Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering
Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Group_42
Claudia Golino, Mattia Mirigaldi, Stefano Palmieri

September 22, 2021

Abstract

The DLX is a RISC (Reduced Instruction Set Computer) processor with a simple 32 bit load/store architecture, used in university for teaching purpose. The DLX Basic version is a five-stage fully pipelined architecture with a limited instructions set (consequently with a basic Control Unit and Datapath) that use a straightforward register file implementation.
The mandatory specifications for this project consist of the VHDL description of the DLX Basic version, followed by its basic synthesis and physical design (Top-bottom approach, from an RTL description down to physical design).

The DLX described in the following is a Pro version and differs from the basic one since :

- Larger instruction set (The complete set can be found at paragraph 2.2)
- More complex Datapath capable to implement the added instructions and optimized to being faster and less power consuming than basic version. Most remarkable additions are :
  - Pentium4 adder/subtractor.
  - Comparator Unit.
  - Logic Unit.
  - Windowed register file, that fix the high delay due to context switching that happens with the straightforward implementation of the register file.
- Advanced synthesis to reduce power consumption and post synthesis VHDL simulation to get realistic timing and power consumption.
- Advanced physical design analysis.

Contents

CHAPTER 1

Introduction

In this project it is presented a fully pipelined DLX processor, which is a simple load-store architecture with 32 32-bit general-purpose registers.
In particular the DLX is a RISC (Reduced Instruction Set Computer) processor based on the Harvard architecture (i.e. used two different memories, one for the instructions and one for data) and it can execute a set of available instructions, contained in the ISA.

1.1 Specification

The designed DLX, like the MIPS design, bases its performance on the use of an instruction pipeline. The process of the instruction execution is divided in 5 stages in order to increase the throughput of the designed processor. By doing this every DLX instruction can be implemented in at most five clock cycles.
Stages of the pipeline:

- IF (Instruction fetch)
  The program counter contains the address of the instruction to be fetched: the instruction is read from memory into the instruction register (IR) and the PC is incremented to address the next instruction to be processed.
- ID (Instruction decode)
  The instruction coming from the IR is decoded and the register file is accessed in order to read the registers. The outputs of the general-purpose registers are read into two temporary registers.
- EX (Execution)
  The temporary registers are used by the ALU to perform the requested operation. Here the jumps and conditional branches are evaluated.
- MEM (Memory access)
  Load and store are performed if an interaction with the memory is needed.
- WB (Write-back)
  The result (coming from alu or from the memory) is written into the register file.

Moreover the DLX instructions are 32-bits long, in which a 6-bit field, always present, defines the OPCODE. Depending on the instruction type, the remaining 26 bits assume different meanings. The instructions can be divided in 3 groups:

- R-type: Register to register ALU operations, with 2 source registers RS1, RS2 and one destination register RD. ALU operation is defined in the extra 11-bit field FUNC.

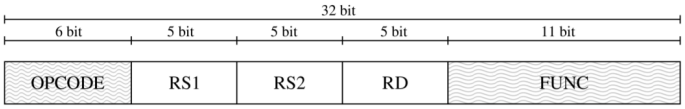


Figure 1.1: R-type instruction.

- I-type: Loads/Stores and conditional branch instructions. This type of instructions has one source register RS1, one destination register RD and an immediate that correspond to a certain value defined by the user.

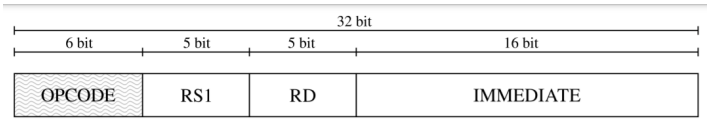


Figure 1.2: I-type instruction.

- J-types: Jump and jump link instructions, which contain the next instruction to be executed in the extra 26-bit field TARGET.

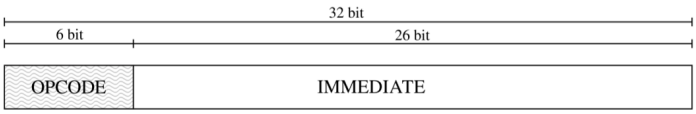


Figure 1.3: J-type instruction.

The list of possible instructions that can be performed by the designed DLX, divided by type is specified below.

The DLX memory is byte addressable in Big Endian mode with a 32-bit address and the memory operations are handled by loads or stores between memory and the general purpose registers (Register File)

I-type	
Mnemonic	Opcode
beqz	0x04
bnez	0x05
addi	0x08
addui	0x09
subi	0x0a
subui	0x0b
andi	0x0c
ori	0x0d
xori	0x0e
lhi	0x0f
slli	0x14
nop	0x15
srli	0x16
srai	0x17
seqi	0x18
snei	0x19
slti	0x1a
sgti	0x1b
slei	0x1c
sgei	0x1d
lb	0x20
lw	0x23
lbu	0x24
lhu	0x25
sb	0x28
sw	0x2b
sltui	0x3a
sgtui	0x3b
sgeui	0x3d

J-type	
Mnemonic	Opcode
j	0x02
jal	0x03
jr	0x12
jalr	0x13

R-type	
Mnemonic	FUNC-code
sll	0x04
srl	0x06
sra	0x07
add	0x20
addu	0x21
sub	0x22
subu	0x23
and	0x24
or	0x25
xor	0x26
seq	0x28
sne	0x29
slt	0x2a
sgt	0x2b
sle	0x2c
sge	0x2d
sltu	0x3a
sgtu	0x3b
sgeu	0x3d

*OP-CODE is always 0x00

Table 1.1: Instrucion set

The instruction memory is a ROM based one whose VHDL description can be found in the file "romem.vhd", is filled with the firmware by a process which reads from the file "hex.txt"; this memory, every time receive a new address in input, gives after two clock cycle a new instruction in output. We created some files in the directory of the ROM that can be used to test the instructions implemented, they can easily be used by copying all the instructions in the "hex.txt".
The data memory is a DRAM whose VHDL description can be found in the file "rwmem.vhd", has a single-port for reading and writing, the address is on 32 bits, data on 64 bits and a has delay of 2 cycles.
The top-level entity is the DLX processor that can be found in the file "DLX.vhd" and it instantiates the :

- Datapath : responsible of the data processing operations that basically is capable of implementing all of the instructions set by providing load and store between memory and registers and ALU operations (that occurs only between registers).
- Control Unit (CU) : directs the operation of the processor by generating the appropriate signals to drive the datapath. The one used is a HARDWIRED one.

A more complex description about these components will be found in the following chapters.

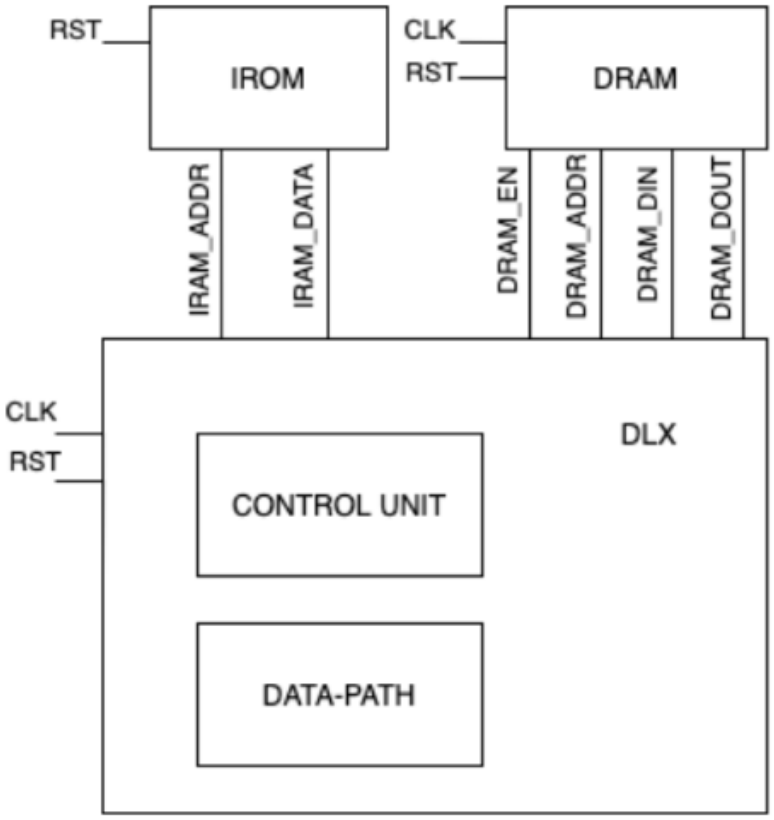


Figure 1.4: Block diagram representation of the Top-level DLX and interfaces with memories.

1.2 Functionality

In order to run an assembly program is not write each instruction code manually in the instruction memory, rather the program is converted into a format readable by the VHDL control unit and then it is run.
These steps are managed by the script "assembler.sh" and it can be tested with one of the code examples present in the directory "asm-example". The behavior then can be simulated with ModelSim. (An example of a ModelSim Simulation script can be found in the appendix).

CHAPTER 2

Control Unit

2.1 Introduction

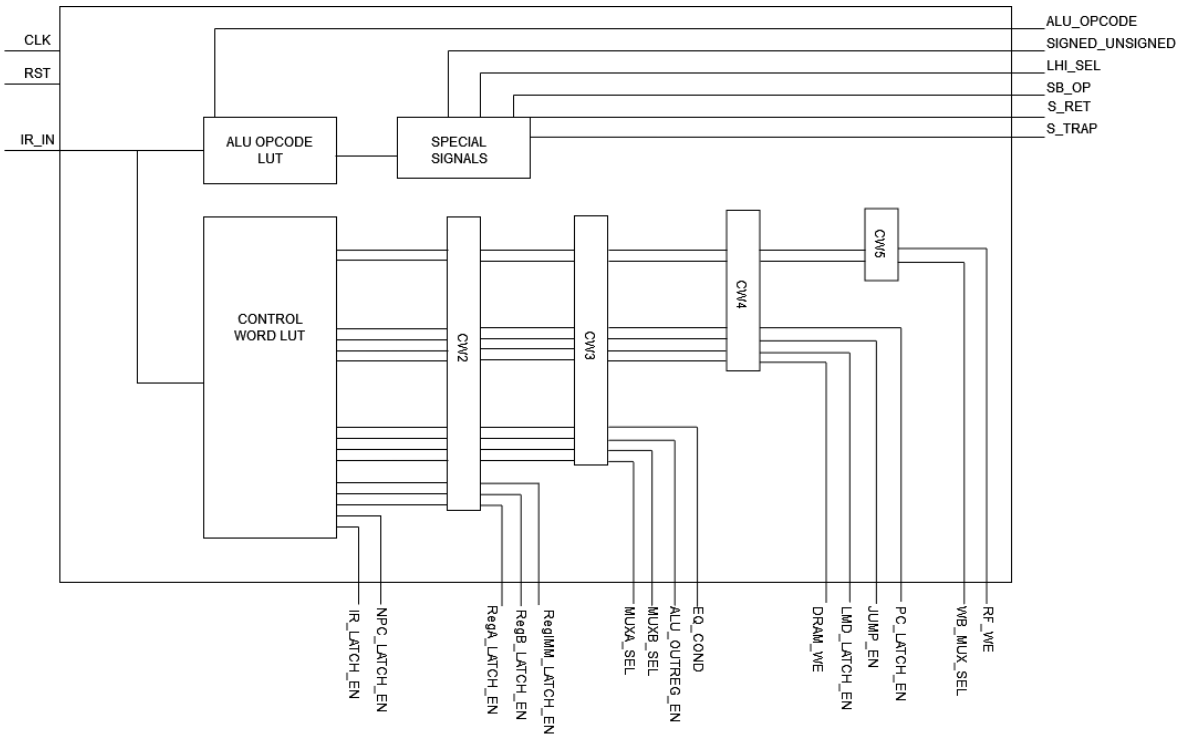
The control unit is a fundamental part of the chipset, in this part all the signals needed by the datapath are generated. The control unit is in charge to manage the pipeline and to allow the datapath to execute instructions in the correct way. Every three clock cycles the Program Counter (PC) is incremented, a new instruction is fetched from the ROM that reads the instruction pointed by the PC value; then the new instruction is stored into the Instruction Register (IR) before entering the CU. The hard-wired design style was chosen to describe the CU behavior due to its simplicity and liability. Two Look-up-Tables were used in the combinational logic that asserts a 32-bit Control Word (CW) signal and the ALU opcode. In each stage of the pipeline (FETCH, DECODE, EXECUTE, MEMORY ACCESS, and WRITE BACK) a portion of the CW is delivered to the data-path. The fetch unit is 3 stages long to allow the reading of the IR by the ROM, all the other stages are 1 clock cycle long.

2.2 Functionality

The Control Unit is composed of a LUT in which there are present all the signals needed for the execution, the opcode of the Instruction Register is the address to access to the correct entry. Moreover the opcode and the func allow the CU to tell the datapath which type of operation has to be executed, in fact they are used as reference to send to the datapath the correct aluOp (alu operation). All the aluOp are stored in the file 000-Global as a new type of signal:

type aluOp is (NOP, ADDS, LLS, LRS, ADD, SUB, ANDR, ORR, XORR, SNE, SLE, SGE, BEQZ, BNEZ, SUBI, ANDI, ORI, XORI, SLLI, SRLI, SNEI, SLEI, SGEI, LW, SW, SUBU, SUBUI, ADDU, ADDUI,SRA1,SEQ, SLT, SGT, SLTU, SGTU, SGEU, LHI, JR, JALR, SRAI, SEQI, SLTI, SGTI, LB, LBU, LHU, SB, SLTUI, SGTUI, SGEUI);

New instructions are fetched every 3 clock cycle, entering the pipeline in order. Therefore, the DLX microprocessor can execute up to 3 instructions simultaneously, each allocated to one of the pipeline stages. The control word is shifted by one position at each positive edge of the clock cycle to ensure that the portion of control signals related to a certain stage is delivered to the data-path in the correct order. In the following lines, a brief description of each one of the control signals is provided.



- **IR_LATCH_EN:** Instruction Register Latch Enable
- **NPC_LATCH_EN:** NextProgramCounter Register Latch Enable.
- **RegA_LATCH_EN:** RF read enable source register of operand A.
- **RegB_LATCH_EN:** RF read enable source register of operand B.
- **RegIMM_LATCH_EN:** Enable signal for gating the Immediate Register.
- **MuxA SEL:** Select signal for choosing between register A and NPC.
- **MuxB SEL:** Select signal for choosing between register B and immediate register.
- **ALU OPCODE:** ALU operation code.
- **DRAM_WE:** DRAM write enable signal.
- **LMD_LATCH_EN:** LMD Register Latch Enable.
- **JUMP EN:** Asserted when there is a direct branch.
- **PC_LATCH_EN:** Program Counte Latch Enable.
- **WB_MUX_SEL:** Select signal that determines the source of the data to be written into the RF.
- **RF_WE:** RF write enable signal during WRITE-BACK stage fed to forwarding unit also.
- **SIGNED_UNSIGNED:** Set to 1 if the operation is between unsigned number.
- **LHI_SEL:** It is set to one if the operation is a lhi.

- **SB_OP:** It is set to one if the operation is a sb.
- **S_TRAP:** It is set to one if the operation is a trap.
- **S_RET:** It is set to one if the operation is a ret.

As evidenced from the image there are some signals called special, they are generated by a process that activates this signal only when they are needed and for a specific period of time, for this reason they are not taken as the other from the LUT.

CHAPTER 3

Datapath

3.1 Introduction

The datapath is a set of functional units that is responsible for the data processing operations. The datapath handles all the instructions in the ISA by the load, store and ALU operations. As discussed in the Introduction in order to improve the performance of the designed processor, the process is divided in five stages and it's speed up by pipeling the stages. The throughput (number of instructions executed per unit time) is incremented because the simultaneous execution of more than one instruction allows to execute ca. one instruction per clock. When the stages are not pipelined every instruction takes 5 clocks to be performed, as in the case of the operation of the first instruction executed in the pipeline. Each stage performs a specific operation and it is connected to input of next stage by interface registers, used to hold the intermediate output between the two stages.

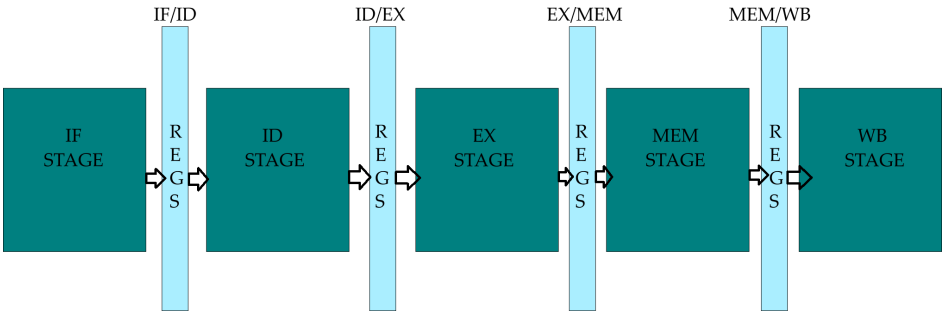


Figure 3.1: Basic representation of the stages of the pipeline and its registers.

3.2 Functionality

The explanation of the entire datapath will proceed stage by stage in order to have a better understanding of every step of the pipeline. A full schema of the datapath is shown below in Figure 3.2 .

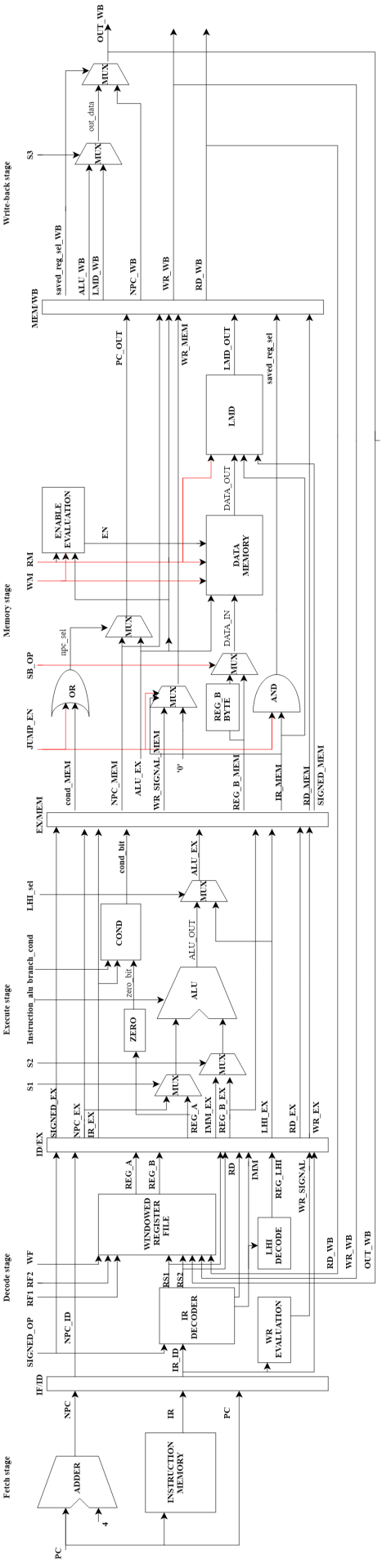


Figure 3.2: Full Schema.

3.3 Fetch stage

In the fetch stage the address contained in the program counter (PC) goes in input to the Instruction Memory to retrieve the instruction that has to be executed; the ouput of the Instruction Memory is the Instruction register (IR). Moreover the next address is computed by an adder with the value of the current PC and it's saved in the next program counter (NPC register).The NPC value is always computed, even if there is taken branch or a jump, which will be evaluated in the EX stage.
The principle components of this stage, as shown in Figure 3.3, are the:

- Program counter memory;
- Instruction Memory.

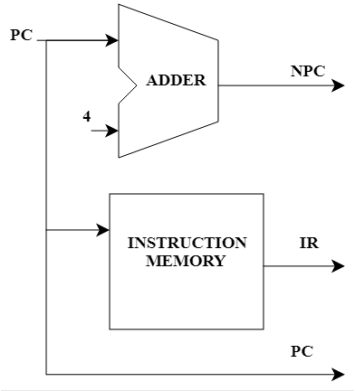


Figure 3.3: Fetch stage components.

3.3.1 Program counter adder

The adder always calculate a simple sum: the current PC +4. This sum will calculated the address of the next instruction because the instructions are 32-bits (4 bytes) long and the Instruction memory has 1-byte entries. It doesn't need to be as complex as the adder present in the ALU, so it is simply described in a behavioural way.

IF/ID REGISTERS
These are the Fetch/Decode pipeline registers:

- PC_Dec: stores the current program counter
- NPC_Dec: stores the next program counter
- IR_Dec: stores the instruction register

N.B. In order to synthesize the DLX the IRAM memory was implemeted with a ROM, which has a fixed delay: a previous pipeline stage with the same signals was added to achieve the synchronization of the entire pipelined process.

3.4 Decode stage

In the decode stage the IR is decoded to retrieve the needed registers in input to the register file (RF). This stage , as shown in Figure 3.4, contains:

- an Instruction Register Decoder;
- a Windowed Register File.

This two fundamental units will be examinated in the next paragraphs.

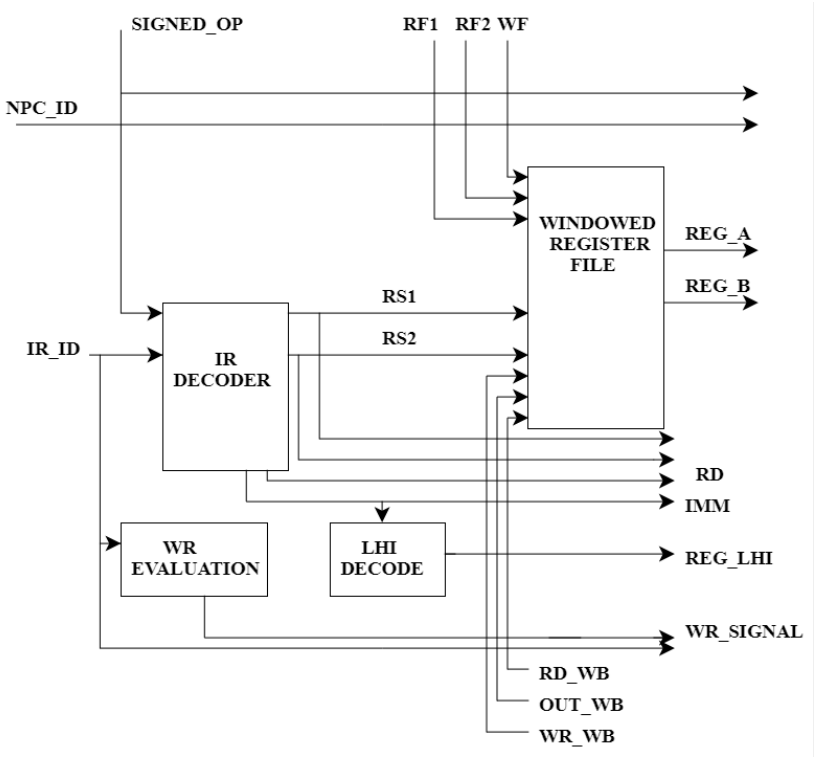


Figure 3.4: Decode stage components.

The output of this stage are:

- the temporary **registers A** and **B** (from the RF),
- the **Immediate** (from the Decoder),
- 2 signal that are needed for the correct execution of all instruction:
  - **WR_signal**: this bit is set to '0' in order to avoid every type of operation during the nop instruction, including the writing. In this stage it's computed its value and it will block the write back to the RF.
  - **LHI_reg**: is a register needed only if an LHI instruction will be execute. It's value is given by moving the 16-bit immediate to the most significant half of the register and clearing the least significant half. This operation is performed after the IR decoding and the need of this register is evaluated in the EX stage.

3.4.1 Windowed Register File

The register file is the unit where all the data, in form of registers, are stored (Figure 3.5).
The architecture works with 32 registers, but in order to implement a windowed RF the number of physical registers is much bigger (equal to $M + 2 * N * F = 88$ in our case).
In this design there are 8 global register and the length of the circular buffer is equal to 80. The

working window is made of 24 registers and for each call or return it is shifted by 16 forward or backward.
This RF support one synchronous writing and two asynchronous reading ports, to avoid delays and to read the two source registers simultaneously.

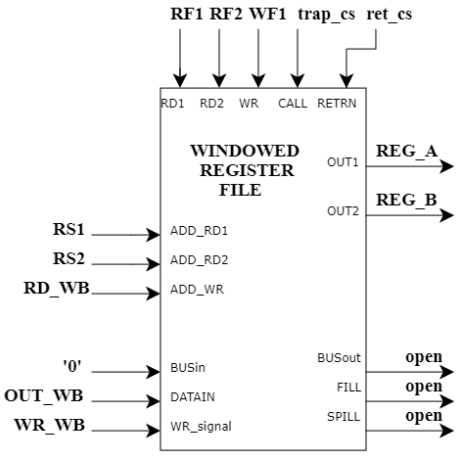


Figure 3.5: Windowed register file interface.

3.4.2 IR decoder

This logic unit is needed to translate the IR into other separated different registers and to extend the Immediate generated into a word register.
The registers RD, RS1, RS2 and Immediate of R-type and I-type instructions are assigned as described in Figure 1.1 and 1.2.
The extension of the immediate in case of I-type, JR and JALR instructions is from 16 to 32 bits, in case of J and JAL jumps the extension is from 26 to 32 bits. To perform correctly the extension of the Immediate, the signal 'SIGNED_OP' express if the instruction is signed or not.
Furthermore in case of JAL and JALR the NPC has to be written in the Link register, so R31 is assigned to the destination register RD.
As shown in Figure 3.6, the Decoder interface is composed by:

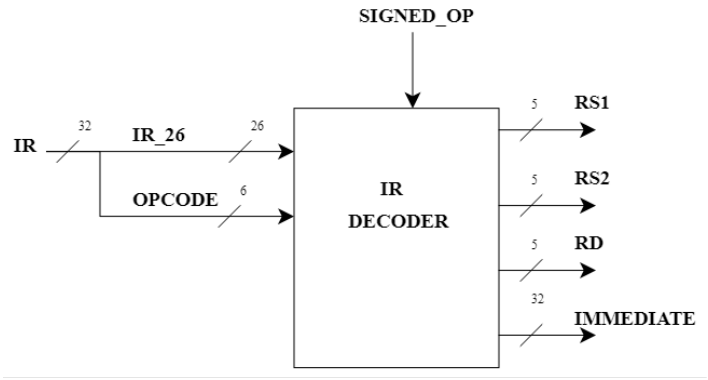


Figure 3.6: Instruction Register Decoder interface.

- IR_26: first 26 bits of the Instruction register ;
- Opcode: 6 most significant bits of the Instruction register;
- SIGNED_OP: bit from the Control Unit that indicates if the instruction is signed;
- RS1, RS2, RD (5 bits): output registers of the decoder.
- Immediate(32 bits): extended version of the output register immediate of 16/26 bits;

The IR DECODER is implemented in vhdl with an if-else structure, based on the type of instructions. Internally some costants are defined in order to compare them to the OpCode.

ID/EX REGISTERS
These are the Decode/Execute pipeline registers:

- NPC_ex: stores the next program counter;
- IR_ex: stores the instruction register;
- regA_ex: stores the temporary register A;
- regB_ex: stores the temporary register B;
- regImm_ex: stores the Immediate;
- regLHI_ex1: stores the LHI register;
- regLHI_ex: stores the LHI register, it is used to synchronize it with the output of the alu;
- RD_ex: stores the destination register in case of write back;
- WR_signal_ex: stores the WR_signal;
- signed_op_ex: store the signal signed_op, which indicate if the operation is between signed or unsigned values;

3.5 Execute stage

This stage contains (Figure 3.7):

- an ALU;
- a comparator named 'ZERO';
- a branch handler 'COND';
- 3 MUXes.

The control signals of the MUXes are coming from the CU. The input of the ALU are selected thanks to 2 MUXes, the output of the ALU goes into the other MUX in order to handle the LHI instructions and to propagate the right register to the next stage.
The output of this stage are:

- the bit **'cond_bit'** (from the COND),
- the register **'ALU_EX'** (from the last MUX),

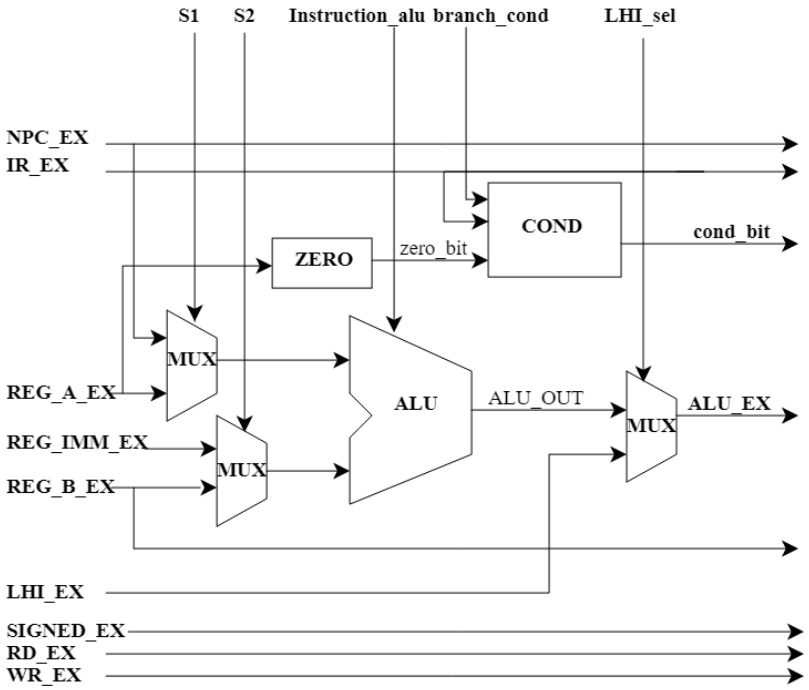


Figure 3.7: Execute stage components.

3.5.1 ALU

The ALU (Arithmetic logic unit) is the the place where all the operations are executed both arithmetic and logic.
To perform all these instructions our ALU (Fig.3.8) is composed by 4 components:

- a generic shifter;
- a comparator;
- a Pentium4 Adder/Subtractor;
- a Logic component.

Pentium4 Adder/Subtractor
This unit consists of a 32-bit adder/subtractor that is the same version that we did during the lab session; we only negated the second operand and put the carry-in to 1 when the operation is a subtraction.

Generic shifter
In order to implement the needed shift operations the givenn generic shifter was used.

Comparator
It is the comparator presented during the lectures and it is a little bit different because instead of using a component approach, the implementation is described in a behavioral way. It takes the output of the Pentium4 Adder/Subtractor and depending on the operations requested and the value of the sum and carry out assign 0 or 1 to the output.

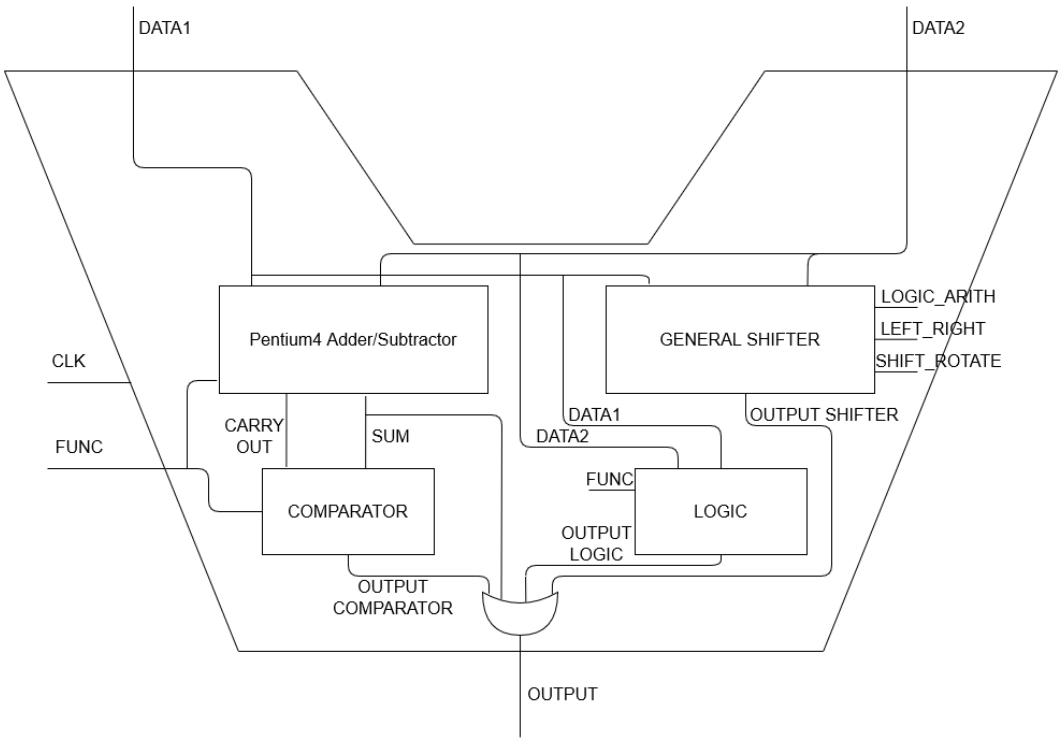


Figure 3.8: Designed ALU components.

Logic
This component in base of the type operation that it has to perform executes a bitwise logic operation between the two inputs.

3.5.2 Zero

This functional unit is (fig. 3.9 needed to evaluate if the temporary register A is equal to zero: the output bit *zero_bit* is '1' when the input *reg A* is zero, otherwise is set to '0'.

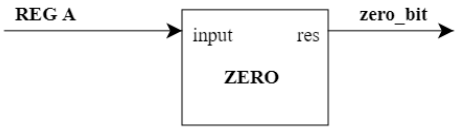


Figure 3.9: Comparator interface.

3.5.3 Cond

This functional unit is needed to evaluate if a branch is taken or not. *branch_op* is '1' when there is a branch instruction.Furthermore the branch is taken when:

- in *beqz* (OpCode 0x04) when register A is equal to 0
- in *bnez* (OpCode 0x05) when register A is not 0

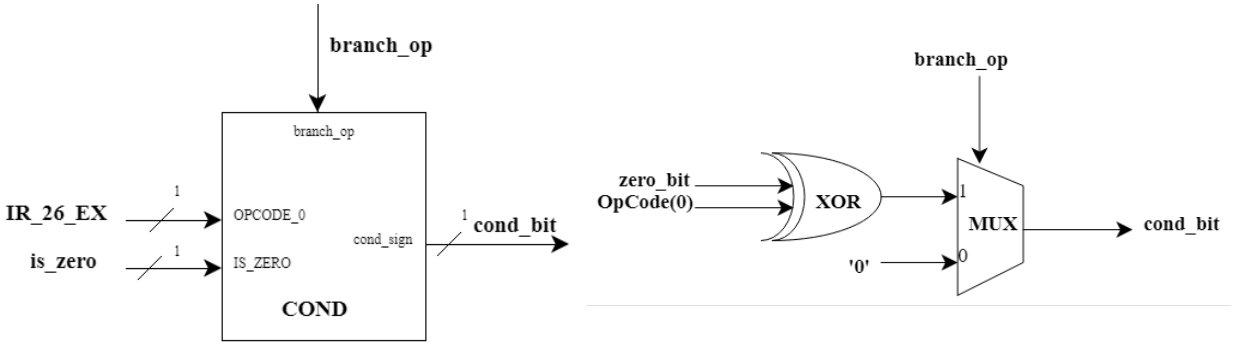


Figure 3.10: Cond unit interface (left) and internal components (right).

This branch handler was implemented as in Figure 3.10.

EX/MEM REGISTERS
These are the Execute/Memory pipeline registers:

- NPC_mem: stores the next program counter;
- IR_mem: stores the instruction register;
- regB_mem: stores the temporary register B;
- RD_mem: stores the destination register in case of write back;
- WR_signal_mem: stores the WR_signal;
- signed_op_mem: store the signal signed_op, which indicate if the operation is between signed or unsigned values;
- cond_mem: stores the value of 'cond', that indicates if the branch is taken;

3.6 Memory stage

In this stage Store and Load instructions are now executed and the real next program counter is selected, taking into account the conditional and unconditional branches.
The branch prediction of this design must be underlined: the branch is always considered not taken; in case of jumps or taken branches the following instructions will always be set to 'NOP', until the correct NPC is found in this stage.
The principal components are (Fig.3.11):

- the Data Memory;
- the LMD;
- the ENABLE handler;
- 3 MUXes;
- 3 logic components (ORs and AND).

The output of this stage are:

- the **LMD output** (from the LMD);

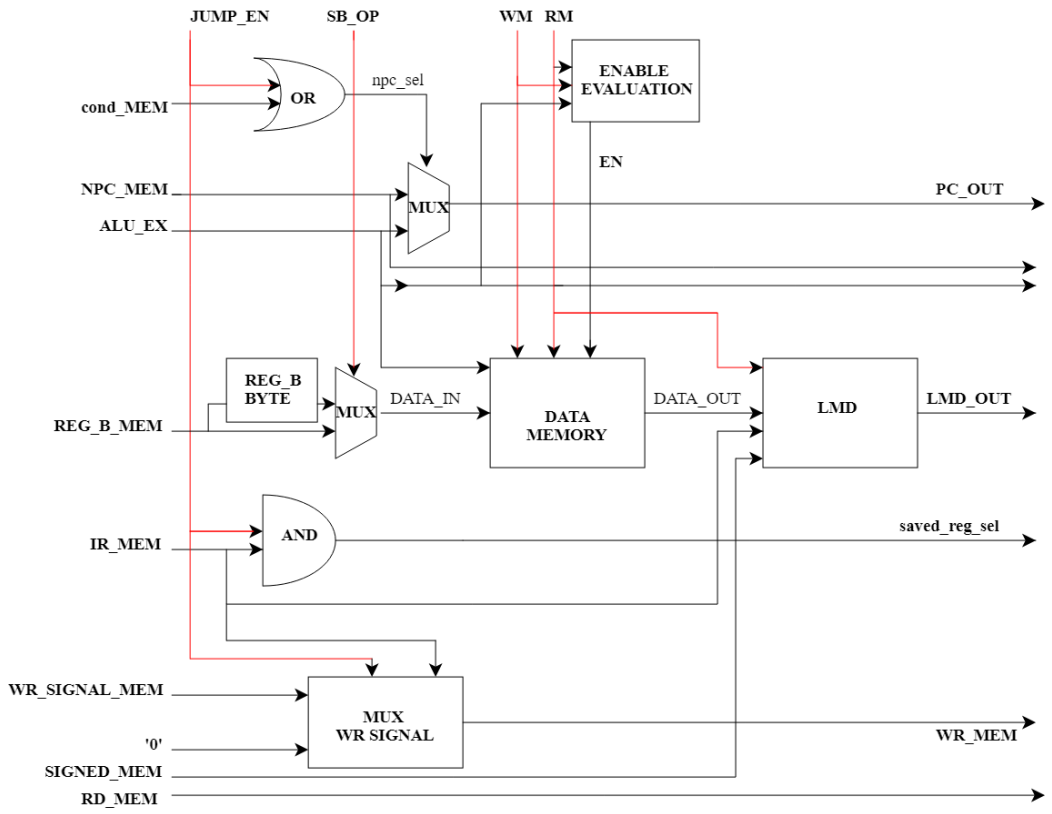


Figure 3.11: Memory stage components.

- the **control signal 'saved_reg_sel'** (from the AND),
  which is set to '1' in case of JAL or JALR instructions and it will allow to store the NPC stored into the R31 register in the following stage;
- the **WR signal** (from the WR MUX),
  which is brought to zero if a jump instruction of type J or JR is present (not the jump and link ones);
- the **real next program counter PC_OUT** (from the npc MUX),
  that will be the target address if the branch is taken or there is jump instruction, otherwise it will simply be the NPC calculated at the first stage.

NB. The MUX present before the DATA MEMORY is needed to handle the store of a Byte. In fact the lower 8 bits of the temporary register B are copied in a register that goes at one port of the MUX.

3.6.1 LMD

The LMD is the LOAD MEMORY DATA, the unit in charge of storing the properly aligned data after accessing the memory in reading mode, i.e. in case of a load. The output register will be used to be written-back to the destination address. The output register LMD_OUT will always be of the dimension of a word, but depending on the type of load instruction it will be filled with the output data memory starting from the LSB. A summary table of the load instruction cases is reported below.

Load_type	Mnemonic	Instruction	LMD_OUT
11	LW	load word	DATA_OUT
01	LHU	load unsigned half-word	$(0)^{16}$## DATA_OUT
00	LB	load signed byte	DATA_OUT$_{(7)}$ ^{16}## DATA_OUT
00	LBU	load unsigned byte	$(0)^{24}$##DATA_OUT
10	-	others	$(0)^{32}$

The LMD is implemented in vhdl with an if-else structure, following the table above. Its interface is composed by (Fig.3.12):

- IR_26_MEM(2 bits) : type of Load instruction, indicated by the last 2 bits of the OPCODE of the IR;
- DATA_OUT: output of the Data Memory;
- SIGNED_MEM: bit that express if the instruction is signed;
- LMD_OUT : output of the LMD.

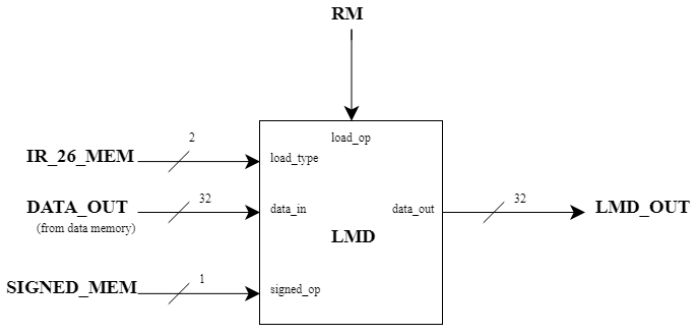


Figure 3.12: Load Data Memory interface.

3.6.2 ENABLE handler

The enable_handler_process is a simple process that set DATA_MEM_ENABLE at 1 if the instruction is a load otherwise set the signal to the result of the logic operation: rm or wm; where rm is the signal of read_from_memory and wm is the signal of write_to_memory. Moreover the enable_handler_process checks if the type of operation is a NOP in this case the DATA_MEM_ADDR is not updated.

EX/MEM REGISTERS

These are the Memory/Write Back pipeline registers:

- NPC_wb: stores the next program counter;
- LMD_wb: stores the output of the LMD;
- ALU_wb: stores the output of the ALU or the LHI register;
- RD_wb: stores the destination register in case of write back;
- WR_signal_wb: stores the WR_signal;
- sel_saved_reg_wb: stores the the value of 'sel_saved_reg', that is the control signal of the mux used for the selection of the data written back to the Register File.

3.7 Write Back stage

This stage only is composed of 2 MUXes in cascade, in Figure 3.13.
The output of this stage are:

- the propagated destination register RD;
- the propagated WR_signal;
- the register 'OUT_WB' which contains the value of the data in case of write back (from the 2 MUXes).

This stage is useful only if the write back will take place, otherwise the registers in this stage will not be used by the Register File in the second stage.

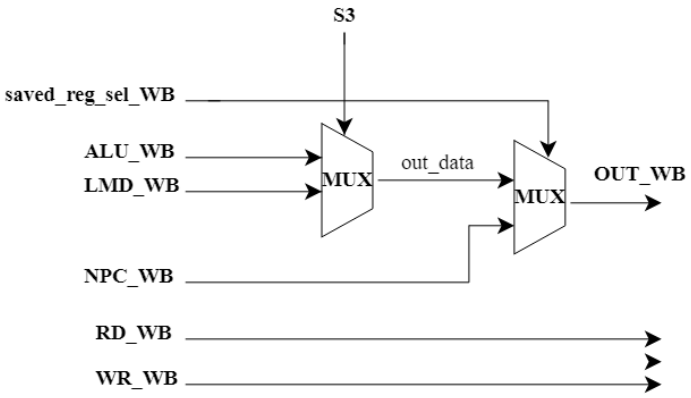


Figure 3.13: Write-Back stage components.

CHAPTER 4

Implementation

4.1 Synthesis

After completing the design and simulation phase can start the synthesis one, using "Design Vision" logic synthesis tool and a 65 nm ST-microelectronics library.
Has been chosen to synthesize two different version of the DLX, one with a simple Register File and one with the Window Register File, and compare them. In the table 4.1 are reported the results obtained by the two versions : $32\,\mathrm{mg\,kg^{-1}}$

Version	Delay [ns]	Area [μm^2]	Power [mW]
DLX-RF	0.49	18945	1.014
DLX-WRF	0.45	36835	2.079

Table 4.1: Comparison between DLX with RF and DLX with WRF

As expected the DLX with WRF has higher area and power consumption.
To choose the operative frequency of the DLX has been performed a static timing analysis where the results for the DLX with the RF are reported in tab 4.2 while for the DLX with WRF can be found in tab 4.3

Frequency [GHz]	Delay [ns]	Slack	Area [μm^2]	Power [mW]
1.7	0.588	-0.11	18798	21.832
1.6	0.625	-0.09	18800	20.562
1.5	0.667	-0.02	18812	19.290
1.4	0.714	0	18822	18.044
1.3	0.769	0	18818	16.778

Table 4.2: STA of the DLX with RF

With frequency higher than 1.5 GHz the slack is violated and indeed only the solutions running at 1.4 GHz and 1.3 GHz are able to meet the constraint.

The solution that ensure a better trade-off is the one at 1.4 GHz.

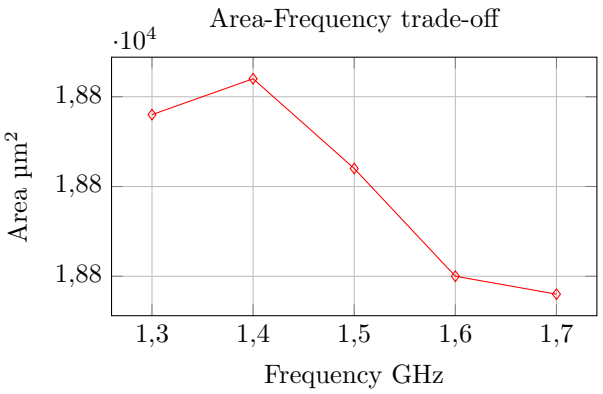


Figure 4.1: DLX RF area vs frequency

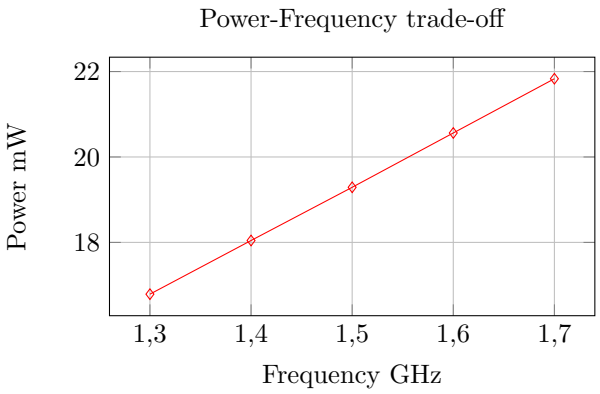


Figure 4.2: DLX RF power vs frequency

Frequency [GHz]	Delay [ns]	Slack	Area [µm^2]	Power [mW]
1.6	0.625	-0.22	39915	43.316
1.5	0.667	-0.15	39973	40.681
1.4	0.714	-0.11	40058	38.077
1.3	0.769	-0.06	39978	35.346
1.2	0.833	0	39886	32.803
1.1	0.833	0	39994	30.11

Table 4.3: STA of DLX with WRF

About the DLX with WRF, as can be observed in table 4.3, with frequency higher than 1.2 GHz the slack is violated and indeed only the solutions running at 1.2 GHz and 1.1 GHz are able to meet the constraint.

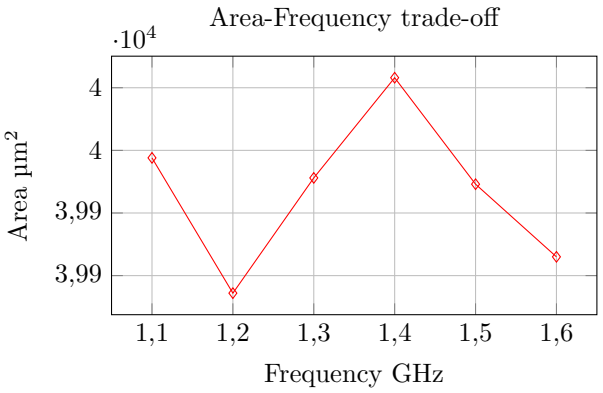


Figure 4.3: DLX WRF area-frequency

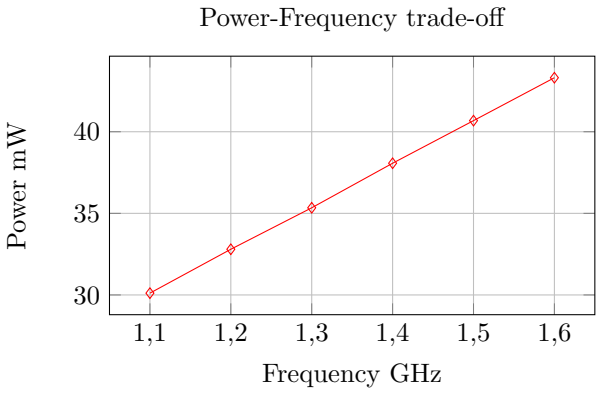


Figure 4.4: DXL WRF area-power

Also in this case has been chosen the solution with higher performance at 1.2 GHz since the gain of power and area of the 1.1 GHz is not very great. The design at the clock frequency chosen are then optimized using the command "compile -map_effort high" and the final results can be observed in tab 4.4

Version	Delay [ns]	Area [μm^2]	Power [mW^2]
DLX-RF	0	19209	18.472
DLX-WRF	0	38268	33.576

Table 4.4: Comparison between DLX with RF and DLX with WRF

CHAPTER 5

Physical Implementation

The final step of the project consist in the implementation of the DLX physical layout, done with Cadence Innovus and using the netlist obtained during the synthesis step.
The final schematic can be observer in figure 5.1 and as can be seen the power is distributed through two rings around the DIE and 12 vertical lines using Metal 9 and Metal 10. The final gate count, with an area of 0.7980 µm2 is reported in table 5.1

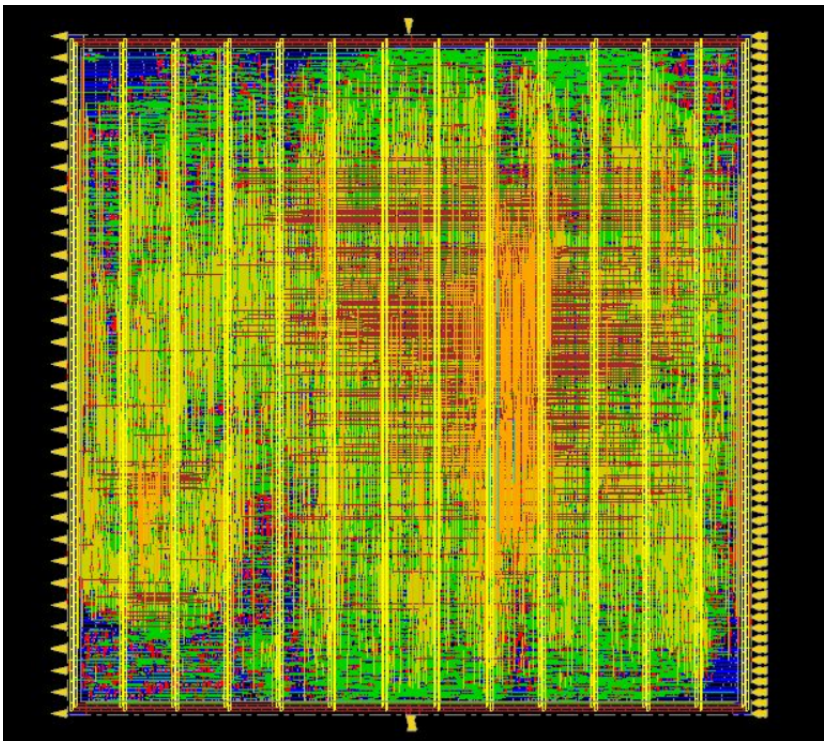


Figure 5.1: Physical layout DLX

Innovus also allows to extract the value of parasitic capacitance and resistance for each metal wire that are important to analyze the behavior of the circuit over time. The complete characterization of each net in the design in terms of its capacitance and resistance network can be found in file DLX_IR_SIZE32_PC_SIZE32.spef file.

LEVEL	Gates	Cells	Area [µm^2]
0 Module DLX	46164	15580	36839.4
1 Module CU	1637	838	130.9
1 Module DTPTH	44224	14604	35290.8

Table 5.1: Gate count of DLX

CHAPTER 6

Discussion and Conclusion

In conclusion, the architecture that we have implemented is a good starting point for a good DLX. In fact our implementation is well performed for several points of view: it is totally pipelined, it can manage many operations between integers and it can easily handle some processes thanks to the windowed register file. Unfortunately to implementing other features would cost much more time and so we do not take care of some features. The architecture, in fact, is a little bit limited in different levels: for example it cannot manage hazards especially control hazard due to the presence of branches; moreover we do not use any kind of power optimization like clock gating.
Another important implementation that could be performed is to increase the number of instructions in the pipeline that is limited due to the delay of the memory. A good solution is to increase the bus of the ROM in this way two instructions can be taken contemporaneously from the memory and they are put in a register, at each clock cycle an instruction can be taken from the register in this way the pipeline can be fully exploited.

APPENDIX A

Synthesis

```
#############################################################
#SCRIPT FOR SPEEDING UP and RECORDING the DLX SYNTHESIS#
#############################################################
exec mkdir -p work
exec mkdir -p report
exec mkdir -p netlist
analyze -library WORK -format vhdl {000-globals.vhd}
analyze -library WORK -format vhdl {01-FFD.vhd}
analyze -library WORK -format vhdl {01-generic_shifter.vhd}
analyze -library WORK -format vhdl {01-iv.vhd}
analyze -library WORK -format vhdl {01-nd2.vhd}
analyze -library WORK -format vhdl {01-reg.vhd}
analyze -library WORK -format vhdl {rocache.vhd}
analyze -library WORK -format vhdl {rwcache.vhd}
analyze -library WORK -format vhdl {a.a-CU_HW.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.a-
    zero_evaluation.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.b-cond_eval.vhd
    }
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.c-sign_ext.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.d-mux21.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.e-mux21_generic
    .vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.g-ir_assigment.
    vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.h-load_eval.vhd
    }
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.i-window_RF.vhd
    }
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.j-comparator.
    vhdl}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.k-logic.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.a-constants.vhd}
```

```
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.b-fa.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.c-g.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.d-p.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.e-lfsr.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.f-pg.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.g-rca.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.h-csb.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.i-carry_generator.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.j-sum_gen.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu.core/a.b.
    l.k-p4_adder.vhd}
analyze -library WORK -format vhdl {a.b-datapath.core/a.b.l-alu2.vhd}
analyze -library WORK -format vhdl {a.b-datapath.vhd}
analyze -library WORK -format vhdl {a-DLX.vhd}
# elaborating the top entity: DLX
set_wire_load_model -name 5K_hvratio_1_4
elaborate DLX -architecture DLX_RTL -library WORK -parameters "IR_SIZE =
    32, PC_SIZE = 32"
# Basic compilation withouth constraints
compile
report_timing > report/DLX_NOPT_TIMING.txt
report_area > report/DLX_NOPT_AREA.txt
report_power > report/DLX_NOPT_POWER.txt
report_power -cell > report/DLX_NOPT_POWER_CELL.txt
report_power -net > report/DLX_NOPT_POWER_NET.txt
# Optimized compilation
compile_ultra
report_timing > report/DLX_OPT_TIMING.txt
report_area > report/DLX_OPT_AREA.txt
report_power > report/DLX_OPT_POWER.txt
report_power -cell > report/DLX_OPT_POWER_CELL.txt
report_power -net > report/DLX_OPT_POWER_NET.txt
# Static Timinig Analysis
#0.588 cycle constraint that corresponds to 1.7 GHz
create_clock -name "CLK" -period 0.588 CLK
set_max_delay 0.588 -from [all_inputs] -to [all_outputs]
compile -incremental
report_timing > report/DLX_TIMING_588.txt
report_area > report/DLX_AREA_588.txt
report_power > report/DLX_POWER_588.txt
```

```
#0.625 cycle constraint that corresponds to 1.6 GHz
create_clock -name "CLK" -period 0.625 CLK
set_max_delay 0.625 -from [all_inputs] -to [all_outputs]
compile -incremental
report_timing > report/DLX_TIMING_625.txt
report_area > report/DLX_AREA_625.txt
report_power > report/DLX_POWER_625.txt
#0.667 cycle constraint that corresponds to 1.5 GHz
create_clock -name "CLK" -period 0.667 CLK
set_max_delay 0.667 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental
report_timing > report/DLX_TIMING_667.txt
report_area > report/DLX_AREA_667.txt
report_power > report/DLX_POWER_667.txt
#0.714 cycle constraint that corresponds to 1.4 GHz
create_clock -name "CLK" -period 0.714 CLK
set_max_delay 0.714 -from [all_inputs] -to [all_outputs]
compile -incremental
report_timing > report/DLX_TIMING_714.txt
report_area > report/DLX_AREA_714.txt
report_power > report/DLX_POWER_714.txt
#0.769 cycle constraint that corresponds to 1.3 GHz
create_clock -name "CLK" -period 0.769 CLK
set_max_delay 0.769 -from [all_inputs] -to [all_outputs]
compile -incremental
report_timing > report/DLX_TIMING_769.txt
report_area > report/DLX_AREA_769.txt
report_power > report/DLX_POWER_769.txt
#0.833 cycle constraint that corresponds to 1.2 GHz
create_clock -name "CLK" -period 0.833 CLK
set_max_delay 0.833 -from [all_inputs] -to [all_outputs]
compile -incremental
report_timing > report/DLX_TIMING_833.txt
report_area > report/DLX_AREA_833.txt
report_power > report/DLX_POWER_833.txt
#0.909 cycle constraint that corresponds to 1.1 GHz
create_clock -name "CLK" -period 0.909 CLK
set_max_delay 0.909 -from [all_inputs] -to [all_outputs]
compile -incremental
report_timing > report/DLX_TIMING_909.txt
report_area > report/DLX_AREA_909.txt
report_power > report/DLX_POWER_909.txt
# Selected the clock that gives best performaces

# saving files
write -hierarchy -format ddc -output netlist/DLX.ddc
write -hierarchy -format vhdl -output netlist/DLX.vhdl
write -hierarchy -format verilog -output netlist/DLX.v
write_sdc netlist/DLX.sdc
```