Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: Group 36

Cavallone Fabrizio, Marossero Jessica, Tartaglia Federico

September 19, 2023

Contents

CHAPTER 1

Introduction

The DLX (”Deluxe”) processor is a highly efficient, fully pipelined computing unit with a RISC (Reduced Instruction Set Computer) design. It draws its architectural inspiration from notorious predecessors, including the AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260, and it is symbolized by the number 560, which corresponds to ”DLX” in Roman numerals.
DLX adopts the Harvard Architecture model, characterized by the separation of instruction and data storage into distinct memory units and this architectural choice enables simultaneous execution of instruction-fetching and data-related operations. It has a 32-bit Load/Store architecture with 32 integer registers from R0 to R31 and it has two types of addressing modes:

- Immediate Mode: Utilized in operations as ”ADDI R1, R2, #3 ” where a single immediate operand is included in the operation itself.
- Displacement Mode: Utilized in operations as ”LW R1, 20(R2)” where memory addressing involves adding an immediate value to the content of a register.

The processor is divided into these following key components:

- **Datapath:** This block connects both external memories and the control unit (CU). It receives internal control signals from the CU and data from the IRAM and DRAM. As for the MIPS architecture, it is managed through a 5-stage pipeline composed by FETCH, DECODE, EXECUTE, MEMORY, and WRITE-BACK phases.
  In addition, two further units can be found internally:
  - **Forwarding Unit :** This unit reads all instructions in execution from the second pipeline stage to the last stage. It can detect data dependencies and, in case of a dependency, it can manage the datapath to forward data from one pipeline stage (from the end of the exe stage or mem stage) to another, instead of waiting the writing of the data into the register file.
  - **Hazard Unit :** The HU operates based on instructions in the second and third pipeline stages. It has the capability to stop certain pipeline stages and eliminate fetched instructions to prevent various types of hazards.

- **Control Unit :** The CU generates appropriate control outputs based on the received instructions.

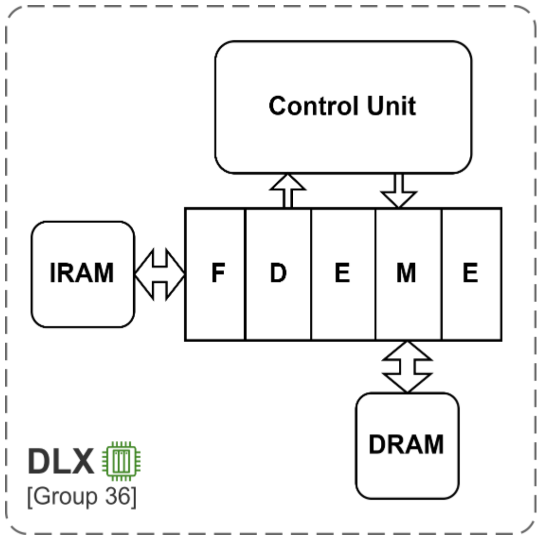


Figure 1.1: DLX architecture

DLX executes three distinct categories of instructions:

- R-Type Instructions: These instructions involve registers, featuring two source registers and one destination register. They primarily encompass ALU (Arithmetic Logic Unit) register operations.

OPCODE(6)	RS1(5)	RS2(5)	RD(5)	FUNC(11)

Table 1.1: R-Type instructions.

- I-Type Instructions: In these instructions, one of the operands is an immediate value, with the option to include a source register and a destination register. I-Type instructions encompass Immediate ALU operations and Branch instructions.

OPCODE(6)	RS1(5)	RD(5)	IMMEDIATE(16)

Table 1.2: I-Type instructions.

- J-Type Instructions: These instructions pertain to Jump and Jump&Link operations, allowing for non-linear program flow control.

OPCODE(6)	FUNC(26)

Table 1.3: J-Type instructions.

1.1 Advanced features

Our processor maintains the base DLX architecture while incorporating a range of optional enhancements, including:

- **Expanded Instruction Set Architecture:** We have expanded the instruction set architecture to insert JR and JALR instructions, unsigned operations, mov operation, load and store operations, mult and multi operations and other operations described better later.
- **Hardwired Control Unit:** The processor integrates a dedicated control unit with hardwired control logic for improved efficiency.
- **Advanced Datapath with Reduced Branch Delay:** We've enhanced the datapath with advanced features that minimize branch delay.
- **Efficient ALU Logic:** Our Arithmetic Logic Unit (ALU) logic is designed for efficiency and is composed a Pentium 4 adder, a Logic Unit, a Comparator and T2 Shifter.
- **Efficient Booth Multiplier** It employs Booth's algorithm to efficiently perform binary multiplication and it is especially useful for signed number multiplication and for speeding up arithmetic operations.
- **Forwarding Capability:** The processor incorporates forwarding capabilities for all the instructions to facilitate data transfer between pipeline stages.
- **Hazard Detection and Stall Management:** Robust hazard detection mechanisms and stall management are integrated to ensure a normal flow of the "pipeline" operations while mitigating potential issues.
- **Advanced script:** Script that automates all the process of the simulation and the synthesis phases.

1.2 Instruction Set Architecture

Instructions implemented for the basic version DLX:

Limited ISA			
R-type		**General Instructions**	
Operation	**Func**	**Operation**	**Opcode**
ADD	r,0x20	ADDI	i,0x08
SUB	r,0x22	SUBI	i,0x0a
AND	r,0x24	ANDI	i,0x0c
OR	r,0x25	ORI	i,0x0d
XOR	r,0x26	XORI	i,0x0e
SGE	r,0x2d	SGEI	i,0x1d
SLE	r,0x2c	SLEI	i,0x1c
SLL	r,0x04	SLLI	i,0x14
SNE	r,0x29	SNEI	i,0x19
SRL	r,0x06	SRLI	i,0x16
		BEQZ	b,0x04
		BNEZ	b,0x05
		J	j,0x02
		JAL	j,0x03
		LW	l,0x23
		SW	s,0x2b
		NOP	n,0x15

Table 1.4: Limitated ISA.

Instructions added for the pro version:

Extended ISA for PRO DLX			
R-type		**General Instructions**	
Operation	**Func**	**Operation**	**Opcode**
MOV	r2,0x2f	**MOVI**	i1,0x33
MULT	r,0x2e	**MULTI**	i,0x32
ADDU	r,0x21	ADDUI	i,0x09
SUBU	r,0x23	SUBUI	i,0x0b
SRA	r,0x07	SRAI	i,0x17
SEQ	r,0x28	SEQI	i,0x18
SLT	r,0x2a	SLTI	i,0x1a
SGT	r,0x2b	SGTI	i,0x1b
SLTU	r,0x3a	SLTUI	i,0x3a
SGTU	r,0x3b	SGTUI	i,0x3b
SLEU	r,0x3c	SLEUI	i,0x3c
SGEU	r,0x3d	SGEUI	i,0x3d
		LHI	i1,0x0f
		JR	jr,0x12
		JALR	jr,0x13
		LB	l,0x20
		LH	l,0x21
		LBU	l,0x24
		LHU	l,0x25
		SB	s,0x28
		SH	s,0x29

Table 1.5: Extended ISA.

The **instructions in bold** inserted in Table 1.2 have also been added to the perl file.

1.3 Datapath

The block diagram of the entire datapath is presented in Figure 1.2. The datapath is organized into five distinct pipeline stages:

1. **Instruction Fetch (IF):** In this initial stage, the primary objective is to fetch instructions from the IRAM. This stage includes the Program Counter (PC) register, an adder responsible for incrementing the PC value by 4 and then the corresponding NPC register, a connection to the IRAM (external to the datapath) and the IR register, Furthermore, a 2-to-1 mux has been inserted at the PC input to choose whether to pass the NPC or the PC due to a branch or jump. For detailed operation, refer to Chapter 2.1.

2. **Instruction Decode (ID):** In the second stage the register file is accessed and the registers content read. The 16 or 26-bit immediate is extended as Signed/Unsigned and the correct one between the two is selected using a MUX. In this stage are also included all the branching features, anticipated with respect to the standard DLX. For detailed operation, refer to Chapter 2.2.

3. **Execution (EXE):** In the third stage, arithmetic and logic operations take place. This is achieved through the Arithmetic Logic Unit (ALU), as discussed in Chapter 2.3.1, or via the

multiplier in cases involving MULT or MULTI operations, as discussed in Chapter 2.3.2. A MUX selects between the two outputs, another MUX determines whether the first operand is sourced from the first register or the NPC and another MUX determines whether the second operand is sourced from the second register or from an immediate value. For detailed operation, refer to Chapter 2.3.

4. **Memory Access (DRAM):** The fourth stage is dedicated to reading from and writing to the DRAM. Just as with the IRAM, the DRAM is conceptually placed here, although it remains an external component connected to the datapath. The memory's address input and data input are connected to the operation output and the second register output, respectively. The data output is more complex and involves a set of bitwidth extenders controlled under the CU's command, selecting the appropriate extender based on the requested load/store type (byte/half-word/word, signed/unsigned). For detailed operation, refer to Chapter 2.4.

5. **Write-Back (WB):** The final stage is where the Register File (RF) is updated. The only additional hardware introduced here is a MUX that chooses the data to be written, selecting from the DRAM output or the logic/arithmetic output. The PC+4 is propagated from the IF stage through dedicated registers for each pipeline stage and is employed when a subroutine call occurs, necessitating the storage of the return address in register 31, which serves as the link register.

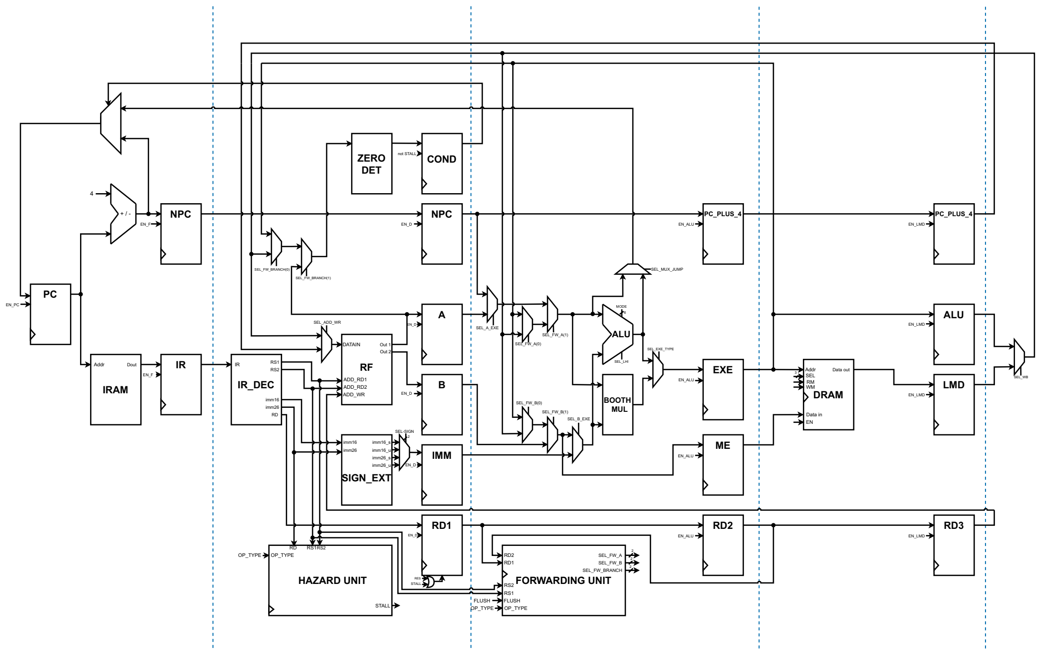


Figure 1.2: Datapath structure.

CHAPTER 2

Architectural design of DLX

2.1 Fetch Unit

The fetch unit is composed and managed by a generic parameter (I_SIZE), which represents the instruction size (in this case, 32 bits). **Input Ports:**

- *DOUT*: data output from the IRAM.
- *PC*: program counter.
- *EN_PC*: program counter enable signal.
- *EN_F*: fetch enable signal.
- *EN_D*: decode enable signal.
- *SEL_BRANCH*: branch selection signal from forwarding unit.
- *CLK*: clock signal.
- *RES*: reset signal.

Output Ports:

- *ADDR*: data output from the IRAM.
- *NPC*: the next program counter(PC+4).
- *IR*: instruction register output.

Structural Architecture

The Structural architecture of the fetch unit is actually implemented with some components like RCA_GEN (a binary adder), FD_GEN (a register), and MUX21_GEN (a 2-to-1 multiplexer). These components are used for tasks such as adding values, registering signals and selecting from different options.
The RCA adder is used to sum the current value of the program counter (PC) with a constant value (4) to calculate the next program counter (next_pc), the registers are used to record various signals and to stall and divide the pipeline from the IF to DEC stage.
The 2-to-1 multiplexer has been inserted at the PC input to choose, thanks to the sel branch, whether to pass the NPC or the PC due to a branch or jump. The implementation of this multiplexer is due to the fact that a branch always not-taken policy is assumed, so, the DLX proceeds normally with the pipeline flow and when a branch/jump is fetched, the new pc calculated by the ALU will be chosen and the instructions that had already entered the pipeline flushed.

2.2 Decode Unit

The "DECODE_UNIT" is a configurable unit that can be adapted for various bit widths, it is composed and managed by two generic parameters: NBIT with a default configuration of 32 bits that represent the instruction and data size and OPCODE_SIZE that represents the OPCODE length of the DLX. The input and output ports of the unit are:

Input Ports:

- *IR*: instruction register, that contains the instruction to be decoded.
- *NPC*: the next program counter (PC+4) from the Fetch Unit.
- *DATAIN*: data input for the Register Rile (RF).
- *ALU_DATA*: ALU (Arithmetic Logic Unit) data output, used for the forwarding.
- *SEL*: select signal for the immediate.
- *EN_D*: decode enable signal.
- *EN_COND*: cond enable signal.
- *EN_RF*: register file enable signal.
- *CLK*: clock signal.
- *RES*: reset signal.
- *RD1*: read data 1 enable signal.
- *RD2*: read data 2 enable signal.
- *WR*: write enable signal.
- *SEL_ADD_WR*: select address write signal, for the register file input data.
- *SEL_FW_BRANCH*: select forward branch signal.
- *PC_PLUS_4*: the program counter plus 4, used for JR and JALR operations.
- *ADDRESS_WRITE*: address write signal for register file.

Output Ports:

- *SEL_MUX_PC*: the select multiplexer PC signal, the "COND" output.
- *A*: data A.
- *B*: data B.
- *IMM*: decoded immediate value on 32 bit.
- *NPC_OUT*: Next Program Counter.
- *RD_OUT*: Register Destination (clocked).
- *RS1*: decoded RS1.

- *RS2*: decoded RS2.
- *RD*: decoded RD (not-clocked).

Structural Architecture

The structural architecture of the decode unit is actually implemented with some general components like FD_GEN (a register with synchronous reset), MUX21_GEN (a 2-to-1 multiplexer), MUX41_GEN (a 4-to-1 multiplexer) and some special components like the ZERO_DETECTOR, COND, IR_DECODER, Register File and the SIGN_EXTENSION.
All the normal registers used in this stage are driven by the same enable signal EN_D from the Control Unit. The RD1 register has as reset signal not only the RES like all the other registers in this Unit but it can reset itself also when a stall occours to eliminate the current data and avoiding a bad lecture from the forwarding unit.
These components are used for tasks such as registering signals, selecting from different options and compute special values.
In particular the COND "register" is used to determine if a branch is taken or not taken assuming a not-taken branch prediction policy based on the result of the ZERO_DETECTOR and the required CONDITION from the OPCODE. Then it generates a synchrous signal (FLUSH) if the new current operations in the pipeline have to be removed from the pipeline. This signal is also used to drive the MUX21 in the fetch unit to select if it is necessary to jump. In this way the output occours in the execution stage and the new program counter is computed without adding resources that consume area. It is important to notice that as enable the cond register has the "NOT STALL" signal because it has to generate the the jump enable signal (and consequently the flush) only if a stall doesn't occour in that cycle.
The IR_DECODER is the component used to retrieve from the instruction register the several signals used in this unit like the addresses for the register file and the others like the Forwarding Unit and the Hazard Detection Unit. The other stuff of the IR_DECODER is to make a distinction in the various operations assigning different data at its output.
The SIGN_EXTEND component is used to take in input the two immediate value and perform a zero-extension or a sign-extension on these to have the data on 32-bit. The four outputs are sent to the 4-to-1 mux (driven by the Control Unit) that select the correct output for each operation.
The last special component is the register file make by an array of cells (memory-like) that contains 32 registers of 32-bit each. So it has 5-bit address signals and a 32-bit datain and output signals. The register file used in our DLX has two read ports and one write port.
There are two 2-to-1 muxes used to provide the forwarding stuff on the branch operations: the first one (driven by the LSB of the SEL_FW_BRACH signal) selects above the execution stage output and the memory stage output; the second (driven by the MSB) selects if the forwarding is necessary or leave the A signal from the register file as normal execution.
The other 2-to-1 mux is used to select between the normal DATAIN for the register file and the PC+4 for the JR and JALR instructions and it is driven by the CU
In conclusion the Decode Dnit is the unit that manages the data used in the next pipeline stage (Execution Unit), performs the branch operations and generates the signals for the forwarding and hazard unit.

2.3 Execution Unit

The "EXECUTION_UNIT" is a configurable unit that can be adapted for various bit widths, it is composed and managed by two generic parameters: NBIT with a default configuration of 32 bits that represent the data size and NBIT_PER_BLOCK that represents the number of bit in each block of the P4_ADDER. The input and output ports of the unit are:

Input Ports:

- *NPC*: the next program counter.
- *A*: signal A.
- *B*: signal B.
- *IMM*: the immediate value.
- *SEL_A*: the select signal for A.
- *SEL_B*: the select signal for B.
- *SEL_EXE_TYPE*: the select signal for the execution type.
- *ALU_MODE*: the ALU mode.
- *EN_ALU*: the execution enable signal.
- *CLK*: the clock signal.
- *RES*: the reset signal.
- *RD*: the Register Destination signal.
- *SEL_MUX_JUMP*: the select signal for the jump multiplexer.
- *SEL_LHI*: the select signal for LHI (Load High Immediate) operation.
- *SEL_FW_A*: the select signal for forwarding to A.
- *SEL_FW_B*: the select signal for forwarding to B.
- *MEM_DATA*: memory data.

Output Ports:

- *ADDR_DRAM*: the address to the DRAM.
- *RD_OUT*: the RD (Register Destination) output.
- *EXE_OUT*: the execution result output.
- *PC_PLUS_4*: the program counter plus 4.
- *PC_OUT*: the program counter output.

Structural Architecture

The Structural architecture of the decode unit is actually implemented with some general components like FD_GEN (a register with synchronous reset), MUX21_GEN (a 2-to-1 multiplexer), and some special components like the ALU and the BOOTHMUL.
The Execution purpose is to generate the operations result for all the instruction types including both the final result and the computation of the program counter used in branch/jump operations.
The two main parts (discussed above) are the ALU and the multiplier, each with two inputs operands and the output operand on 32-bit. The ALU operates on the two input of 32 bit each and the mode signal (6-bit) and the SEL_LHI sent by the Control Unit to select the operation execution type. On the other hand the Multiplier used two low half-word as inputs to generate the result without overflow. This convenction is assmumed to have the signed result on a single register and to make no distinction between register-register multiplication and the register-immediate one.
There are several 2-to-1 muxes for different stuff: the first one (MUX_A_NPC) is used to select between A and the NPC inputs using SEL_A signal driven by the CU, the muxes of the forwarding part for operand A(MUX_FW_ALU_MEM_A and MUX_FW_FIN_A): the first choice between the previous operation execution output and the Memory stage output data using the LSB of SEL_FW_A generated by the forwarding unit. The output of this one is sent to another mux that sent in output the normal operands or the forwarding one and it is driven by the MSB of the same control signal. The same two muxes for forwarding for A are replied for the operand B of the ALU and Multiplier, they are also driven by the forwarding unit with SEL_FW_B.
The output of the forwarding unit for the operand B is sent in the register ME used in the Store operations as datain and finally in the mux that selects between that signal and the immediate value using SEL_B generated by the CU as driver.
The last mux (MUX_JUMP) is used in jump operations and it selects the next program counter: the choice is between the input of the ALU in case of JR and JALR (after the forwarding) and the output of the ALU if it has to be computed using the previous program counter and the immediate value depending on its selector sent by CU.
In conclusion the Execution Unit is engaged to compute all the operations that the DLX can perform and is the only unit where computation unit are present, apart from fetch unit where the adder to compute the plus 4 is implemented. It is important to notice that all the operations that required RS1 and RS2 have the possibility to forward their signals, including the input data for the DRAM memory in the store operations.

2.3.1 ALU

As discussed before, the ALU is the unit engaged to compute all the operations apart from multiplications. It receives as inputs the two operands, the MODE signal to select the operation type and the selector for LHI operation. This selector is not included in the MODE vector since there is not the same version in the R-type operations. A schematic is present in figure 2.1.
The ALU is composed by different submodules:

- *ALU_DECODER.*
- *P4_ADDER.*
- *T2_SHIFTER.*
- *COMPARATOR.*
- *LOGICALS.*

In addition to these modules, also other general components are present: five 2-to-1 muxes and an inverter. The first one (B_ADDER) in engaged to select between B and not B generated by the generic inverter. In a further improvement of the DLX those two components can be substituted with a bitwise xor with one input always equal to cin and B on the other port.
Other muxes(the two before the P4 Adder) are used to add zero if a movi or a mov operation is present and thei are driven by the corresponding sel_movi and sel_mov signals. Finally, the two muxes before the shifter are used to invert B with a constant value (16) and A with B if an LHI operation is needed. These two are driven by the ALU input SEL_LHI.

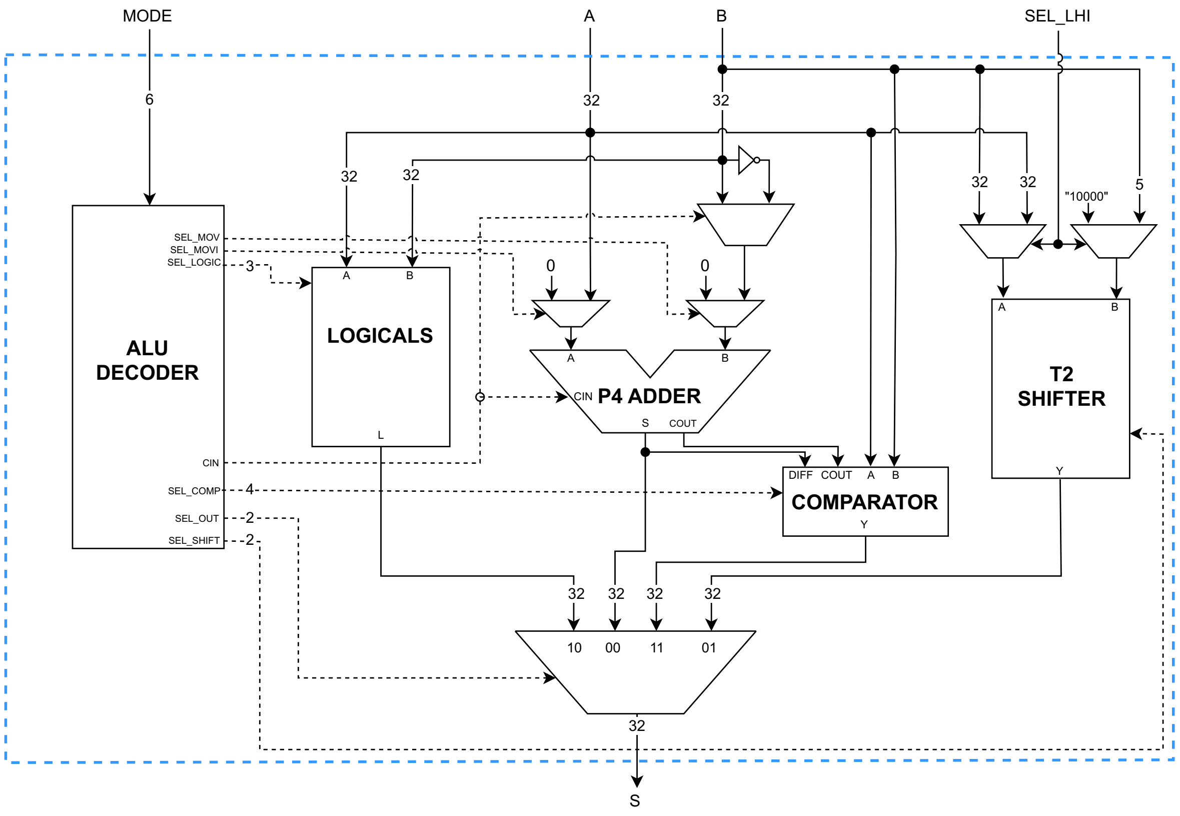


Figure 2.1: Alu structure

ALU Decoder

The Decoder is used to generate the control signals for all the blocks (dashed lines in 2.1). It is basically a Look-Up table that assigns the different outputs for each MODE signal in input. This signal is the ALU_Op_code and is sent by the Control Unit during the execution stage.
The signals generated are:

- *SEL_MOV*: for the mux used in the mov operation.
- *SEL_MOVI*: for the mux used in the movi operation.
- *SEL_LOGIC*: for the Logic unit.
- *SEL_COMP*: for the Comparation unit.
- *SEL_SHIFT*: for the Shift unit.
- *SEL_OUT*: for the Preferred output from the different units.
- *cin*: used to select between B or not-B and to perform subtractions.

For example if a substraction is needed the signals generated are cin = "1", SEL_MOV and SEL_MOVI = "0" and the SEL_OUT = "00" leaving all the other ALU control signals at default value zero.

P4 Adder

The structure is shown in Figure 2.2. It is composed by two main components: the sum geneartor and the carry generator. The sum generator is made by several carry-select-loke-blocks and a sparse tree for the carry generation. Each carry-select-block generates a set of sum bits and an outgoing carry. One adder assumes that the incoming carry into the group is 0, while the other assumes that it is 1. Thus the two adders generate in parallel the results. When the incoming carry is assigned, its final value is selected out by means of two multiplexers, one for the sum and the other for the carry. The advantage in using this structure comes when the m-bit adder is divided in k groups of n-bit carry select adders to have a lower delay. The sparse tree carry generator is composed by three main components: G block, PG block and PG network for the generation and propagation of the carry in each stage.

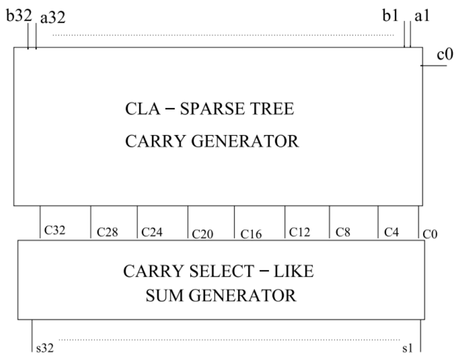
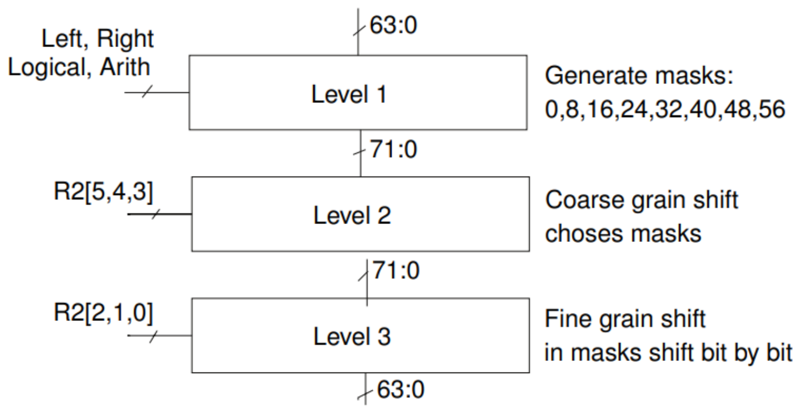


Figure 2.2: P4 Adder general organization

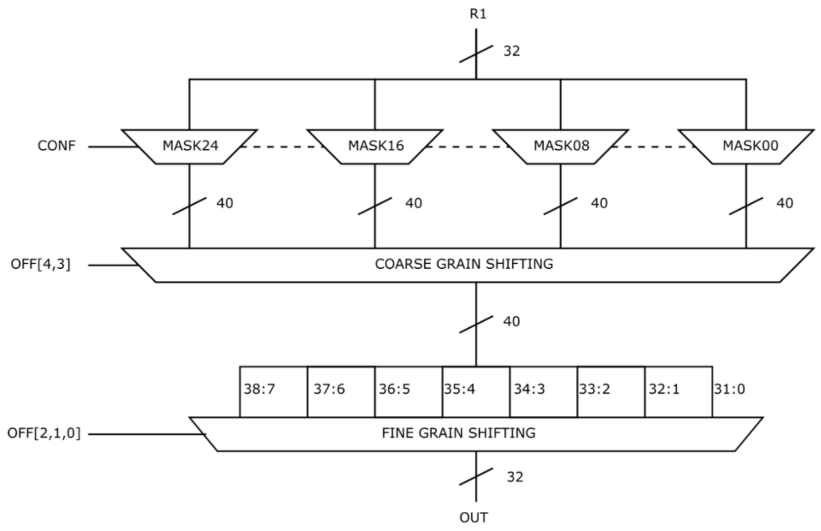
It can perform signed operations since the sparse-tree carry generator uses cin as parameter in the first PG block of the PG_network There is no overflow detection, this has to be taken into account where the DLX is used to perform operations like additions or subtractions.

T2 Shifter

The operand to be shifted is R1, while R2[4,3,2,1,0] is used for defining the shift amount. Other inputs define if the shift should be right, left, arithmetical (fill with MSB) or logical (fill with 0). The general shifter block diagram is in figure 2.3.
In our DLX it is only a 32-bit version, while in the figure is shown the real block-diagram used in the T2. It is organized in three levels. The first consist in preparing 4 possible ”masks”, each already shifted of 0,8, 16,....24, left or right depending on the configuration. The second level performs a coarse grain shift, that is it chooses among the 4 masks the nearest to the shift to be operated. For example, in case a shift by 4 is needed, then mask 0 is chosen, while if a shift by 11 is needed then mask 8 is selected. The choice is operated using bit 4 and 3 of operand R2. Finally the third level receives such a mask and implements the real shift according to bits 2,1 and 0 of operand R2.
The supported operations by the shifter are: shift right arithmetic, shift right logic and shift left. there are present also rotate operations but they are not used by the instruction set of the DLX. This can be a possible future improvement.



(a) Block diagram



(b) Implementation

Figure 2.3: T2 shifter block diagram and implementation

Comparator

The comparator in the ALU is a comparator able to perform both signed and unsigned comparations. It receives in input the two values to be compared, their difference with the corresponding cout generated by the P4 adder and The 4-bit select signal for the type of comparation to be performed. As output it has the requested result on 32-bit (zero-extended). For simplicity the 32-to-1 NOR will be represented by Z. The implementation is a fully-structural with the following relations:
Signed:

$$A < B \Rightarrow ((Z + \overline{COUT}) \oplus A(31) \oplus B(31)) * \overline{Z}$$
$$A \leq B \Rightarrow (Z + \overline{COUT}) \oplus A(31) \oplus B(31)$$
$$A > B \Rightarrow (COUT \oplus A(31) \oplus B(31)) * \overline{Z}$$
$$A \geq B \Rightarrow COUT \oplus A(31) \oplus B(31)$$
$$A = B \Rightarrow Z$$
$$A \neq B \Rightarrow \overline{Z}$$

Unigned:

$$A < B \Rightarrow (Z + \overline{COUT}) * \overline{Z}$$
$$A \leq B \Rightarrow Z + \overline{COUT}$$
$$A > B \Rightarrow COUT * \overline{Z}$$
$$A \geq B \Rightarrow COUT$$

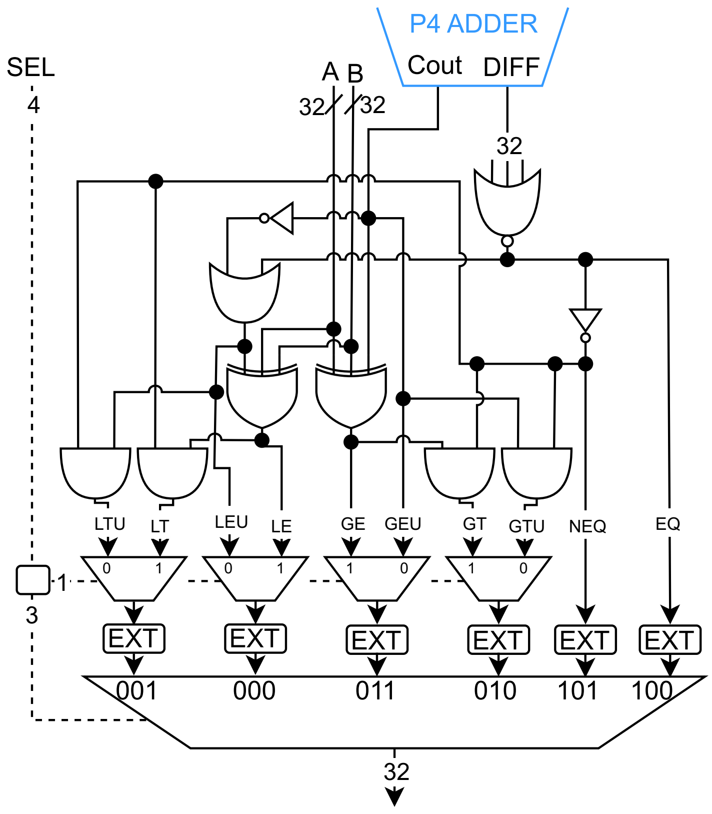
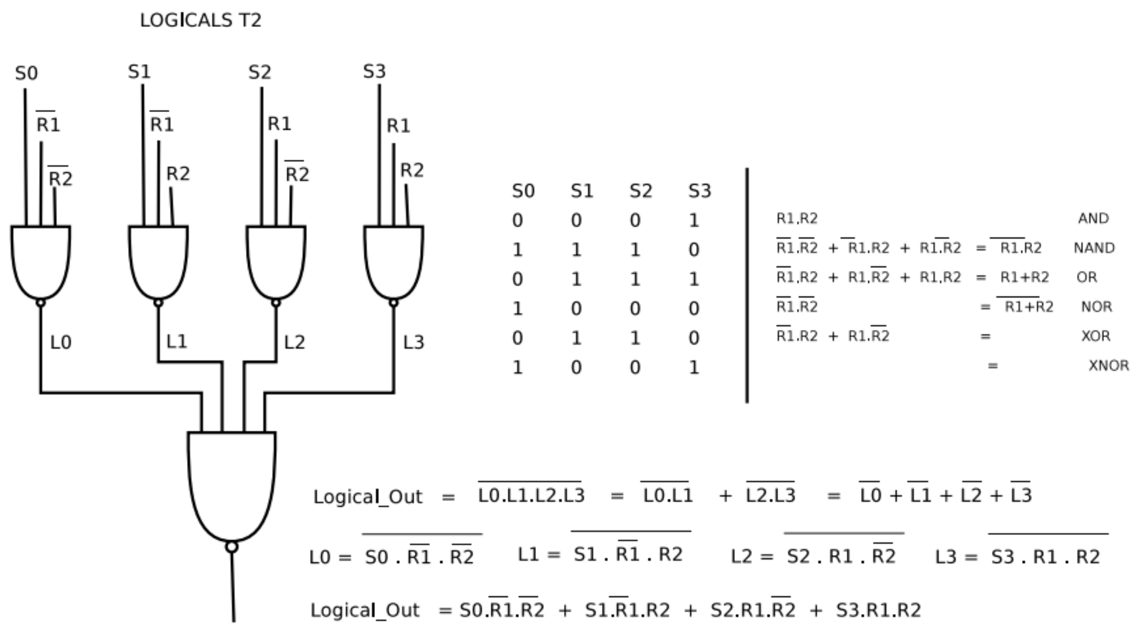


Figure 2.4: Comparator structure

Logical

The logic unit is the T2 logic unit. It is implemented with two NAND gates (nand) level: the first one has four 3-inputs nands, each input is 64-bits wide; the second level has a 4-inputs nand, whose inputs are the outputs of the first level gates. The block scheme is shown in figure 2.5.
For simplicity in the implementation is used only one signal between S1 and S2 since they assume always the same behaviour. Each first level gate has a select input (extended to 64 bits), called selectN, with N=0,1,2,3. The other two inputs are the R1 and R2 operands, or their negative value. In order to compute one of the logical instructions, the select signals are properly activated as follow:



S0	S1	S2	S3		
0	0	0	1	$R1.R2$	AND
1	1	1	0	$\overline{R1}.\overline{R2} + \overline{R1}.R2 + R1.\overline{R2} = \overline{R1.R2}$	NAND
0	1	1	1	$\overline{R1}.R2 + R1.\overline{R2} + R1.R2 = R1+R2$	OR
1	0	0	0	$\overline{R1}.\overline{R2} = \overline{R1+R2}$	NOR
0	1	1	0	$\overline{R1}.R2 + R1.\overline{R2} =$	XOR
1	0	0	1	$=$	XNOR

$$\text{Logical\_Out} = \overline{L0.L1.L2.L3} = \overline{L0.L1} + \overline{L2.L3} = \overline{L0} + \overline{L1} + \overline{L2} + \overline{L3}$$

$$L0 = \overline{S0 . \overline{R1} . \overline{R2}} \quad L1 = \overline{S1 . \overline{R1} . R2} \quad L2 = \overline{S2 . R1 . \overline{R2}} \quad L3 = \overline{S3 . R1 . R2}$$

$$\text{Logical\_Out} = S0.\overline{R1}.\overline{R2} + S1.\overline{R1}.R2 + S2.R1.\overline{R2} + S3.R1.R2$$

Figure 4.8: Logicals T2.

Figure 2.5: T2 Logicals structure

2.3.2 Booth multiplier

The Booth multiplier is based on the Booth Algorithm.

```
$i = 0$
$P = 0$
while $i < M - 2$ loop
    $P \leq P + vp(b[i+1], b[i], b[i-1])$
    $A \leq A \cdot 4$
    $i \leq i + 2$
end loop
```

Booth recodes the multiplier term, with three bits in one block as b[i+1], b[i], b[i-1], such that each block overlaps the previous block by one bit. Grouping starts from the LSB, and the first block only uses two bits of the multiplier, assuming a zero for the third bit.

Encoding			
b[i+1]	**b[i]**	**b[i-1]**	**vp**
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

Table 2.1: Booth encoding

The general structure of a generic Booth Multiplier is shown in figure 2.6.
In this implementation, A is 24-bit and B is 8-bit wide for simplicity. In the DLX A and B are represented both on 16-bit, so there are seven 32-bit RCAs and eight stages of muxes and encoding. In the DLX implementations there are RCAs instead of other adder types to save a lot of area considering the high number of instances accepting an higher delay due to this implementation.

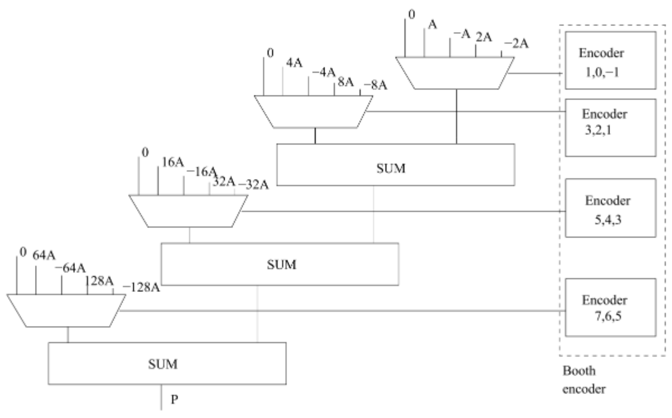


Figure 2.6: Booth multiplier structure

It is important to notice that in this implementation the 5-to-1 muxes and the encoder are fused in order to save other area. Each stage is equal and shown in figure 2.7.
The base idea is to use a combination of the selection signal used by the initial encoder to drive the 2-to-1 muxes used in the 5-to-1 version.
The last mux (MUX5) is used to have the possibility to use sel2 as Cin for the RCA to implemententent with the same Adder the sum or the subtraction

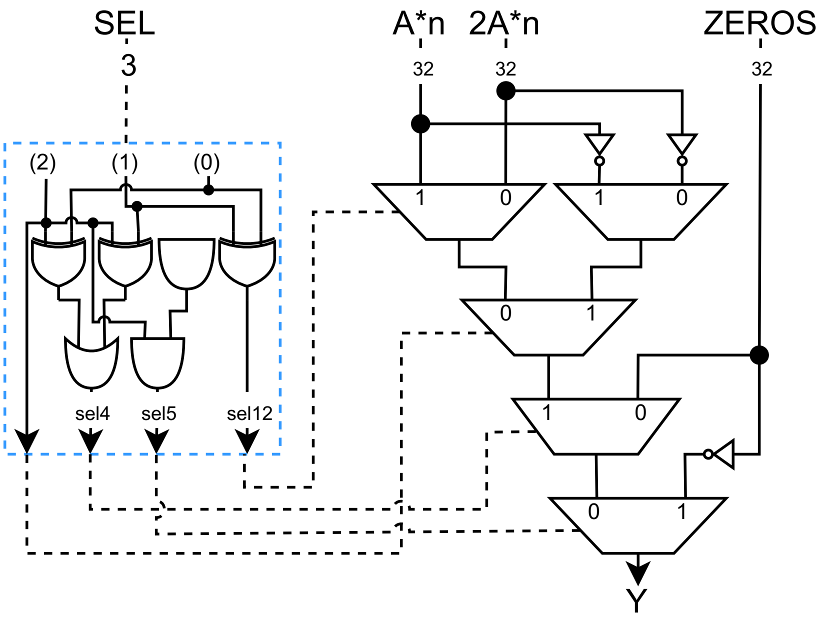


Figure 2.7: Special encoder

Select signals generation:

$$sel12 \Rightarrow SEL(1) \oplus SEL(0)$$
$$sel4 \Rightarrow (SEL(2) \oplus SEL(1)) + (SEL(2) \oplus SEL(0))$$
$$sel5 \Rightarrow SEL(2) * SEL(1) * SEL(0)$$

2.4 Memory Unit and Write-Back Unit

Memory Unit

The MEMORY UNIT is fourth stage of the pipeline and its aim is to read and write the DRAM. It is a configurable unit that can be adapted for various bit widths, using generic parameter NBIT, with a default configuration of 32 bits that represent the data size. The input and output ports of the unit are:

Input Ports:

- *PC_PLUS_4*: the program counter plus 4.
- *EN_LMD*: the memory stage enable signal, used as register enables.
- *CLK*: the clock signal.
- *RES*: the reset signal.
- *RD*: the Register Destination signal.
- *DRAM_DATA*: the DRAM data.
- *ALU_DATA*: the DRAM address.

Output Ports:

- *RD_OUT*: the RD (Register Destination) output.
- *LMD*: the DRAM data output.
- *PC_PLUS_4_OUT*: the program counter plus 4.
- *PC_OUT*: the program counter output.
- *ALU_DATA_OUT*: the execution result output.

Structural Architecture

In this stage, in addition to the reading and writing DRAM operations, there are 4 registers that are used to proceed the PC + 4 (necessary for writing into register R31 after a jump with register link), carry forward the value computed in the EXE stage, the value read from the memory and the RD out, to write during the write-back stage, in the right destination register.

Write-Back Unit

The WRITE-BACK UNIT is the last stage of the pipeline and it is a configurable unit that can be adapted for various bit widths, using generic parameter NBIT, with a default configuration of 32 bits that represent the data size. The input and output ports of the unit are:

Input Ports:

- *LMD*: the DRAM data output.

- *ALU_DATA*: the execution result output.
- *SEL*: the 2-to-1 multiplexer selector.

Output Ports:

- *DATA_OUT*: the mux output which has to be written back into the register file.

Structural Architecture

This last part of the DLX datapath is made up of a single multiplexer which has the task of choosing whether to write the output data from the execution stage or from the memory stage into the register file.

2.5 Control Unit

The implemented Control Unit (CU) operates in a hard-wired and micro-programmed fashion and this approach was chosen to ensure the fastest and most straightforward implementation, making it easy to understand.
The CU communicates with the datapath using a control word (CW) of 27 bits, which is taken from a LUT (present in MyTypes package), that is addressed directly by the FUNC field of a R-Type instruction or by the OPCODE field in the case of an I-type instruction.
Additionally, the CU provides a 6-bit Operation Code (ALU_MODE,) that determines how the Arithmetic Logic Unit (ALU) behaves during the Execution (EXE) stage, and the 1-bit execution unit type selector (SEL_EXE_UNIT_TYPE), which chooses whether to activate the alu or the booth multiplier component. The input and output ports of the unit are:

Input Ports:

- *IR_IN*: IRAM data output.
- *CLK*: clock signal.
- *RES*: reset signal.
- *STALL*: stall signal.
- *FLUSH*: flush signal.

Output Ports:

- *EN_FETCH* : Enable Fetch Stage
- *SEL_SIGN_DECODE* : Select Sign Decode
- *EN_DECODE* : Enable Decode Stage
- *EN_RF* : Enable Register File
- *RD1_EN_RF* : Enable Register File Read Port 1
- *RD2_EN_RF* : Enable Register File Read Port 2
- *SEL_A_EXE* : Selection of Operand A in Execution Stage
- *SEL_B_EXE* : Selection of Operand B in Execution Stage
- *EN_ALU_EXE* : Enable ALU in Execution Stage
- *SEL_MUX_JUMP_ALU_REGA* : Select ALU or Register A in Mux Jump
- *SEL_LHI* : Select LHI Operation
- *SEL_EXE_UNIT_TYPE* : Select Execution Unit Type
- *ALU_MODE* : ALU Mode
- *EN_LMD_MEM* : Enable Load Memory Data in Memory Stage
- *SEL* : Selector for Different Data Width Reads/Writes

- *RM* : Read Enable Memory
- *WM* : Write Enable Memory
- *EN* : Enable of the Memory
- *SEL_WB* : Select Write Back Stage
- *WR_EN_RF* : Enable Register File Write Back
- *SEL_ADD_WR* : Select Address Write
- *EN_PC* : Enable Program Counter
- *OP_TYPE_HAZARD* : Operand Type Hazard
- *OP_TYPE_FORWARD* : Operand Type Forward

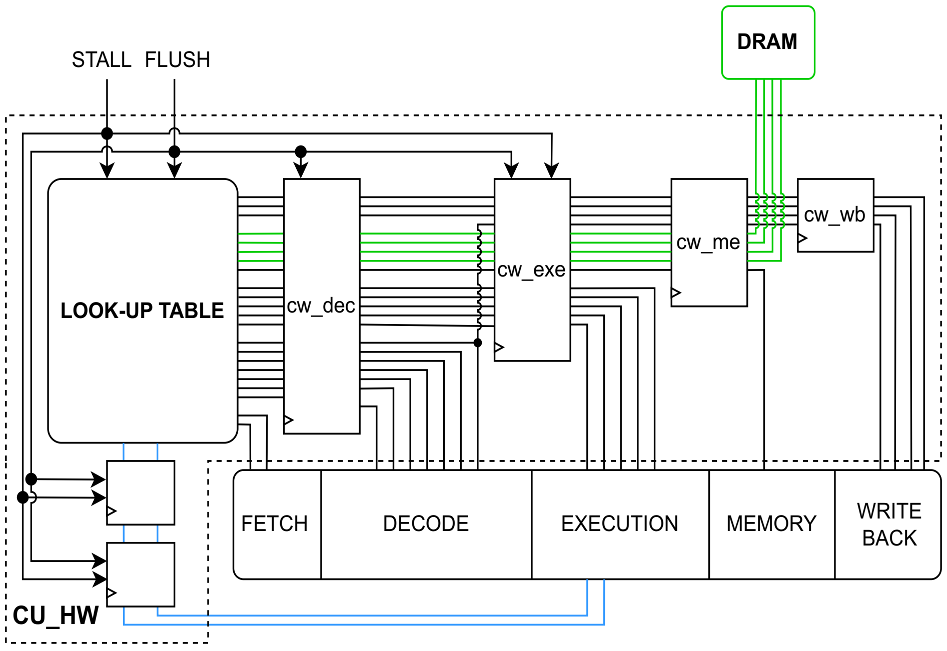


Figure 2.8: Control Unit structure.

Control signals assignment for each stage

The 27 bits of the control word:

26	EN_FETCH
25, 24	SEL_SIGN_DEC
23	EN_DEC
22	RD1
21	RD2
20	SEL_A
19	SEL_B
18	EN_ALU
17	SEL_MUX_JUMP
16	EN_LMD
15, 14, 13	SEL_MEM
12	RM
11	WM
10	EN
9	SEL_WB
8	WR_EN_RF
7	SEL_ADD_RF
6	EN_PC
5, 4	OP_TYPE_H
3, 2	OP_TYPE_F
1	EN_RF
0	LHI

Table 2.2: Control Word.

have been divided for each stage as follows:

26	EN_FETCH
6	EN_PC

Table 2.3: FETCH stage signals of CW.

25, 24	SEL_SIGN_DEC
23	EN_DEC
22	RD1
21	RD2
5, 4	OP_TYPE_H
3, 2	OP_TYPE_F
1	EN_RF

Table 2.4: DECODE stage signals of CW.

20	SEL_A
19	SEL_B
18	EN_ALU
17	SEL_MUX_JUMP
0	LHI

Table 2.5: EXECUTION stage signals of CW.

16	EN_LMD
15, 14, 13	SEL_MEM
12	RM
11	WM
10	EN

Table 2.6: MEMORY stage and DRAM signals of CW.

9	SEL_WB
8	WR_EN_RF
7	SEL_ADD_RF
1	EN_RF

Table 2.7: WRITE-BACK stage signals of CW.

Types of processes

The control unit is divided into three large processes:

1. *CW_ASSEGNATION*: This process is responsible for assigning control signals based on the opcode of the input instructions and the signals of stall or flush. These control signals determine which stages of the processor should be active or deactivated during the execution of instructions. For example, the process can enable or disable the fetch or decode stage based on certain conditions.

2. *CW_PIPE*: It manages the flow of information between different stages, ensuring that instructions are processed correctly and in the correct order.

3. *ALU_OP_CODE_P*: It is responsible for determining the correct operation code for the ALU unit based on the opcode and function of the input instructions. In practice, this process defines how the ALU unit should perform the mathematical and logical operations required by the instructions.

2.6 Forwarding and Hazard Unit

The following two units are one of the pro features in our DLX. They consist in Units that operate through different pipeline stages and produce signals to control the internal structure of the DLX or which are sent to Control Unit to stall the pipeline. The hazard that can occour in our architecture are:

- *RAW HAZARD*: Occurs when an instruction tries to read a register before a previous instruction that writes to that register has finished writing. This can be resolved through forwarding or pipeline stalls.
- *WAR HAZARD*: Occurs when an instruction writes to a register that a later instruction is trying to read. This can be resolved through forwarding (adopted solution) or by reordering instructions.
- *CONTROL HAZARD*: These hazards are caused by the conditional branching in code. In DLX, control hazards can lead to incorrect instruction fetching or pipeline stalls. DLX deals with control hazards through branch prediction techniques like branch target prediction (in the DLX a branch always not-taken policy is adopted) and instruction reordering.

In the described architecture all the possible forwarding types are solved (they will described above) while the hazards that remain to control are solved with pipeline breaks (stalls).
Forwarding operations involve detecting data hazards and forwarding data from the output of one stage directly to the input of another stage to resolve hazards. This allows instructions that depend on previous results to proceed without stalls, improving overall pipeline efficiency. For example, if an instruction in the "Execute" stage produces a result that is needed by an instruction in the "Decode" stage, forwarding operations can route the data directly from the "Execute" stage output to the "Execute" stage input to avoid a pipeline stall.

2.6.1 Forwarding Unit

The "FORWARDING_UNIT" is a configurable unit that can be adapted for various bit widths, it is managed by one generic parameter: NBIT with a default configuration of 5 bits that represent the RF address size.

Input Ports:

- *CLK*: clock signal.
- *RES*: reset signal.
- *FLUSH*: FLUSH signal.
- *OP_TYPE*: Operation type: R-type, immediate, store, jump or branch.
- *RS1*: RS1 address from decode unit.
- *RS2*: RS2 address from decode unit.
- *RD1*: RD address of the instruction actually in execution stage.
- *RD2*: RD address of the instruction actually in memory stage.

Output Ports:

- *SEL_FW_A*:control signal for the A forwarding muxes in execution stage.
- *SEL_FW_B*:control signal for the B forwarding muxes in execution stage.
- *SEL_FW_BRANCH*:control signal for the branch forwarding muxes in decode stage.

Architecture: All data apart from RD1 and RD2 are passed to this unit during the decode stage, directly from the IR_DECODER. All the forwarding verifications are performed only if the flush is not occoured the two cycles before to avoid the possibility to use destination registers that have been flushed and the taken data are false data. Then at each clock cycle, the generated selectors for A and B are sent in the execution stages while the one for the branches is sent immediately because it is used in decode. In case of reset signal all the signals are cleared and reported at their default zero value. In fact the zero address means the use of the R0 register that is not allowed in this architecture as it is a special register. The architecture is divided in three main portions:

- *SEL_FW_A_GEN*: it is used to computre the forwarding control signal for the next execution for the A register. It is conputed in all the operations that uses as input this register. For example the load instructions, since they have only a register destination, are not allowed to use this forwarding type. it is computed comparing the actual RS1 with RD1 and by verifying that this value is not the R0 address for the previous ALU data and the same check but on RD2 for the memory data.
- *SEL_FW_B_GEN*: it is used to computre the forwarding control signal for the next execution for the B register. It is conputed in all the operations that uses as input this register: They can be only R-type or store instructions. For example an immediate instruction, since it uses only one register as source cannot use the forwarding on the second register B but only on A. it is computed comparing the actual RS2 with RD1 and by verifying that this value is not the R0 address for the previous ALU data and the same check but on RD2 for the memory data.
- *SEL_FW_BRANCH_GEN*: it is used to computre the forwarding control signal for the actual decode unit. It is generated only for BNEZ and BEQZ operations. it is computed comparing the actual RS1 with RD2 and by verifying that this value is not the R0 address for the memory data.

In all the cases where forwarding control signals are not generated, a default zero value is assigned at the signal.

2.6.2 Hazard Unit

The "HAZARD_UNIT" is a configurable unit that can be adapted for various bit widths, it is managed by one generic parameter: NBIT with a default configuration of 5 bits that represent the RF address size.

Input Ports:

- *CLK*: clock signal.
- *RES*: reset signal.
- *OP_TYPE*: MSB for branch, LSB for load.

- *RS1*: RS1 address from decode unit.
- *RS2*: RS2 address from decode unit.
- *RD*: RD address from decode unit.

Output Port:

- *STALL*: signal to be sent at the Control Unit.

Architecture:

All data are passed to this unit during the decode stage, directly from the IR_DECODER. Then, at each clock cycle, the RD register is passed into a signal to store the value that will be used in the next cycle and the optype for load is stored in another signal to maintain the value to be used in the next stage. When the reset occours those two signals are cleared and reported to their default zero value to loose informations. The architecture is divided in two main portions:

- *Load stall*: it compares the current RS1 and RS2 with the RD of the previous operation checking that it has to be different from 0, if one of the two comparations are true and the rd_prev is not zero the stall signal is generated. The check is performed only if the previous operation that came in the Unit was a load type operation
- *Branch stall*: it compares the current RS1 with the RD of the previous operation, if it is zero the stall signal is generated. The check is performed only if the current operation that come in the Unit is a branch type operation

In all the cases where the stall signal is not generated, a default zero value is assigned.

2.7 Memories: IRAM and DRAM

In the DLX are present an IRAM and a DRAM. Both memories are described in VHDL only for simulation purpose because they are not synthesizable, in fact we expect them to be connected from the outside of the DLX. Both memory are read asynchronously, but IRAM is written asynchronously, while DRAM is written synchronously.

2.7.1 IRAM

The generic declaration are RAM_DEPTH, which is an integer representing the size of the IRAM, and the I_SIZE, which is an integer representing the size of each instruction stored in the IRAM in bits. The inputs and outputs port of the IRAM are:

Input Ports:

- *Rst*: the reset signal.
- *Addr*: the address used to read data from the IRAM.

Output Ports:

- *Dout*: the data output from the IRAM.

The IRAM purpose in the DLX processor is to serve as the storage for program instructions and enable their retrieval and execution. The IRAM cannot be written during the execution of the program, in fact, the stored instruction are read previously from an external file, that must be written in hexadecimal using ASCII coding. To do this, we implemented a process (FILL_IRAM) that is responsible for filling the instruction RAM with firmware from an external file. It opens the external file in reading mode, for each line of the file reads an instruction and stores it in the memory bytes until the end of the file is reached.
The IRAM when an Addr is provided in input must send the correct data to the output Dout. The architecture begins by converting the address input Addr to an integer Addr_to_int. This conversion is important for accessing the appropriate location within the IRAM. The architecture then selects 4 bytes from the memory array MEM based on the address input. It does so for each of the four bytes that make up an instruction. If the address is outside the valid range, the output bytes are assigned 'Z' to indicate an invalid or uninitialized memory location. The selected bytes are concatenated to form the Dout signal, representing the complete instruction fetched from memory.

2.7.2 DRAM

The role of the DRAM is storing and retrieving data within the DLX processor's architecture.
The entity declaration of the DRAM includes two generic, the RAM_SIZE, that is an integer specifying the size of the data memory, and the WORD_SIZE, that is an integer representing the size in bits of each data word stored in the memory.
The inputs and outputs port of the DRAM are:

Input Ports:

- *RST*: the reset signal.
- *ADDR*: the address input used for reading and writing data.
- *DATA_IN*: data input for read/write operations.

- *SEL*: the selector input for specifying different data width reads/writes.
- *RM*: read enable signal.
- *WM*: write enable signal.
- *EN*: enable signal for the entire memory.
- *CLK*: clock input for synchronous operations.

Output Ports:

- *DATA_OUT*: data output for read operations.

The architecture manages read and write operations to the memory based on various control signals and data widths specified by the selector. It also handles reset, clock synchronization, and high impedance states when the memory is not in use.
A synchronous write process is defined, sensitive to changes in the reset signal RST and the clock signal CLK. If the reset is asserted (RST = '1'), the memory is reset to all zeros. During the rising edge of the clock (rising_edge(CLK)), and if the memory is enabled (EN='1'), the process checks if write is enabled (WM = '1'). Depending on the selector input SEL, the process writes data into the selected address of the data memory. It handles full word writes, byte writes, and half-word writes, filling the appropriate bits in the memory element. An asynchronous read process is defined, sensitive to changes in the memory enable signal EN, read enable signal RM, address ADDR, and selector SEL. If the memory is enabled (EN='1'), and read is enabled (RM= '1'), the process reads data from the selected address in the data memory. Depending on the selector input SEL, the process extracts the desired data width (full word, byte, or half-word) and provides it as the DATA_OUT output signal. High impedance ('Z') is assigned to DATA_OUT when the memory is not used (EN='0') to indicate that the output is invalid.

CHAPTER 3

Simulation and Benchmarking

In order to see the functionality of our DLX we needed to simulate the design using ModelSim. To simplify the simulation we have realized some script.

3.1 Scripts for simulation process

3.1.1 simulation.sh

This simple bash script is designed to be launched from the terminal and it set up the environment for the simulation. In particular it execute the command ***setmentor***, necessary to set up all the ModelSim tools, deletes file and directory from previous simulation, if presents, and then run the ***Simulation.tcl*** script through the ***source*** command.

3.1.2 simulation.tcl

This tcl script is responsible for the compilation of all the files used in the design. In particular it uses the command ***vcom*** for compiling all the VHDL files, furthermore it performs syntax checking, manages libraries, reports errors and warnings, and can apply optimizations during compilation of the files. At the end of compilation another script is launched through the use of the command ***vsim***. Vsim command launches simulations of compiled designs and in particular provides waveform visualization and executes testbenches. In our case we added the *-do* flag to allow it to launch the ***simulation.do*** script.

3.1.3 simulation.do

This script is designed to be used in ModelSim and contains a series of commands that automate various tasks, making the simulation stage simpler and faster.
At first it execute again the command ***vsim*** to set up the simulation environment in ModelSim. The flag *-gui* is used to launch ModelSim in graphical mode, *-t ns* sets the simulation time unit to nanoseconds, *work.DLX_tb* specifies the top-level testbench entity to be simulated from the work library and the flag *-voptargs=+acc* is used to obtain better signal resolution.
Now the script execute the command ***add wave*** to show the waveform viewer, and then the commands ***run 100 ns***,to run the simulation for 100 nanoseconds, and ***wave zoom full*** to provide a full view of the simulation.
At this point two commands ***mem save*** are executed and the simulation is completed. The first command is used to create the file *registerfile.mem* and to save in this file all the data contained in the

register file of the DLX at the end of the simulation. In the same way, the second command creates the file *dramdata.mem* and saves in it all the data contained in the DRAM of the DLX at the end of the simulation.

3.1.4 assembler.sh

Another optional script can be used to simplify further the simulation phase. This bash script can be used to streamline the process of compiling and converting the assembly code, that the DLX must receive in input. In particular it launch the command ***perl*** to compile the DLX assembly code provided. The assembler then generates two output files: the *.asm.exe*, that is the compiled executable, and the *.list*, that is the assembly code listing. The name of this two file is provided as parameter in the command line when the script is launched.
Then, the script executes a shell script named conv2memory.sh, passing the compiled executable as input. The output of this shell script is saved in the test.asm.mem file.
At the end of the script there is the possibility to launch directly the *Simulation.sh* script, through a *soruce* command, to perform all the simulation.

3.2 Example of a simulation

This is an example of assembled program that can be run and tested in the DLX.
This one in particular is good to see the functionality of the flush and the stall events combined in different ways and also the real behavior of the forwarding unit describet in its own section.
The code starts with an initial counter value assigned to r1 register and the clearing of register r3. Then there is iteration of two times of the code between lines 3 and 6 where r1 decrements its value and r3 increases its one. It is immediatively clear the presence of a RAW type hazard solved with the use of forwarding between line 2 and its consecutive instruction thar uses r3. Another point to underline is another data dependency at line 6 where the branch requires the updated result of the previour line instruction. In this case the solution adopted is different because there is the need to have the result in decode while it is not yet produced in the execution stage of the previous operation: to solve this the stall signal raises (Figure 3.1: 11 ns) and the entire pipeline before the execution stage is broken, so no other new instructions are fetched and the stages starting from the execution to write back flow normally. At that time the forwarding become possible and it is used to provide at the branch the new data of r1 and the decision to take the branch is taken. During the next cycle, the Flush signal is raised to delete the new instruction decoded assuming the branch not taken. Then, during the second iteration the operations required are the same but this time the branch is not taken and the pipeline continues with her normal execution without interrupts thanks to the not-taken policy. Then, at line 9 the JAL operation is taken causing another flush operation and taking the jump. It immediately takes another jump using the register r31 that contains the return address and the code writes into r8 and r9 registers. At that point the jump to the last section is taken and a new flush is performed on the pipeline. The two operations on r10 and its consecutive one are left in the code to demostrate that the flush signal correctly delete the instruction actually in the pipeline. All the discussed operations can be verified in the waveform Figure 3.1. The final content of the register file is also reported above to have a better clarification.

```
line  address  contents                                 iter. 1     iter.2      iter.3
   1  00000000  20210002 addi r1, r1, 2       ; $r1 = 2
   2  00000004  00631826 xor r3, r3, r3       ; $r3 = 0
   3  00000008           cycle:               ;
   4  00000008  20630001 addi r3, r3, 1       ; $r3 = 1     $r3 = 2
   5  0000000c  28210001 subi r1, r1, 1       ; $r1 = 1     $r1 = 0
   6  00000010  1420fff4 bnez r1, cycle       ; branch T    branch NT
   7  00000014  01084026 xor r8, r8, r8       ;
   8  00000018  0061202d sge r4, r3, r1       ;             $r4 = 1
   9  0000001c  0c000014 jal final            ;             jump
  10  00000020  2108000f addi r8, r8, 15      ;                         $r8 = 15
  11  00000024  212900ff addi r9, r9, 255     ;                         $r9 = 255
  12  00000028  0800000c j end                ;                         jump
  13  0000002c  214a0001 addi r10, r10, 1     ; should remain at zero value
  14  00000030  214b0002 addi r11, r10, 2     ; should remain at zero value
  15  00000034           final:               ;
  16  00000034  4be00000 jr r31               ;             return
  17  00000038           end:                 ;
  18  00000038  216b0001 addi r11, r11, 1     ;                         $r11 = 1
  19  0000003c  216c0002 addi r12, r11, 2     ;                         $r12 = 3
  20  00000040  cc0d00ff movi r13, 255        ;                         $r13 = 255
```

The content of the file ***registerfile.mem*** at the end of the simulation of the program in the DLX is reported here using directly the memory file generated by the script.

```
// memory data file(do not edit the following line - required for mem load use)
// instance=/dlx_tb/test/datapath_inst/DECODE_UNIT_inst/REG_FILE/REGISTERS
// format=bin addressradix=h dataradix=b version=1.0 wordsperline=2
//
 @0 00000000000000000000000000000000 00000000000000000000000000000000
 @2 00000000000000000000000000000000 00000000000000000000000000000010
 @4 00000000000000000000000000000001 00000000000000000000000000000000
 @6 00000000000000000000000000000000 00000000000000000000000000000000
 @8 00000000000000000000000000001111 00000000000000000000000011111111
 @a 00000000000000000000000000000000 00000000000000000000000000000001
 @c 00000000000000000000000000000011 00000000000000000000000011111111
 @e 00000000000000000000000000000000 00000000000000000000000000000000
@10 00000000000000000000000000000000 00000000000000000000000000000000
@12 00000000000000000000000000000000 00000000000000000000000000000000
@14 00000000000000000000000000000000 00000000000000000000000000000000
@16 00000000000000000000000000000000 00000000000000000000000000000000
@18 00000000000000000000000000000000 00000000000000000000000000000000
@1a 00000000000000000000000000000000 00000000000000000000000000000000
@1c 00000000000000000000000000000000 00000000000000000000000000000000
@1e 00000000000000000000000000000000 00000000000000000000000000100000
```

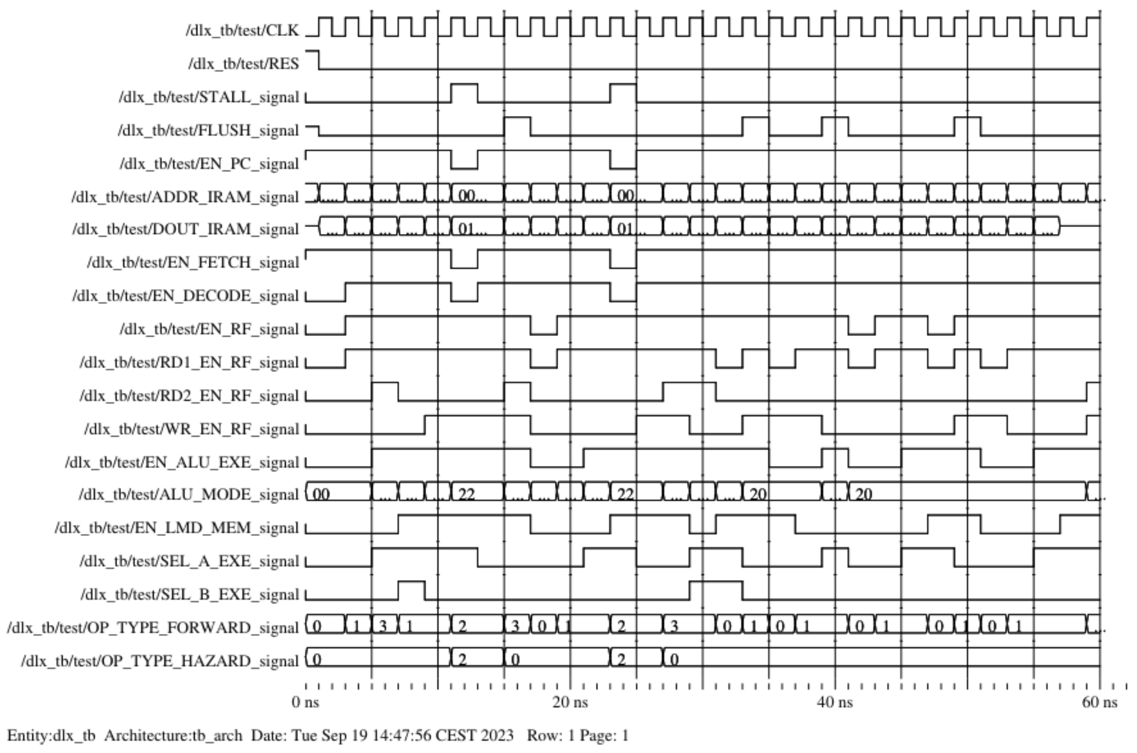


Figure 3.1: Waveform at the end of the simulation.

3.3 Testbenches

In "sim" folder there are also other testbenches in addition to the one used to simulate the entire design. to test the single components (especially the units that make up the ALU or the multiplier) it is necessary to modify the "Simulation.tcl" script removing the unused files, adding the corresponding testbench and in many cases add the compilation line for the linear feedback shift register (LFSR) component which is used to generate random numbers to test the components. The path to the LFSR component is the same as the other .vhd files in the "Components" folder present the DLX_vhd one.

CHAPTER 4

Synthesis

The synthesis phase of the design has been achieved through the use of two script. The first one is a bash script that streamlines the synthesis process by automating some preliminary tasks. In particular it copies, if it is not presents, the file ***.synopsys_dc.setup*** in the directory where we are running the script, check if the copy was successful, execute the command ***setsynopsys*** to set up the Synopsys tool and then run the script *Synthesis.tcl* in Design Vision with or without the graphical interface.
The second one is the core of the synthesis. It is a tcl script that automates the entire synthesis process of a DLX in the Synopsys tools.
The first part of the script is dedicated to the definition of a procedure through the command *proc*. The procedure ***report_slack*** implemented, that will be useful later in the synthesis phase, allows us to print on the terminal the 100 worst path sorted by slack.
The next part of the script is dedicated to the analysis of all VHD files used to describe the DLX through the command *analyze*. This command load all the files analyzed in the library WORK. We provided as parameters to the command analyze the format of the file, which is VHDL, the path of the file and the name of the library. The IRAM and DRAM are not present in the list of file because the memories are not synthesizable, as we expect them to be provided from outside the DLX.
Then the script proceed with the elaboration of all the architecture of the DLX through the use of the command ***elaborate***, using the library WORK created in the previous step.
Then there is the definition of the clock ***CLK*** of period 5 ns through the command ***create_clock***. The clock is necessary to compile in the next steps.
After compiling for the first time with the command ***compile*** we saved all the results using the report commands, in particular we saved the report of timing (*report_timing*), area (*report_area*), power (*report_power*) and clock (*report_clock*).

The representation of the top-level component in the synthesis phase is expressed in Figure 4.1 where the physical connection between the Control Unit and the DataPath (the only two submodules of the DLX) is illustred. Now start the optimization of the design. At first in the script is defined the ***MAX_PATH***, obtained from the timing analysis, and then its value is calculated in ns summing all the contribution in the *foreach_in_collection* structure. After this, another clock is defined, but this time with periods equal to *REQUIRED_TIME*, that is90% of the value of MAX_PATH previously calculated. Then, we defined and applied some constraint parameters:

- *max_transition_time*: limit signal switching times, reducing dynamic power consumption.
- *min_delay*: ensures that signals have a minimum delay to prevent metastability issues.
- *input_delay*: regulates the arrival time of input signals relative to the clock.

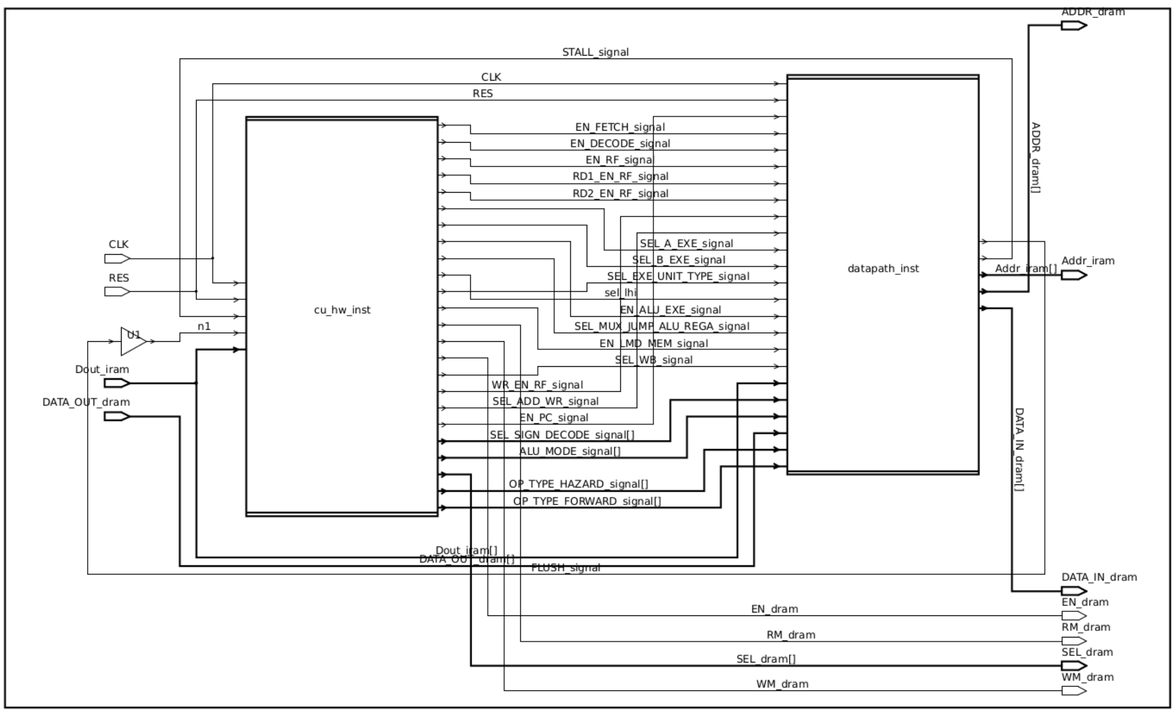


Figure 4.1: DLX block diagram

- *output_delay*: controls the output timing with respect to the clock signal.

The last step before compiling again is the optimization of the register through the command ***optimize_registers***, that aims to minimize the clock-to-Q delay of the registers to meet the desired clock period. Now the script compile again the design but this time with all the constraint. For the compilation this time we use ***compile_ultra*** that perform an high-effort optimization strategy. The flag -timing_high_effort_script is used to perform intensive timing optimization, -no_autoungroup disables automatic ungrouping of elements, and -gate_clock specifies that clock gating should be applied to reduce power consumption by removing the clock signal when the circuit is not in use or ignores clock signal. The -scan flag is used to enable the insertion of scan chains, which are sequences of scan flip-flops, into the synthesized digital design, allowing for efficient testing of integrated circuits by facilitating the loading of test patterns and capturing response data.
The clock gate option was tried also with specific in its configurations values but the results have no benefits compared to default configuration.
Now the script save all the reports and repeat all the steps for another three times with value of the clock equals to 80%,70% and 50% of the *MAX_PATH* value. Notice that the value of 50% is used only to force the slack to a negative value because canonical values normally range from 5% to 25%.

4.1 Results

We extracted all the relevant data form the report and obtained these table:

4.1.1 Timing

Worst Path			
	Required time (ns)	**Arrival time (ns)**	**Slack (ns)**
5 ns	4.96	4.95	0.01
90%	4.42	4.20	0.23
80%	3.93	3.72	0.21
70%	3.43	3.41	0.01
50%	2.44	2.93	-0.48

Table 4.1: Worst Path Timing

From the table 4.1, we can see that the arrival time of the worst critical path of the design non optimized is 4.95 ns with 5 ns of clock. The best result post optimization is obtained with the reduction of 30% of the initial worst path, in fact we obtained a worst path of 3.41 ns. With 10% and 20% reduction we obtained also good results, respectively 4.20 ns and 3.72 ns, while with 50% reduction the timing constraint is not met.

4.1.2 Area

Area			
	Number of nets	**Number of cells**	**Total cell area (μm^2)**
5 ns	30348	14960	17687
90%	21127	9357	14345
80%	20648	8871	15488
70%	20348	8745	15164
50%	20601	9143	15516

Table 4.2: Number of nets, number of cell and total area

In the table 4.2, we can observe the trend of the area in relation to the timing constraint. The initial area of the design with clock of 5 ns and no constraint is equal to 17687 μm^2. The best result after the optimization is 14345 μm^2 and is obtained with 90% of the value of the clock, which is the less restrictive constraint. In fact we can clearly see that as the constraints become more restrictive the total area increases.

4.1.3 Power

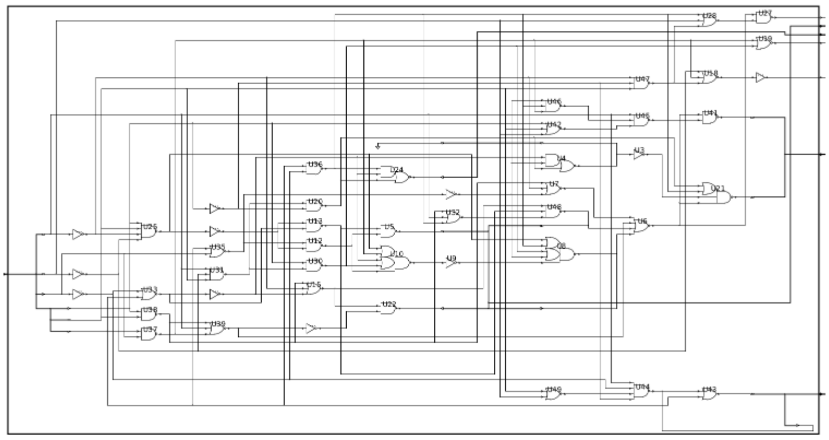
Power				
	Internal(μW)	**Switching(μW)**	**Leakage(nW)**	**Total(μW)**
5 ns	948.04	361.47	3.63e+5	1.67e+3
90%	1.27e+3	421.73	2.58e+5	1.95e+3
80%	1.79e+3	590.21	2.76e+5	2.65e+3
70%	1.96e+3	646.16	2.82e+5	2.89e+3
50%	2.84e+3	988.11	2.83e+5	4.11e+3

Table 4.3: Power

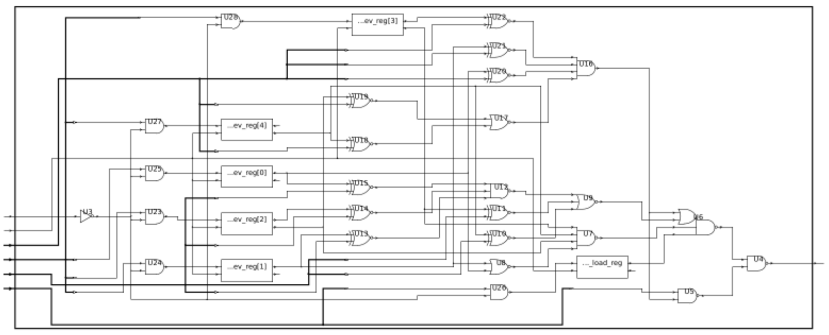
In the table 4.3 are reported all the different contribution to the total power consumption of the design. Looking at the table we see that the non optimized design has an internal power consumption of 948.04 μW and a total power consumption of 1.67e+3 μW. The internal power is the contribution that affects the most on total power consumption. We see that optimizing the design the internal power increases a lot, while the switching power remains almost stable and the leakage power decreases. Making more restrictive constraint all the contributions increase to allow all the condition to be met.

4.1.4 Considerations

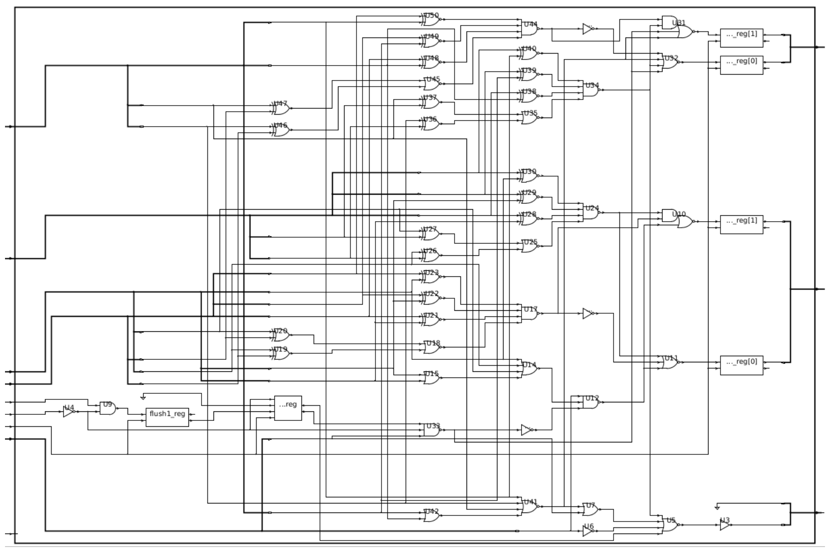
It can be intresting to see how a fully-behavioral VHDL code is translated during the synthesis. The following three figures show how these units are implemented in reality. A particular focus on Figure 4.2a where during the implementation phase it was thought that the real implementation would be carried out with a look-up table but instead everything was transformed into a combinational circuit made of logic gates. All the three photo are made after the first compile command, without adding other optimizations apart from the clock constraint of 5 ns that was verified multiple times before its usage.



(a) ALU Decoder after synthetis



(b) Hazard Unit after synthetis



(c) Forwarding Unit after synthetis

Figure 4.2: Implementations after synthesis

CHAPTER 5

Physical Design

The last stage of the project consists in placement and routing using Innovus by Cadence to obtain the the physical design.
The configuration is the initial step so, we started importing all the necessary file, like the SDC and Verilog file obtained from the synthesis. We proceeded structuring the floorplan defining the space for the cells and power supply ring, two metal rings used to connect Vdd and ground. Then, power rings, vertical stripes and horizontal stripes are inserted. The vertical and horizontal stripes are used to connect the power rings and correctly distribute the power and ground signals to the center of the chip. At that point the cell placement was executed and we prosecuted placing the input/output pin along the chip's edges.

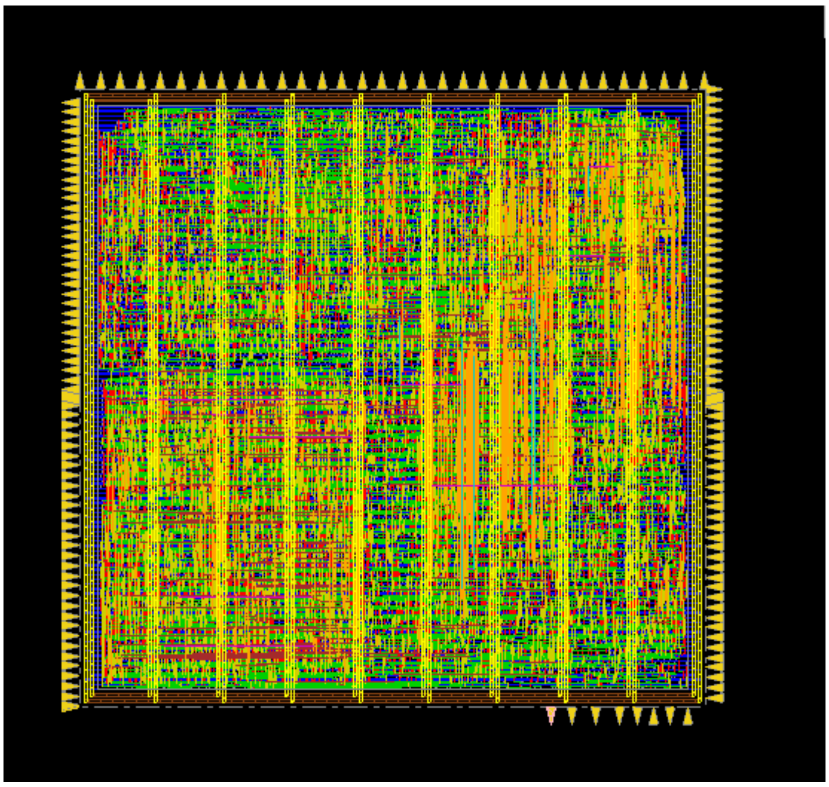


Figure 5.1: Overview of the physical layout obtained after the place and route operation.

As it is clear in the figure 5.1, the pins are not left to the nominal position along side, but they have been ordered in order to have a better clarification. On the left-edge there are the IRAM ports (Address and Dout, respectively output and input direction), on the right edge there are present two of the three data for the DRAM (Address and Dout), while in the top side the datain signal is present. All the control bits for the memories have been put on the bottom side starting from the right corner: CLK, RES, SEL_DRAM, EN_DRAM, WM_DRAM, RM_DRAM.
Before proceeding routing the signals we performed a Post Clock-tree-synthesis optimization and we placed filler cells to fill gaps in the layout. Then, we performed the routing. In this step the logical connections among cells are converted to physical interconnections. The final stage consists in the post routing optimization. In this phase the complete design undergoes to an additonal optimization to achieve the required time constraints.
Now the design is completed, finally we performed some analyses for timing and integrity.
The first analysis regards the behavior of the circuit over time, to do so we needed the parasitic resistance and capacitance of each metal wire that we found in the .spef file. Another check that we performed is about the delay of the design, to ensure that it respected all the timing constraints. The last analysis was needed to verify the connectivity and the design rules to ensure a reliable circuit. This was crucial for preventing issues like floating wires.

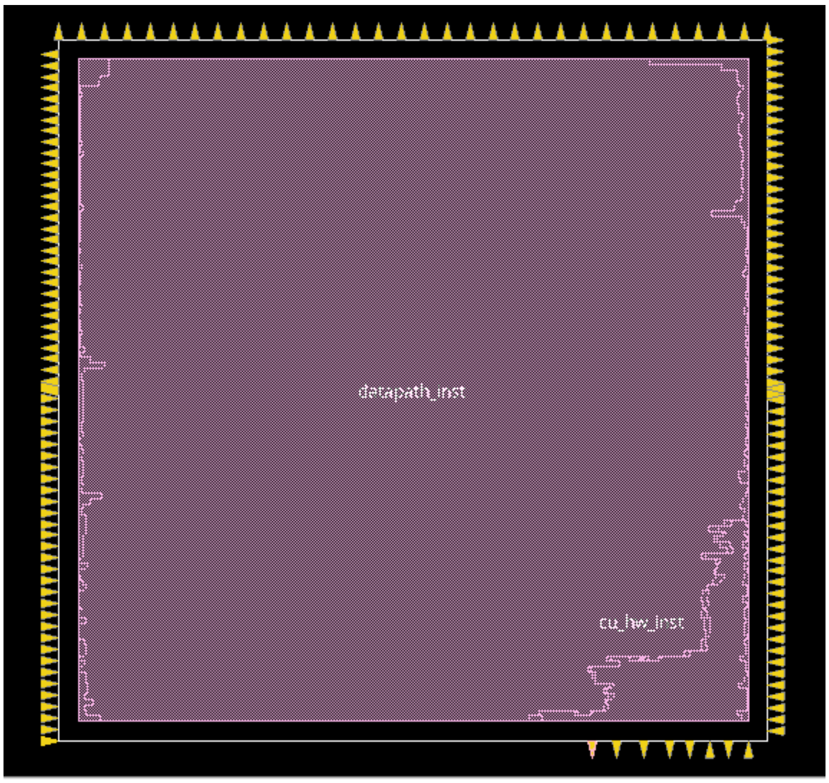


Figure 5.2: Allocation of space on the die.

5.1 Results

In the table 5.1 we can see some of the most relevant results regarding the area of the DLX. We see that most of the area is due to the datapath, and in particular the register file and the ALU. In the figure 5.2 we see the allocation of space on the die. From this image we can notice that most of the space is occupied by the datapath, according to the data in the previous table, while the control unit is settled mostly in the bottom right corner. In figure 5.1 the DLX after place and route operations is shown. From this image we can clearly see the 8 vertical stripes used to connect the power rings from top to bottom.

Module	**Gates**	**Cells**	Area um^2
DLX	19685	11888	15708
Datapath	18843	11557	15037
Register file	6163	2690	4918
Alu	2629	2048	2098
Control unit	835	325	666

Table 5.1: Area extracted by post-routing analysis