used to select among arithmetic and logic, the third bit is used to select the direction of the shift/rotate (left/right) and the first one is used only if the operation is a rotate. This is the encoding:

`SEL`	**Operation**
000	Shift logic right
001	Shift logic left
010	Shift arith right
011	Shift arith left
100	Rotate right
101	Shift right

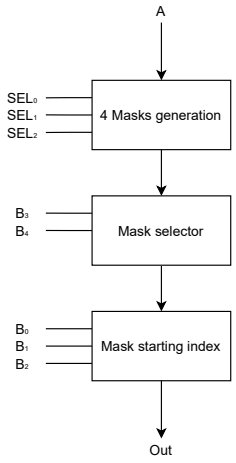


Figure 5.10: Blocks of the Shifter/Rotate Unit

The unit performs the requested operation in three stages, sketched in figure 5.10:

1. The first consists of preparing 4 possible "masks", each already shifted of 0, 8, 16, 32 left or right depending on the configuration. This allows shifting for all 32 bits. It copies the input `A` into the 4 masks that will be used by the next stage. Being on 32 bits, the generated masks are on $32 + 8 = 40$ bits. The only difference between this implementation and the T2 one, is that, in case of rotate, the additional 8 bits of the masks are filled with the corresponding 8 bits that are going "out" during the rotation.

2. The second level perform a coarse grain shift, that consists of selecting one mask among the 4 possible ones generated in the previous stage. This selection is done using the bits 4, 3 of `B`.

3. The third level, using the bits 2, 1, 0 of `B` and the selected mask, performs a fine grain refinement. The 3 bits allow choosing the starting index from the mask, granting a selection among 8 positions.

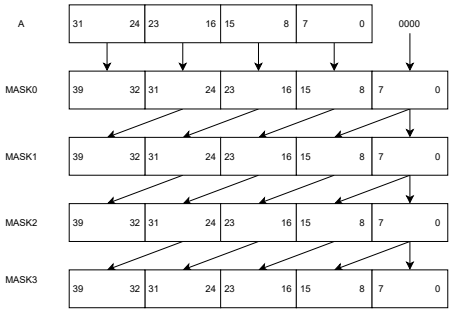


Figure 5.11: Masks for left shift

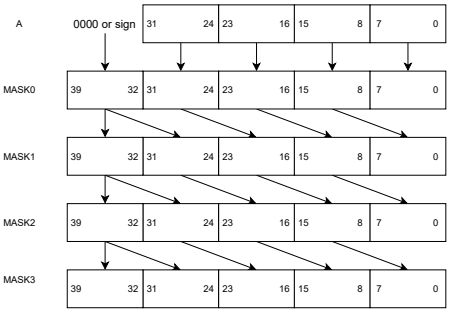


Figure 5.12: Masks for right shift

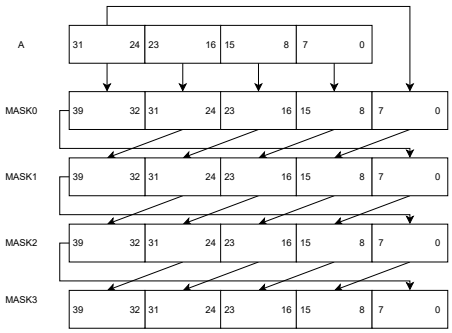


Figure 5.13: Masks for left rotate

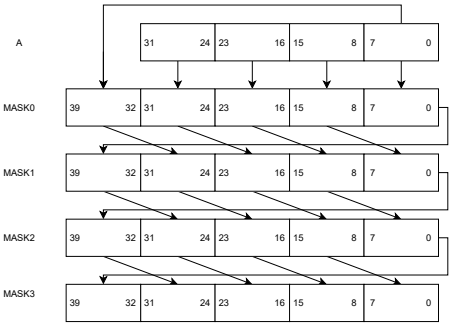


Figure 5.14: Masks for right rotate

Figure 5.15: Masks for shift unit on 32 bits

Examples
For example, if a shift left of 9 bits `A` is needed, where `A=18`, the corresponding `B` value will be 1001; this means that the second masks will be taken and the output result will from the bit at position $40 - 1 = 39$ to the one at $39 - 32 = 7$ included.

$$\text{MASK 2: } 0\underbrace{\mathbf{0000000\ 00000000\ 00000000\ 00010010\ 0}}_{\text{shifted A}}0000000$$

On the other hand, if a right shift is needed, the masks are generated in the opposite way, so the zeros are put in the MSB of the mask, shifted by 0, 8 ... positions. There is also a distinction between the arithmetic and the logic shift; in the first case, instead of filling the "empty" bits with zero, the operand sign is used. For example, if to right shift `A=-18` of `B=3` bits, the first mask is used:

$$\text{MASK 1: } 11111\underbrace{\mathbf{111\ 11111111\ 11111111\ 11111111\ 11101}}_{\text{shifted A}}110$$

In the last case, let's suppose to rotate right `A=1255` (=10011100111) by 5 position:

$$\text{MASK 1: } 11\underbrace{\mathbf{100111\ 00000000\ 00000000\ 0000100\ 111}}_{\text{rotated A}}00111$$

As you can see, in case of MASK 1 for the right rotation, the 8 LSB of `A` are copied into the 8 MSB of the mask.

5.2 Set-Like Operations Unit

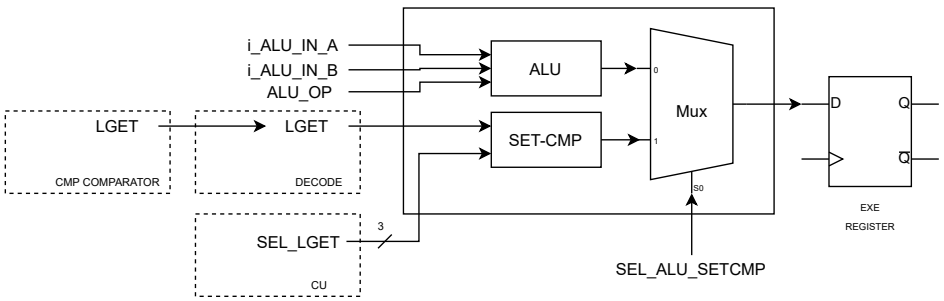


Figure 5.16: Execute stage with focus on the SET-CMP unit

The DLX execution stage includes also the possibility to set the value of a register accordingly to the outcome of the comparison of two operands; the operands can came from two sources:

1. Both from the register file, trough `REG_B` and `REG_A` (refer to figure 5.1);
2. One from the register file (`REG_A` by setting `S1 = 0`) and the immediate from `REG_IN2` (refer to figure 5.1).

The unit designed to perform this set operation is called `set_comparator`. Using a behavioural process, the latter is able to generate the corresponding '1' or '0' to be set to the register, accordingly to the comparison result. The output on $NBIT_DATA$ bits will be muxed with the one coming from the ALU unit, using the CW that is configured in the Control Unit.

In order to decrease the area and since the comparison was already generated by the comparator unit in the decode stage (refer to section 4.4), the `set_comparator` unit takes the `LGET` signal that came from the decode unit and perform the following checks:

Operation	**VHDL implementation**
SEQ	`not LGET(0)`
SNE	`LGET(0)`
SLE	`LGET(1) = '0' or LGET(0) = '0'`
SLT	`LGET = "01"`
SGE	`LGET(1) = '1' or LGET(0) = '0'`
SGT	`LGET = "11"`

Table 5.5: Performed checks in order to generate '1' or '0' accordingly to the comparison outcome. Refer to table 4.2.

CHAPTER 6

Memory Stage

The memory stage allows to perform operations with the memory, the DRAM in this case, in order to load or store a value from/to the memory respectively. Figure 6.1 shows the Memory Stage of the DLX pipeline.

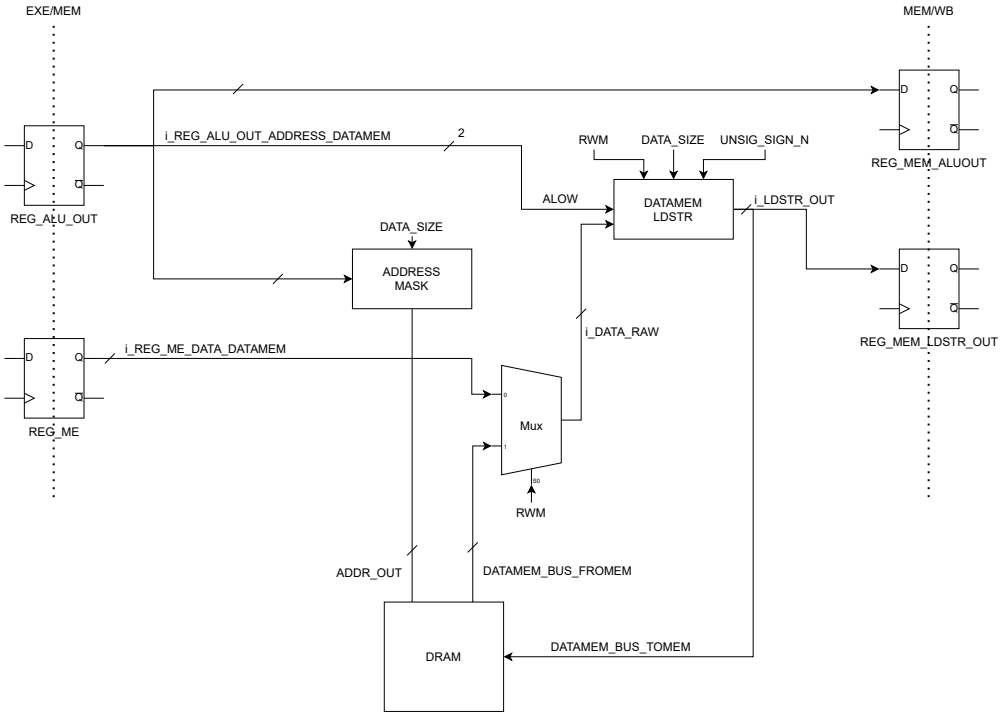


Figure 6.1: Memory stage

If the instruction that reaches the memory stage is a load, the value stored into the `REG_ALU_OUT` register is fed to a block that performs the masking of the address itself. Then the masked address is given as input to the external memory in order to retrieve the data. A multiplexer with two inputs allows to select the data coming from the memory and redirects it to another unit, called `datamem_ldstr` that uses the `DATA_SIZE` and `UNSIG_SIGN_N` inputs to select the correct bits for word, half-word and byte.

In the case of a store, the address in which the data must be saved into the memory is masked too via the Address Mask Unit and its output is given to the DRAM. The data stored in the `REG_ME` register

is selected, by correctly configuring the multiplex to choose it, and put as input to the `datamem_ldstr` unit. The output of this unit is put as input data to the DRAM, that performs the store.

6.1 Address Mask Unit

The basic idea behind the Address Mask Unit is to modify the address, given as input, depending on the data size to store or load. All the possibility and masking procedures are summed up in Table 6.1.

DATA_SIZE	Dimension	Masking
00	word	`ADDR_IN(ADDR_IN'length-1 downto 2) & "00"`
01	half word	`ADDR_IN(ADDR_IN'length-1 downto 1) & "0"`
10	byte	`ADDR_IN`

Table 6.1: Address masking for all the three possible cases

Give an address, it must be correctly aligned depending on the dimension of the data the processor wants to write/read to/from memory. Refer to the 6.1 VHDL snippet. The problem with the alignment arises when data to be accessed are larger than the addressable unit (word vs byte).

After the generation of the correct address, is a DRAM duty to output the data in the proper byte lanes, as explained in Table 2.3. The operation of reading from the correct byte lane and apply a sign extension (if requested) is made by the 6.2 *Load-Store Unit*.

```
1     with DATA_SIZE select
2         ADDR_OUT <= ADDR_IN(ADDR_IN'length-1 downto 2) & "00" when "00",
3         ADDR_IN(ADDR_IN'length-1 downto 1) & "0"  when "01",
4         ADDR_IN when others;
```

Listing 6.1: VHDL code for address alignment

6.2 Load-Store Unit

The Load-Store Unit is used to perform the following operations:

- Extract the correct bits from the data coming from the memory, using the address generated by the Address Mask Unit. Depending of the `DATA_SIZE` and the `ALOW` signals, the data is selected.

  When the address has been correctly masked to fetch a word (32 bits), the data doesn't need to be extracted, nor the sign extension is needed. In the other two cases, when managing half-word or byte, the correct portion of bits must be taken.

- If the unit is managing half-word or byte, the bits can be simply taken and set as the output of the Load-Store Unit only if working with unsigned. In this case, all the others bits are set to 0. On the other hand, if the byte or the half-word granularity is used and the `UNSIG_SIGN_N` is set to '0', a sign extension must be performed. In this case, the correct portion of bits is taken, as defined in Table 6.2, and the first bit of it used extends the sign.

- Data Replication in case of a Store to memory. Refer to Figure 2.7.

DATA_SIZE[1:0]	ALOW[1:0]	**Type**	**Selected bits**
00	--	word	[31-0]
01	0-	half A	[31-16]
01	1-	half A+2	[15-0]
10	11	byte A+3	[7-0]
10	10	byte A+2	[15-8]
10	01	byte A+1	[23-16]
10	00	byte A	[31-24]

Table 6.2: Bits selection using DATA_SIZE and ALOW

A more detailed explanation of the memory interface is available in section 2.4.3.

CHAPTER 7

Write Back Stage

The Write Back Stage is the last stage in the DLX pipeline and it is used to write the results, that can come from the memory (due to a load) or the computation of the ALU, into the File Register. Since this DXL is a pipelined processor, the destination register is available when the instruction is decoded and must be propagated until the write back register is needed. For this purpose, inside the entire Datapath, there is a skewing network that is used to have the register index exactly when necessary.

From the instruction itself, depending on its type, the destination address must be extracted. The position on a 32 bit instruction of the destination address is the following (refer to 2.5):

- R-Type: [25...21]
- I-Type: [20...16]

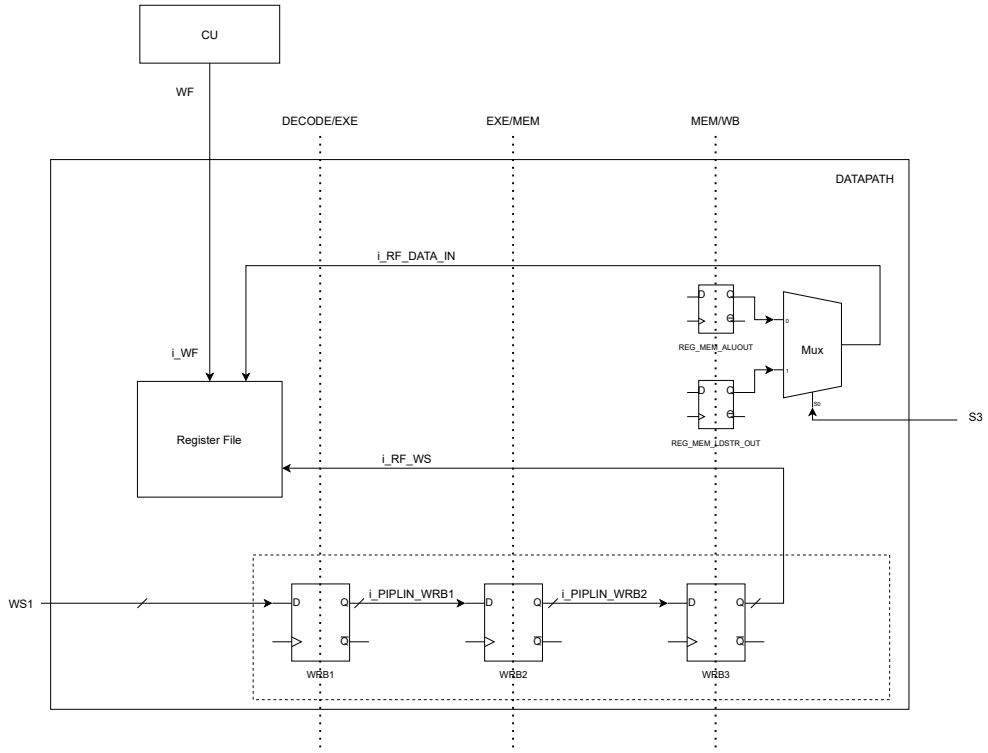


Figure 7.1: Write Back stage

Figure 7.1 shows the simplified diagram used to delaying the destination register (WS1), based on three registers in the Datapath. When the instruction reaches the Write Back stage in the pipeline, the register index on 5 bits is fed back to the write input of the Register File and an additional signal, called WF is used to enable the write.

An additional signal, called S3, is used to select which data to take as input for the register file. Data can come from two different sources:

- It can take directly the output of the Load and Store unit, via the REG_MEM_LDSR_OUT in case the selection signal S3 = 1
- It can take the output of the ALU (that includes set-cmp unit), via the REG_MEM_ALU_OUT in case the selection signal S3 = 0

The R0 register must be protected against the Write Back, in fact, it has to be always 0. For this reason, a masking procedure has been implemented in order to solve this problem. The WF signal is gated with a non-zero check over the i_RF_WS signal; in few words, the write enable signal is set to 0 if a write on R0 is requested.

```
1       i_WF <= WF when (TO_INTEGER(unsigned(i_RF_WS)) /= 0) else '0';
```

Listing 7.1: VHDL code gating for the WS signal coming from the CU

CHAPTER 8

Testing and Verification

Testing and verification are used to verify the correctness of the DLX. Testing can be defined as the process of observing and verifying the results of a component under specific conditions in order to find possible inconsistencies between the actual and the required behaviour.

The flow used to verify a component follows multiple steps:

- Creation of the Testbench
- Simulation
- Post Synthesis simulation (after the Synthesis and Optimization, refer to section 9.1)

8.1 Testbenches

All Testbenches developed for the DLX verification have been implemented using VHDL. The most critical components have been tested with their testbenches, like:

- Booth's multiplier
- P4 adder
- Comparator
- ALU
- Shifter
- Windowing Register File

The smaller components have been intensively tested while verifying the correct behaviour of the DLX itself.

The DLX Testbench is implicitly based on two processes; the first one manages and creates only the clock signal with a period of 1 ns, while the second one asserts the reset signal for the first clock period and then negates it to start the correct test execution.

To create the most flexible test possible multiple components have been instantiated; these provide all the external interfaces the DLX needs. So, besides the DLX instance, the Testbench also contains the following component:

- Instruction RAM (IRAM): this is a read-only memory that takes an external file, in this case, a .mem file, and uses it as a source for fetching the instruction. When the reset is performed, all

the instructions in the file are written in an internal array of 32 bits. At each clock cycle, the array is indexed with an index that comes from outside that corresponds to the DLX `PC`.

- Data RAM (DRAM): similarly to the IRAM, the DRAM is loaded at the startup using the same .mem explained before. This compiled file does not only contain the code instructions but, after a blank line, also the memory content the program should have once it starts. Since a DRAM must also be writable, the memory is implemented as a read/write one.

  To simplify the verification of the memory content, each time a value is written, the external file, that is the exact copy of the DRAM content, is updated too. In order to correctly access the data that are stored in the memory, the address comes from outside and corresponds to the one used in the Memory Stage (refer to Chapter 6).

  To replicate a more realistic environment and increase the test reliability, the DRAM has been enhanced with a `ready` signal that is '1' when it is ready. This has been implemented with a counter, that only after `data_delay` clock cycles makes the memory ready and allows to generate a meaningful value.

  Last but not least, the DRAM must be able to correctly manage all the different data sizes, both during the write and the read operation. So, an additional signal on two bits, called `MAS`, has been included. A more accurate description of the memory data size management is available in section 2.4.3.

Reducing the setup time needed to start a new simulation was a crucial point to speed up the entire verification process. For this reason, a bash script has been developed to generate the .mem file from an .asm one and move it into the correct testing folder. This allows to change only the .mem file reference in the Testbench. The bash script is the one at Appendix A.1 and some .asm examples are available at Appendix D.

8.2 Simulation

The verification of the behaviour of the DLX is based on the simulation of the DLX itself, using the Testbench wrote in the previous phase. Before starting the simulation, all the components must be compiled; an organizational improvement has been added for simplifying the compilation file organization. The folder structure is based on a hierarchical organization of the components, by using the alphabet letters as starting names. Each folder includes a `compile.script` file that is used to compile all the files in the folder and call all the others `compile.script` files in the sub-folders. Folders are organized in the following way:

```
DLX_vhd_fully_synthesizable/
├── compile.script
├── a.b-DataPath.core/
│   ├── compile.script
│   ├── a.b.a-windowedRF.core/
│   │   └── compile.script
│   └── a.b.d-ALU.core/
│       ├── compile.script
│       ├── a.b.d.c-adder.core/
│       │   └── compile.script
│       └── a.b.d.b-multiplier.core/
│           └── compile.script
└── a.d-decode.core/
    └── compile.script
```

```
└── test_bench_and_memory/
    ├── compile.script
    ├── TB_packages/
    │   └── compile.script
    ├── TB_romem/
    │   └── compile.script
    └── TB_rwmem/
        └── compile.script
```

Once *Questa Sim* is loaded, the `DLX_vhd_fully_synthesizable/compile.script` can be executed using the `source` command. After a while, all the components of the DLX will be compiled. Another TCL script, called `launch_sim_waves.script` (refer to Appendix A.2), has been developed in order to start the simulation and load the waves of the registers from `R0` to `R31` in the Wave view.

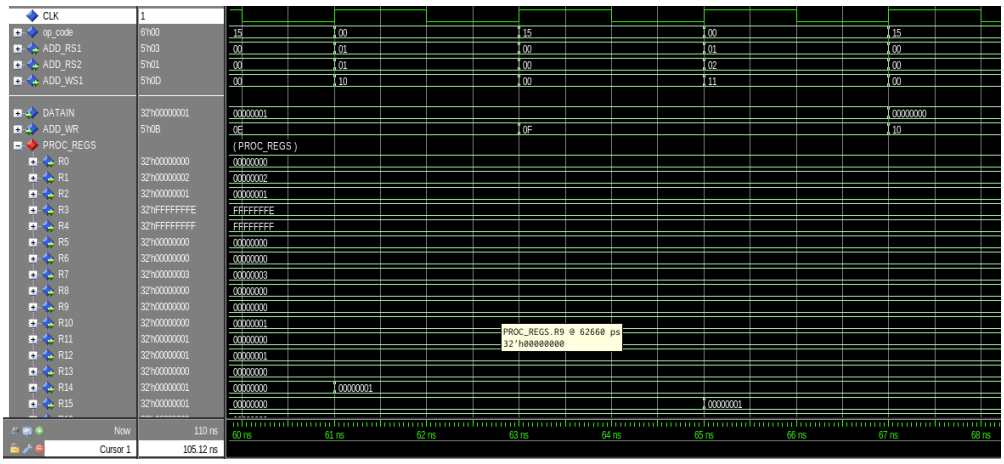


Figure 8.1: Questa Sim Simulation

8.3 Post Synthesis Simulation

After the Synthesis and Optimization phase, which allows to improve the circuit under multiple points of view and perform the tech mapping, it's possible to simulate the behaviour of the circuit using the standard cells.

Another script, that is similar to the `launch_sim_waves.script` one has been created in order to manage the standard cells. Basically, the script used for the synthesized circuit, called `launch_sim_synthesis.script`, instead of simply starting the simulation also needs to link the *nangate45* library. It can be done by using the following line:

```
1    vsim work.dlx_testbench -t ns -L /software/dk/nangate45/verilog/qsim10.7c
```

CHAPTER 9

Physical Design

Nowadays, the trend is to build more complex systems (in terms of transistors) in less time (reduce the time-to-market), so there is the need for some powerful tools that allow having optimized ICs. Therefore, the design flow strategy is based on multi-abstraction 3-step iteration:

1. The hardware is described using a Hardware Description Language, as VHDL
2. The Synthesis phase takes as input the abstract model and generates a more detailed model that contains additional information about the timing, power consumption and area. The next step is the Optimization one, which is used to generate an equivalent behaviour circuit and at the same time satisfy some conditions, like timing
3. A post-synthesis simulation is run to check the functional properties of the final model

9.1 Synthesis

The synthesis has been done with intensive script usage; in fact, two scripts have been developed in order to set up the environment, perform the synthesis and clean up all the useless temporary files generated during the process.

The first script is a bash script (refer to Appendix B.1) and, as anticipated before, it is used to set up the environment by coping the `.synopsys_dc.setup` file, copy the library and call the synthesis script suing `dc_shell`. Once the synthesis is over, the bash script removes all temporary folders like `ARCH`, `DOBY`, etc ... and moves the synthesized DLX, in both Verilog and VHDL, and all the generated reports into a specific folder.

The second script, that is run under the `dc_shell` to perform the actual synthesis, executes multiple steps (refer to Appendix B.2):

- Analyze all the .vhd files needed for the DLX
- Elaborate the DLX design, by correctly configuring the generics
- Set the wire model and create a clock, that is the constraint
- Perform the compilation
- Save the synthesized DLX
- Save the timing, area and power report

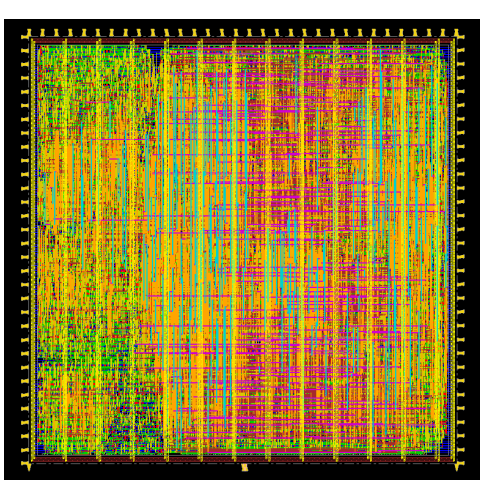
The clock timing, which is set to 2.47 ns, has been selected after many trials and errors in order to find the lowest possible value. Like the one that passes through the adder, critical paths have been reduced using an optimized design, e.g. P4 adder.

All complete reports are detailed in the Appendix C; from the area report, it's possible to observe that the total cell area is 35280.37. It is divided into 19748.90 for the combinational part and 15531.47 for the non-combinational one.

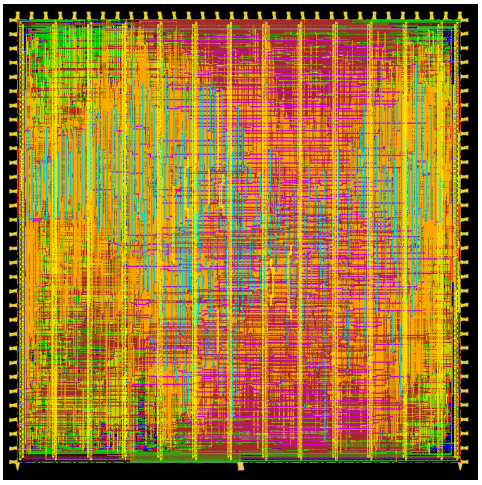
Given a timing constraint, another essential piece of information that can be extracted from the timing report is the *slack*. It represents the time margin that the worst path has; in this way, the clock in the synthesis script can be reduced as much as possible in order to increase performance.

9.2 Place and Route

The Placement step consists of placing all the block and I/O pins within a defined area that is the chip. After this macro-step all the units are placed, and so the routing is performed in order to connect all blocks. After the place and route are both completed, a simulation ensures that everything is correct. The final result can be observed in Figure 9.1b.



(a) DLX processor before routing, only logical connections are present



(b) DLX processor after Place and Route phases

Figure 9.1: DLX Place and Route

In order to obtain a fully placed and routed DLX, many steps are performed:

- Structuring the Floorplan: in this step the Verilog file has been loaded using a global file called `DLX.globals` and a specif amount of internal area is dedicated to the core, while the external one is used for the power rings;
- Power distribution: around the core, two metal rings have been located for distributing the power supply, so both GND and V_{dd}. This is not enough since the power and ground signals must be correctly distributed. For this reason, multiple vertical metal wires, called stripes, has been added to the physical layout. There is a trade-off in the number of stripes since many of them could lead to some problems during the cells routing. Moreover, horizontal wires have been placed to prepare V_{dd} and GND for the standard cells. The result is visible in Figure 9.2;

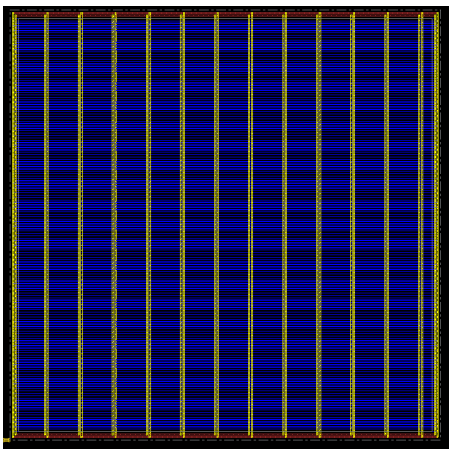


Figure 9.2: Result after placing GND and V_{dd} rings with vertical and horizontal stripes.

- I/O placing: at this point, cells and I/O pins can be placed. Before filling the empty spaces with filler cells, a Post Clock-Tree-Synthesis (CTS) optimization has been performed. The result is visible in Figure 9.1a;
- Routing: the last step is the routing; logical interconnections have been replaced with physical interconnections between cells, considering the available stripes and metal rings. The design is now complete, but a post routing optimization has been performed in order to respect the required timing constraints.

Once the Place and Route step has been done, a timing analysis has been performed using the *Innovus Debug timing* in order to visualize the delay distribution. The paths delay distributions is visible at 9.3.

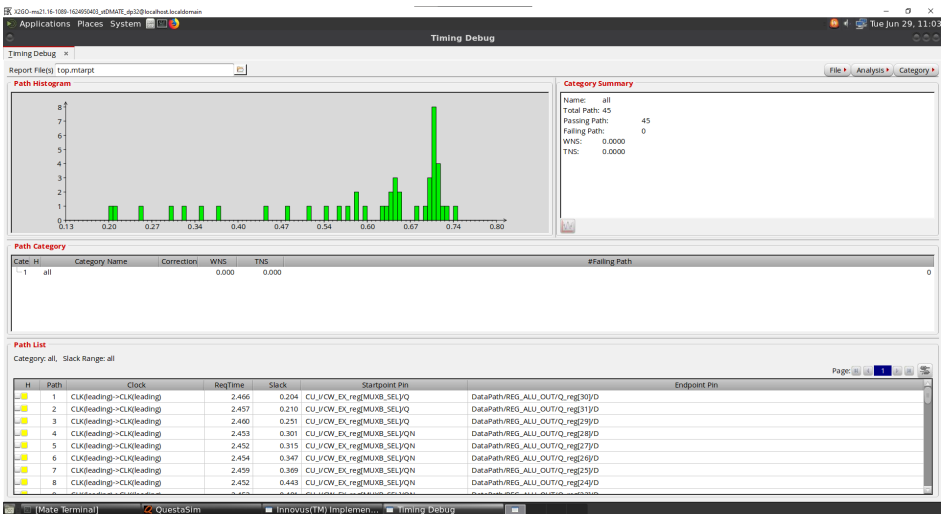


Figure 9.3: Result after placing GND and V_{dd} rings with vertical and horizontal stripes.

Last but not least, before ending the place and route process, design analysis and verification have been performed to ensure that the connectivity and design rules are respected.

CHAPTER 10

Conclusions

To sum up, after the Synthesis and Optimization phase, the peak frequency is 400 MHz. The generated reports highlight that the main power contribution is the dynamic one. This is due to the complex logic that has been implemented in the entire DLX. This DLX implementation has been intensively tested using complex assembly programs, starting from the arithmetic instructions to the sub-routine management.

The entire development has been focused on a modular approach to simplify the integration of new components and possible improvements. For example, further improvements could be the management of exceptions, a cache for the instruction and a branch target buffer.

All team members contributed actively and with a proactive approach to all the challenges this project brought.

Appendix A

A.1 ASM compile script

```
#!/bin/sh

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

./assembler.sh $1

filename=$(basename $1)
filename="${filename%.*}"

assembler.bin/conv2memory ${1%/*}/$filename.bin > DLX_vhd_fully_synthesizable/
    test_bench_and_memory/mems/${filename}.asm.mem
```

Listing A.1: Bash script for generating the .mem file from an .asm one

A.2 Simulation script

```
vcom test_bench_and_memory/TB_TOP_DLX.vhd

vsim -t 10ps work.DLX_TestBench
log -r *

add wave -position insertpoint sim:/DLX_TestBench/CLK

add wave -position insertpoint sim:/DLX_TestBench/DDLX/DECODEhw/op_code sim:/
    DLX_TestBench/DDLX/DECODEhw/ADD_RS1 sim:/DLX_TestBench/DDLX/DECODEhw/
    ADD_RS2 sim:/DLX_TestBench/DDLX/DECODEhw/ADD_WS1

add wave -divider
add wave -position insertpoint sim:/DLX_TestBench/DDLX/DataPath/RF/DATAIN
add wave -position insertpoint sim:/DLX_TestBench/DDLX/DataPath/RF/ADD_WR

add wave -position insertpoint -group PROC_REGS -label R0 sim:/DLX_TestBench/
    DDLX/DataPath/RF/REGS(0)/GLOB_BLK/BLOCK_GLOB/Q
add wave -position insertpoint -group PROC_REGS -label R1 sim:/DLX_TestBench/
    DDLX/DataPath/RF/REGS(1)/GLOB_BLK/BLOCK_GLOB/Q
add wave -position insertpoint -group PROC_REGS -label R2 sim:/DLX_TestBench/
    DDLX/DataPath/RF/REGS(2)/GLOB_BLK/BLOCK_GLOB/Q
add wave -position insertpoint -group PROC_REGS -label R3 sim:/DLX_TestBench/
    DDLX/DataPath/RF/REGS(3)/GLOB_BLK/BLOCK_GLOB/Q
add wave -position insertpoint -group PROC_REGS -label R4 sim:/DLX_TestBench/
    DDLX/DataPath/RF/REGS(4)/GLOB_BLK/BLOCK_GLOB/Q
```

```
19  add wave -position insertpoint -group PROC_REGS -label R5 sim:/DLX_TestBench/
        DDLX/DataPath/RF/REGS(5)/GLOB_BLK/BLOCK_GLOB/Q
20  add wave -position insertpoint -group PROC_REGS -label R6 sim:/DLX_TestBench/
        DDLX/DataPath/RF/REGS(6)/GLOB_BLK/BLOCK_GLOB/Q
21  add wave -position insertpoint -group PROC_REGS -label R7 sim:/DLX_TestBench/
        DDLX/DataPath/RF/REGS(7)/GLOB_BLK/BLOCK_GLOB/Q
22
23  add wave -position insertpoint -group PROC_REGS -label R8 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[31:0]
24  add wave -position insertpoint -group PROC_REGS -label R9 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[63:32]
25  add wave -position insertpoint -group PROC_REGS -label R10 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[95:64]
26  add wave -position insertpoint -group PROC_REGS -label R11 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[127:96]
27  add wave -position insertpoint -group PROC_REGS -label R12 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[159:128]
28  add wave -position insertpoint -group PROC_REGS -label R13 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[191:160]
29  add wave -position insertpoint -group PROC_REGS -label R14 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[223:192]
30  add wave -position insertpoint -group PROC_REGS -label R15 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[255:224]
31  add wave -position insertpoint -group PROC_REGS -label R16 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[287:256]
32  add wave -position insertpoint -group PROC_REGS -label R17 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[319:288]
33  add wave -position insertpoint -group PROC_REGS -label R18 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[351:320]
34  add wave -position insertpoint -group PROC_REGS -label R19 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[383:352]
35  add wave -position insertpoint -group PROC_REGS -label R20 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[415:384]
36  add wave -position insertpoint -group PROC_REGS -label R21 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[447:416]
37  add wave -position insertpoint -group PROC_REGS -label R22 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[479:448]
38  add wave -position insertpoint -group PROC_REGS -label R23 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[511:480]
39  add wave -position insertpoint -group PROC_REGS -label R24 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[543:512]
40  add wave -position insertpoint -group PROC_REGS -label R25 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[575:544]
41  add wave -position insertpoint -group PROC_REGS -label R26 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[607:576]
42  add wave -position insertpoint -group PROC_REGS -label R27 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[639:608]
43  add wave -position insertpoint -group PROC_REGS -label R28 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[671:640]
44  add wave -position insertpoint -group PROC_REGS -label R29 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[703:672]
45  add wave -position insertpoint -group PROC_REGS -label R30 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[735:704]
46  add wave -position insertpoint -group PROC_REGS -label R31 sim:/DLX_TestBench/
        DDLX/DataPath/RF/SEL_BLK/curr_proc_regs[767:736]
```

Listing A.2: Tcl script for staring the simulation and add the registers from R0 to R31

Appendix B

B.1 Bash synthesis script

```
1  #!/bin/sh
2
3  source /software/scripts/init_synopsys_64.11
4
5  cp ../.synopsys_dc.setup ./
6  cp -r ../alib-52 ./
7  dc_shell -no_log -f synthesis.script
8
9  rm -rf rm -rf ARCH/ BODY/ ENTI/ PACK/
10 rm -rf alib-52
11 rm -f *.mr
12 rm -f ./.synopsys_dc.setup
13 rm -f default.svf
14
15 mv DLX_SYN.vhdl ./synthesis_result
16 mv DLX_SYN.v ./synthesis_result
17 mv dlx_sdf.sdf ./synthesis_result
18 mv dlx_sdc.sdc ./synthesis_result
19 mv report_*.txt ./synthesis_result
```

Listing B.1: Bash script for the DLX synthesis

B.2 Synthesis script

```
1  analyze -library WORK -format vhdl {
2          ./000-globals.vhd
3          ./001-globals_CU.vhd
4          ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.m-utils.vhd
5          ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.a-addr_encoder.vhd
6          ./a.b-DataPath.core/a.b.a-windowedRF.core/
               a.b.a.b-address_generator.vhd
7          ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.c-connection_mtx.vhd
8          ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.d-decoder.vhd
9          ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.e-equal_check.vhd
10         ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.f-in_loc_selblock.vhd
11         ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.g-mux.vhd
12         ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.h-nwin_calc.vhd
13         ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.i-reg_generic.vhd
14         ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.l-select_block.vhd
15         ./a.b-DataPath.core/a.b.a-windowedRF.core/a.b.a.n-latch_generic.vhd
16         ./a.b-DataPath.core/a.b.a-windowedRF.vhd
17         ./a.b-DataPath.core/a.b.b-mux2to1.vhd
18         ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.a-logic.vhd
```

```
19              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.b-multiplier.core/
                    a.b.d.b.a-adder.vhd
20              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.b-multiplier.core/
                    a.b.d.b.b-encoder.vhd
21              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.b-multiplier.core/
                    a.b.d.b.c-mux3_1.vhd
22              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.b-multiplier.vhd
23              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.a-carry_generator.vhd
24              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.b-carry_select_block.vhd
25              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/a.b.d.c.d-fa.vhd
26              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/a.b.d.c.e-GG.vhd
27              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.f-mux21_generic.vhd
28              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/a.b.d.c.g-PG.vhd
29              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.h-prop_gen.vhd
30              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.i-prop_gen_generic.vhd
31              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.l-rca_generic.vhd
32              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.core/
                    a.b.d.c.m-sum_generator.vhd
33              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.c-adder.vhd
34              ./a.b-DataPath.core/a.b.d-ALU.core/a.b.d.e-shifter.vhd
35              ./a.b-DataPath.core/a.b.d-ALU.vhd
36              ./a.b-DataPath.core/a.b.e-wRF_CU.vhd
37              ./a.b-DataPath.core/a.b.f-set_comparator.vhd
38              ./a.b-DataPath.core/a.b.g-datamem_ldstr.vhd
39              ./a.b-DataPath.core/a.b.h-addr_mask.vhd
40              ./a.b-DataPath.vhd
41              ./a.d-decode.core/a.d.a-hazard_table.vhd
42              ./a.d-decode.core/a.d.b-comparator.vhd
43              ./a.d-decode.core/a.d.c-ha_counter.vhd
44              ./a.d-decode.vhd
45              ./a.a-CU_HW.vhd
46              ./a-DLX.vhd
47          }
48
49          elaborate DLX -architecture dlx_rtl -library WORK -parameters "IR_SIZE = 32,
                PC_SIZE = 32, RAM_DEPTH = 32"
50
51          set_wire_load_model -name 5K_hvratio_1_4
52          create_clock -name "CLK" -period 2.47 CLK
53
54          compile_ultra -timing_high_effort_script
55
56          write -hierarchy -format vhdl -output ./DLX_SYN.vhdl
57          write -hierarchy -format verilog -output ./DLX_SYN.v
58          write_sdf dlx_sdf.sdf
59          write_sdc dlx_sdc.sdc
60
61          report_timing > report_timing.txt
62          report_area > report_area.txt
63          report_power > report_power.txt
64
65          exit
```

Listing B.2: TCL script for the DLX synthesis

Appendix C

C.1 Area report

```
1
2         ****************************************
3         Report : area
4         Design : DLX_IR_SIZE32_PC_SIZE32_RAM_DEPTH32
5         Version: F-2011.09-SP3
6         Date   : Fri Jul 16 16:59:22 2021
7         ****************************************
8
9         Library(s) Used:
10
11        NangateOpenCellLibrary (File: /home/mariagrazia.graziano/do/libnangate/
              NangateOpenCellLibrary_typical_ecsm.db)
12
13        Number of ports:                        270
14        Number of nets:                       14478
15        Number of cells:                      12492
16        Number of combinational cells:         9127
17        Number of sequential cells:            3360
18        Number of macros:                         0
19        Number of buf/inv:                     1016
20        Number of references:                    66
21
22        Combinational area:       19748.904346
23        Noncombinational area:    15531.473475
24        Net Interconnect area:      undefined  (Wire load has zero net area)
25
26        Total cell area:          35280.377821
27        Total area:                 undefined
```

C.2 Power report

```
1         ****************************************
2         Report : power
3         -analysis_effort low
4         Design : DLX_IR_SIZE32_PC_SIZE32_RAM_DEPTH32
5         Version: F-2011.09-SP3
6         Date   : Fri Jul 16 16:59:26 2021
7         ****************************************
8
9
10        Library(s) Used:
11
12        NangateOpenCellLibrary (File: /home/mariagrazia.graziano/do/libnangate/
              NangateOpenCellLibrary_typical_ecsm.db)
```

```
13
14
15        Operating Conditions: typical   Library: NangateOpenCellLibrary
16        Wire Load Model Mode: top
17
18        Design        Wire Load Model            Library
19        ------------------------------------------------
20        DLX_IR_SIZE32_PC_SIZE32_RAM_DEPTH32
21        5K_hvratio_1_4    NangateOpenCellLibrary
22
23
24        Global Operating Voltage = 1.1
25        Power-specific unit information :
26        Voltage Units = 1V
27        Capacitance Units = 1.000000ff
28        Time Units = 1ns
29        Dynamic Power Units = 1uW    (derived from V,C,T units)
30        Leakage Power Units = 1nW
31
32
33        Cell Internal Power  =   9.5411 mW   (95%)
34        Net Switching Power  = 474.3534 uW    (5%)
35        ---------
36        Total Dynamic Power    =  10.0154 mW  (100%)
37
38        Cell Leakage Power     = 657.4517 uW
39
40
41        Internal         Switching           Leakage            Total
42        Power Group      Power            Power               Power               Power
              (   %    )  Attrs
43        ----------------------------------------------------------------------------------------------------
44        io_pad             0.0000            0.0000            0.0000
              0.0000  (   0.00%)
45        memory             0.0000            0.0000            0.0000
              0.0000  (   0.00%)
46        black_box          0.0000            0.0000            0.0000
              0.0000  (   0.00%)
47        clock_network      0.0000            0.0000            0.0000
              0.0000  (   0.00%)
48        register        9.3982e+03           16.0318        2.6602e+05        9.6802e
              +03  (  90.70%)
49        sequential         12.6095            0.7893        3.1563e+03
              16.5551  (   0.16%)
50        combinational     130.4222          457.5387        3.8827e+05
              976.2318  (   9.15%)
51        ----------------------------------------------------------------------------------------------------
52        Total           9.5412e+03 uW       474.3597 uW     6.5744e+05 nW     1.0673e
              +04 uW
```

C.3 Timing report

```
1
2         ****************************************
3         Report : timing
4         -path full
5         -delay max
6         -max_paths 1
7         Design : DLX_IR_SIZE32_PC_SIZE32_RAM_DEPTH32
```

```
        Version: F-2011.09-SP3
        Date   : Fri Jul 16 16:59:22 2021
        ****************************************

        # A fanout number of 1000 was used for high fanout net computations.

        Operating Conditions: typical   Library: NangateOpenCellLibrary
        Wire Load Model Mode: top

        Startpoint: CU_I/CW_EX_reg[MUXB_SEL]
        (rising edge-triggered flip-flop clocked by CLK)
        Endpoint: DataPath/REG_ALU_OUT/Q_reg[31]
        (rising edge-triggered flip-flop clocked by CLK)
        Path Group: CLK
        Path Type: max

        Des/Clust/Port     Wire Load Model       Library
        ------------------------------------------------
        DLX_IR_SIZE32_PC_SIZE32_RAM_DEPTH32
        5K_hvratio_1_4        NangateOpenCellLibrary

        Point                                                   Incr       Path
        -----------------------------------------------------------------------
        clock CLK (rise edge)                                   0.00       0.00
        clock network delay (ideal)                             0.00       0.00
        CU_I/CW_EX_reg[MUXB_SEL]/CK (DFF_X1)                    0.00 #     0.00 r
        CU_I/CW_EX_reg[MUXB_SEL]/QN (DFF_X1)                    0.07       0.07 r
        U9386/ZN (INV_X1)                                       0.04       0.11 f
        U11439/ZN (NAND2_X1)                                    0.04       0.15 r
        U8853/ZN (NAND2_X1)                                     0.04       0.19 f
        U8919/ZN (INV_X1)                                       0.05       0.24 r
        U9225/Z (BUF_X1)                                        0.11       0.36 r
        U8950/ZN (INV_X1)                                       0.06       0.41 f
        U8949/ZN (XNOR2_X1)                                     0.08       0.49 f
        U8993/ZN (XNOR2_X1)                                     0.07       0.56 f
        DP_OP_751_130_6421/U1157/S (FA_X1)                      0.14       0.70 r
        DP_OP_751_130_6421/U1089/S (FA_X1)                      0.12       0.81 f
        DP_OP_751_130_6421/U1021/CO (FA_X1)                     0.10       0.91 f
        DP_OP_751_130_6421/U952/S (FA_X1)                       0.15       1.06 r
        DP_OP_751_130_6421/U884/S (FA_X1)                       0.12       1.18 f
        DP_OP_751_130_6421/U816/S (FA_X1)                       0.14       1.32 r
        DP_OP_751_130_6421/U748/S (FA_X1)                       0.12       1.43 f
        DP_OP_751_130_6421/U680/S (FA_X1)                       0.14       1.57 r
        DP_OP_751_130_6421/U612/S (FA_X1)                       0.12       1.69 f
        U9220/ZN (OR2_X1)                                       0.07       1.75 f
        U9221/ZN (AOI21_X1)                                     0.06       1.81 r
        U9222/ZN (OAI21_X1)                                     0.04       1.85 f
        U8972/ZN (NAND2_X1)                                     0.03       1.88 r
        U8971/ZN (NAND2_X1)                                     0.03       1.92 f
        U9058/ZN (NAND2_X1)                                     0.03       1.95 r
        U9101/ZN (NAND2_X1)                                     0.03       1.98 f
        U9099/ZN (NAND2_X1)                                     0.03       2.01 r
        U9097/ZN (NAND2_X1)                                     0.03       2.04 f
        U9093/ZN (NAND2_X1)                                     0.03       2.07 r
        U9043/ZN (NAND2_X1)                                     0.03       2.10 f
        U9041/ZN (NAND2_X1)                                     0.03       2.13 r
        U8936/ZN (NAND2_X1)                                     0.03       2.16 f
        U8340/ZN (NAND2_X1)                                     0.04       2.20 r
        U8337/ZN (NAND3_X1)                                     0.03       2.23 f
        U8324/ZN (AND2_X1)                                      0.04       2.28 f
        U8323/ZN (AOI21_X1)                                     0.05       2.32 r
        U8320/ZN (XNOR2_X1)                                     0.06       2.39 r
```

```
70          U8319/ZN (OAI21_X1)                                     0.03       2.42 f
71          DataPath/REG_ALU_OUT/Q_reg[31]/D (DFF_X1)               0.01       2.43 f
72          data arrival time                                                  2.43
73
74          clock CLK (rise edge)                                   2.47       2.47
75          clock network delay (ideal)                             0.00       2.47
76          DataPath/REG_ALU_OUT/Q_reg[31]/CK (DFF_X1)              0.00       2.47 r
77          library setup time                                     -0.04       2.43
78          data required time                                                 2.43
79          --------------------------------------------------------------------------
80          data required time                                                 2.43
81          data arrival time                                                 -2.43
82          --------------------------------------------------------------------------
83          slack (MET)                                                        0.00
```

Appendix D

D.1 Finonacci Code

```
1  addi r24, r0, #6
2
3  call fib
4  j exit
5
6  fib:    ; N is on r8, result in r9
7      addi r16, r0, #1
8      ble r8, r16, ret_1
9
10     subi r24, r8, #1
11     call fib ; n-1
12
13     add r9, r25, r0
14     subi r24, r8, #2
15
16     call fib ; n-2
17
18     add r9, r9, r25
19     ret
20
21 ret_1:
22     ; if n == 0 -> 0
23     ; if n == 1 -> 1
24     beqz r8, zero
25
26     addi r9, r0, #1
27     j out
28
29 zero:
30     add r9, r0, r0
31
32 out:
33     ret
34
35 exit:
36
37     sw 0(r0), r25
38
39     j exit
```

Listing D.1: Fibonacci DLX Assembly example

D.2 Factorial Code

```
1  addi r24, r0, #10           ;0
2
3  ticktmr r16                 ;4
4  call fact                   ;8
5  ticktmr r17                 ;C
6
7  j exit                      ;10
8
9
10 ; R8 => INPUT for current call from prev call
11 ; R9 => OUTPUT for current call towards prev call
12 ; R24 => INPUT FOR NEXT CALL
13 ; R25 => OUTPUT FROM LAST CALL
14 fact:
15     seqi r16, r8, #1        ;14
16     bnez r16, end           ;18
17
18     ; continue rec
19     ; n * call fact(n-1)
20     subi r24, r8, #1      ; n = n-1         ;1C
21     call fact               ;20
22     mult r9, r8, r25        ;24
23     ret                     ;28
24
25  end:
26     addi r9, r0, #1         ;2C
27     ret                     ;30
28
29 exit:
30     sw 0(r0), r25    ; store result
31
32     sub r17, r17, r16 ; total elapsed time (end - start)
33     sw 16(r0), r17   ; store elaps time
34
35 halt:
36     j halt
```

Listing D.2: Factorial DLX Assembly example

D.3 Bubble Sort Code

```
.text

j start_code

swap: ; r8 <- array, r9 <- j
    slli r20, r9, #2
    add r17, r8, r20    ; ptr to array + j
    lw r18, 0(r17)
    lw r19, 4(r17)
    ble r18, r19, not_swap
    ; swap here
    sw 0(r17), r19
    sw 4(r17), r18
    addi r1, r0, #1 ; swapped = true

not_swap:
    ret


bubbleSort: ;R8 <- N, R9 <- array

subi R8, R8, #1 ; r1 <- N--
addi r17, r0, #0    ; index i
fori:
    addi r1, r0, #0     ; swapped = false, global
    addi r19, r0, #0    ; index j
    sub r20, r8, r17    ; r20 <- n-1-i
    forj:
        add r24, r0, r9     ; arg #0 : ptr
        add r25, r0, r19    ; arg #2 : index j
        call swap

        addi r19, r19, #1   ;j++
        seq r18, r19, r20   ; if j == n-i-1
        bnez r18, exit_forj
        j forj


    exit_forj:
        beqz r1, end_sort

        addi r17, r17, #1   ; i++
        seq r18, r17, r8    ; if i == n-1
        bnez r18, end_sort
        j fori


end_sort:
    ret

start_code:
    addi r24, r0, #12 ; N
    lw r25, array_0(r0)
    call bubbleSort

stop:
    j stop


```

```
60
61
62
63 ;###############
64 ; DATA SEGMENT #
65 ;###############
66
67     .data
68
69 code_space:
70     .space 256
71
72 ; The available assembler assigns addresses starting from zero each time
73 ; it finds a new segment. In this way data and text segments are going to
74 ; overlap and their addresses cannot be shared. To avoid this problem we
75 ; need to generate absolute code in which starting address of data segment
76 ; is located after last address of text segment. Therefore we use a trick
77 ; reserving the first part of data segment (code_space) to empty space while
78 ; all useful data structures are defined soon after. In this way, when
79 ; compiling, the text segment would overlap the data segment just for this
80 ; initial empty space so that as result we have the compiled code followed
81 ; by useful data addresses. Obviously the size of code_space has to be enough
82 ; to contain all compiled code (256 byte are more than enough).
83 ; array to be sorted
84
85 array:
86     .word 3,18,-12,24,17,1,7,3,-8,20,100,-33
87
88 array_0:
89     .word array
```

Listing D.3: Bubble Sort DLX Assembly example

D.4 Advanced Branch Code

```
1
2  a:
3      addi r1, r0, #1 ; 1
4      addi r3, r0, #4 ; 1
5      ble r3, r1, c ; 8
6      bge r1, r3, c
7      beqz r1, c
8
9  b:
10     addi r5, r0, #12312 ;10
11     blt r3, r1, d
12     bgt r3, r1, d
13
14 c:
15     addi r6, r0, #8 ;18
16     jal a
17
18 d:
19     bge r1, r1, e
20     j b
21
22 e:
23     addi r7, r0, #7
24     bge r0, r0, a
```

Listing D.4: Advanced Branch Code DLX Assembly example

D.5 Complete Arithmetic Instructions Code

```
1
2       nop
3       addi r1,r0,#2         ;2           ;2
4       subi r2,r1,#1         ;1           ;1
5       addi r3,r1,#-4        ;-2          ;-2
6       subi r4,r3,#-1        ;-1
7       add r7,r1,r2          ;3
8       sub r8,r5,r6          ;0
9       sge r10,r1,r2         ;1
10      sge r11,r2,r1         ;0
11      sge r12,r1,r1         ;1
12      sle r13,r1,r2         ;0
13      sle r14,r2,r1         ;1
14      sle r15,r1,r1         ;1
15      sne r16,r1,r1         ;0
16      sne r17,r1,r2         ;1
17      seq r18,r1,r2         ;0
18      seq r19,r1,r1         ;1
19      sgt r10,r2,r1         ;0
20      sgt r11,r1,r2         ;1
21      slt r12,r2,r1         ;1
22      slt r10,r1,r2         ;0   2 < 1
23      sgeu r10,r1,r3        ;0   2 >= 654..
24      sgeu r11,r3,r1        ;1
25      sgtu r12,r1,r3        ;0
26      sgtu r13,r3,r1        ;1
27      sltu r14,r3,r1        ;0
28      sltu r15,r1,r3        ;1
29      sgei r16,r1,#4        ;0
30      sgei r17,r1,#1        ;1
31      slei r18,r1,#0        ;0
32      slei r19,r1,#2        ;1
33      snei r10,r1,#1        ;1
34      snei r11,r1,#2        ;0
35      seqi r12,r1,#0        ;0
36      seqi r13,r1,#2        ;1
37      sgti r14,r1,#3        ;0
38      sgti r15,r1,#1        ;1
39      slti r16,r1,#1        ;0
40      slti r17,r1,#3        ;1
41      sgeui r10,r1,#4       ;0
42      sgeui r10,r1,#-4      ;0
43      sgeui r11,r1,#1       ;1
44      sgtui r12,r1,#3       ;0
45      sgtui r13,r1,#1       ;1
46      sltui r14,r1,#1       ;0
47      sltui r15,r1,#3       ;1
48      addi r12,r0,#5        ;5
49      or r13,r11,r12        ;5
50      ori r14,r12,#65535    ;0000FFFF
51      and r15,r14,r2        ;1
52      andi r16,r14,#1       ;1
53      sll r17,r16,r2        ;2
54      slli r18,r16,#1       ;2
55      srl r19,r16,r2        ;0
56      srli r20,r16,#1       ;0
57      sra r21,r12,r2        ;2
58      srai r22,r12,#1       ;2
59      mult r23,r1,r5        ;0
```

```
60      xor r24,r1,r1        ;0
61      ;xori r25, r1, #0xf   ;14
62
63      addi r6, r0, #5      ;5
64      addi r1, r0, #2      ;2
65      ror r30, r6, r1       ;0100 0000 0000 0000 0000 ... 0001
66      addi r7, r0, #5      ;5
67      rol r29, r7, r1      ;0000 0000 0000 0000 ... 0001 0100
68
69      addui r28, r0, #0xffff   ; 0x0000ffff
70      subui r28, r0, #0xffff   ; 0xffff0001
71
72      lhi r28, #-40            ; 0xfffd8000
73
74      sw 0(r0), r14
75      lh r27, 2(r0)        ;ffffffff
76      lhu r26, 2(r0)       ;0000ffff
77
78      nop
79  halt:
80      j halt
```

Listing D.5: Complete Arithmetic Instructions DLX Assembly example

D.6 Load & Store Code

```
1
2   myloop:
3       addi r1, r0, #4                      ;0
4       addi r8, r0, #10                     ;4
5       sw 4(r1), r8                        ;8
6
7       addi r9, r0, #-10                    ;C
8       sh 5(r1), r9                        ;10
9       lb r10, 5(r1)                       ;14
10      lbu r11, 5(r1)                      ;18
11      lhu r11, 5(r1)                      ;1C
12      addi r10, r10, #1                    ;20
13
14      lbu r2, 5(r1)                        ;24
15      sh 20(r1), r2                       ;28
16
17      lh r3, 20(r1)                        ;2C
18      lhi r3, #9                           ;30
19
20      addi r4, r5, #15                     ;34
21      addi r7, r0, myloop                 ;38
22
23      jal myloop                          ;3C
24                                          ;40
```

Listing D.6: Load & Store DLX Assembly example

Appendix E

E.1 Windowing Register File

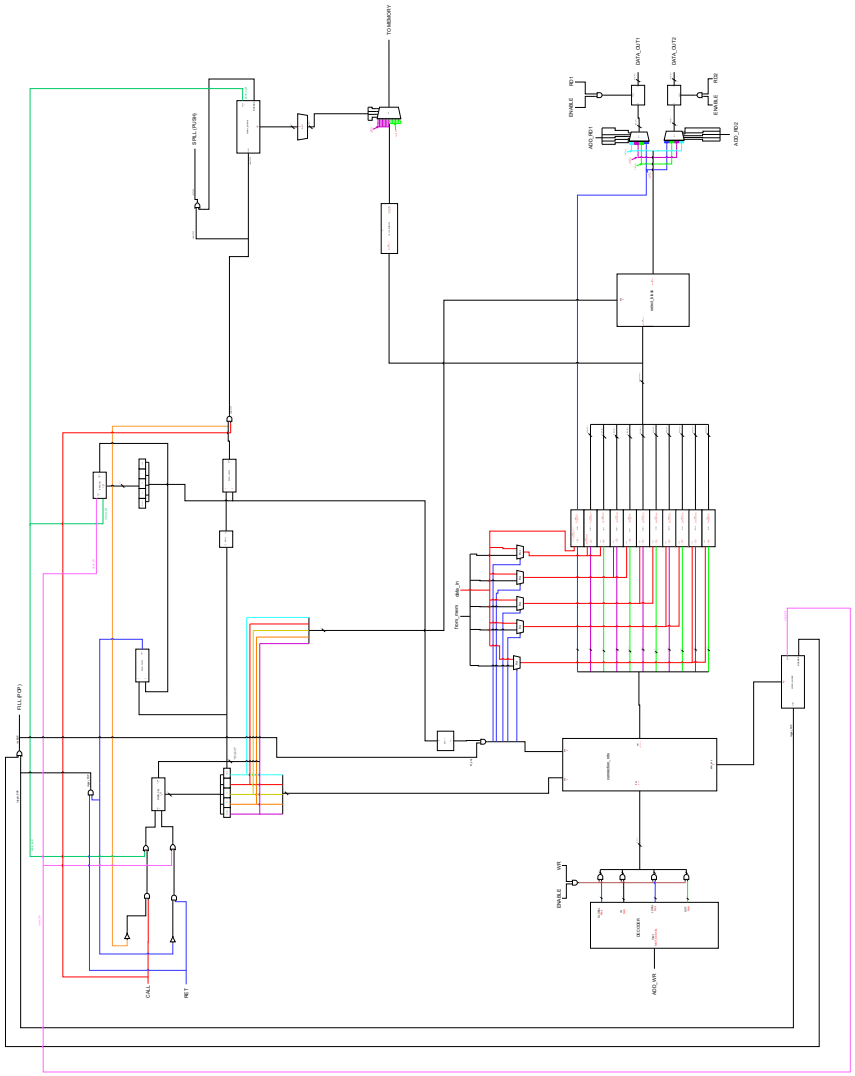


Figure E.1: Complete and exhaustive Windowing Register File