Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development
Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group 11

Stefano Loviselli s332681, Dennis Teran Polenta s317525, Giulia Di Fante s331555

October 16, 2024

Contents

CHAPTER 1

Introduction

The purpose of this project is the design of a DLX, starting from the VHDL level and going down to the synthesis and physical design phase. Our group chose to implement the basic version of the microprocessor.

The DLX has a typical RISC architecture, with 32 32-bit general-purpose registers and two memories (one for the instructions and one for the data) which are byte addressable in Big Endian mode with a 32-bit address.

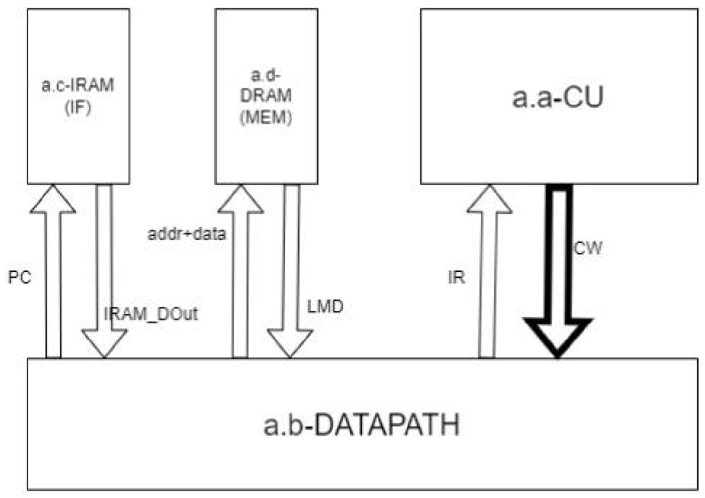


Figure 1.1: external structure of DLX

CHAPTER 2

Background

The DLX is a RISC processor architecture, originally designed by John L.Hennessy and David A. Patterson. It uses only two addressing modes: immediate and displacement. All the instructions are 32 bits long, with a 6-bit primary opcode. They are grouped in three types: I-type, R-type, J-type. I-type are load and store instructions, operations with immediates or conditional branches. R-type are ALU register to register operations, where FUNC defines which is the operations. J-type are normally jump, trap or return from exception.

Every DLX instruction can be implemented in five clock cycles: IF ID EX MEM WB, as shown in Figure 2.1.

Instruction cycle	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				IF	ID	EX	MEM	WB
i+4					IF	ID	EX	MEM

Figure 2.1: Pipe Stages

1. **Fetch:** The processor fetches the instruction from the memory (IRAM) using the address provided by the program counter (PC) . PC is then incremented to point to the next instruction.

2. **Decode:** The fetched instruction is decoded, which means that the processor interprets what needs to be done. At this stage, the necessary registers are read from the register set (Register File). Also, if the instruction involves a conditional jump, the condition is evaluated to determine whether to make the jump.

3. **Execute:** During the execution phase, the instruction is actually executed. This could mean a sum, multiplication, logical operation or comparison. If the instruction is a memory access operation (load/store), the address is calculated at this stage. For jump instructions, the jump destination address is determined. In addition, branch instructions are evaluated in this stage.

4. **Memory:** At this stage, the processor accesses memory, but only if the instruction requires it (such as a load or store operation). If the instruction is a load operation, the memory is read; if it is a store operation, the computed value is written to memory.

5. **Write back:** In the last stage, the results of the operations are written to the target register. This is the stage where the output of the instruction is made available for other subsequent instructions.

The pipeline is managed by a *Hard-wired* control unit. Its main features are:

- Velocity: because the logic is hard-wired and does not need to be interpreted or executed as code, hard-wired control units are very fast. This is one of the main reasons they are used in high-performance processors.
- Strictness: once designed, a hard-wired CU is difficult to modify. Adding new instructions or changing the instruction set would require physical modification of the circuit. This reduces the flexibility of the architecture.
- Complexity: to implement a hard-wired CU that supports a complex instruction set, the logic becomes very intricate and difficult to design. This is a limitation for processors that need to perform many complex operations.
- Efficiency: Hard-wired CU is more efficient in terms of hardware resources for specific tasks and easier to optimize for speed.

It was chosen this kind of CU because is more efficient and faster for a RISC-type architecture.

CHAPTER 3

Design

3.1 Datapath

The datapath is composed of the five stages discussed previously. Its implementation is shown in Figure 3.4.

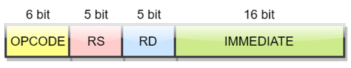
Starting from the left, the **Instruction Fetch** is introduced. It consists of issuing the Program Counter (PC) to fetch the instruction from memory (IRAM) into the Instruction Register (IR). In this implementation the PC represent the number of bytes in the memory, hence, to point to the next instruction the PC has to be incremented by four. This value is then stored in a register called *Next Program Counter (NPC)*.

The **Instruction Decode** consists of decoding the instruction and accessing the register file (RF) to read the registers; in case the instructions of type J they are evaluated in this stage. The *Instruction Register (IR)* and NPC are retrieved from the fetch stage. In particular, the IR is a 32 bit hexadecimal value that contains the OPCODE and the information needed by the datapath . DLX Instructions are grouped into 3 main types, as shown in Figure 3.1a, 3.1b and 3.1c :

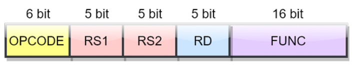
- I-type: Loads/Stores and conditional branch instructions
- R-type: Register-register ALU operations
- J-types: Jump and jump link instructions

It is intuitive that the IR has to be divided into sections and feeded into the *Register File (RF).* To retrieve the two operands A and B, the sections of IR(25 downto 21) and IR(20 downto 15) are sent to the RF respectively. These operands are stored in two registers called A_reg and B_reg, the destination register is stored in a register called RT_reg. For the instructions of type I and J, a immediate value is provided. Its size can vary between 16 or 26 bits, but this application operates on 32 bits, so a sign extend is needed for both (sign_extend block). When a J operation is detected, the control signal J_EN is set and so switching the origin the PC from NPC to NPC+IMM. In this particular part a problem has risen: if there is a conditional branch and right after a unconditional jump, the circuit formed by an AND gate and an inverter gives the priority to the conditional branch.

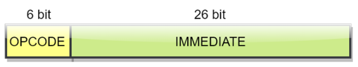
The next stage is the **Execute unit** in where the operands are elaborated to give in output the desired result. The first operand could be between A and NPC, on the other hand the second operand could be between B and IMMEDIATE. With a combination of these four elements the DATAPATH can yield the result of almost every instruction in this implementation, with exception of loads and stores.



(a) I-type instruction



(b) R type instruction



(c) J type instruction

Figure 3.1: Instruction types

- SLL: The value A is shifted left by a number determined by B
- SRL: The value A is shifted right by a number determined by B
- ADD: A + B
- SUB: A - B
- AND: bitwise AND between A and B
- OR: bitwise OR between A and B
- XOR: bitwise XOR between A and B
- SNE: set the register target RT to 1(LSB) if A is not equal to B
- SLE: set the register target RT to 1(LSB) if A is less or equal to B
- SGE: set the register target RT to 1(LSB) if A is greater or equal to B
- ADDI: A + IMM
- SUBI: A - IMM
- ANDI: bitwise AND between A and IMM
- ORI: bitwise OR between A and IMM
- XORI: bitwise XOR between A and IMM
- SLLI: The value A is shifted left by a number determined by IMM
- SRLI: The value A is shifted right by a number determined by IMM
- SLEI: set the register target RT to 1(LSB) if A is less or equal to IMM
- SGEI: set the register target RT to 1(LSB) if A is greater or equal to IMM

In the EXU there is also a custom ZERO block that set a flag called B_EN if the input A is equal to 0 (BEQZ_OR_BNEQ = 1), it works in opposite way if BEQZ_OR_BNEQ control signal is set to 0. The flag B_EN then changes the PC by switching its route from NPC to IMMEDIATE (last multiplexer on this stage).

The **Memory unit** is used for Load and Store instructions data from or to the DRAM. A control signal DRAM_WE is set whenever there is the need to compute a store instruction. Its address is calculated by the ALU in the previous stage. The same control signal is set to 0 whenever there is a load instruction. A register called LMD_LATCH has to be activated to let the content of the DRAM pass through.

In conclusion, there is the **WriteBack unit** that consists of writing the result to the register file, coming from the memory system or the ALU. In the ALU previously the address of the RF was calculated.

The only exception to these structure is Jump And Link, because it is computed on the decode stage but the memorization of the return point in R31 register is done on the WriteBack stage.

3.2 Control Unit

The CU was implemented with an hardwired approach, so it can be seen as a Look-up table. The LUT stores all the control words for every possible instruction, and at each clock cycle, the appropriate control word is fetched and delivered to the data path.

The CU receives in input the IR coming directly from the IRAM, so in a combinational way. It looks at the OPCODE to determine the kind of instruction is dealing with and select the appropriate control word CW. In the first clock cycle it manages both the control signal regarding the fetch and decode stages all together. In particular the CW for the fetch part is set just after the deactivation of the reset. In the clock cycles after, the CW is splitted in to assign values to the control signals of the next stages.

In the presence of a unconditional Jump or an already stipulated conditional branch (B_EN = 1), a flushing system is triggered by setting the next instruction as NOP.

OPCODE identifies the type of the operation, that could be:
SLL
SRL
ADD
SUB
AND
OR
XOR
SNE
SLE
SGE
J
JAL
BEQZ
BNEZ
ADDI
SUBI
ANDI
ORI
XORI
SLLI
NOP
SRLI
SLEI
SGEI
LW
SW

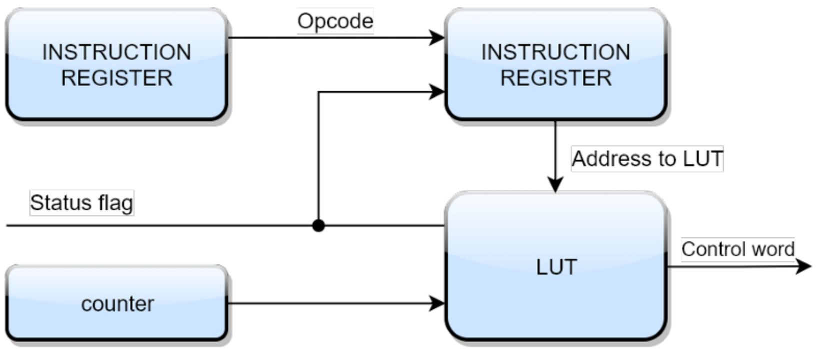


Figure 3.2: Hardwired Control Unit

In case of R-Type instructions the OPCODE indicates only that a mathematical operation must be executed, but the type of the operation is specified by the FUNC field.
The purpose of the control unit is to send the control signal to every pipe stage. In case of a jump, the following instruction needs to be flushed as JTA (jump target address) is computed at decode stage. In case of a branch taken (EX stage) I need to flush two instructions, because two instructions are fetched before PC is correctly updated.

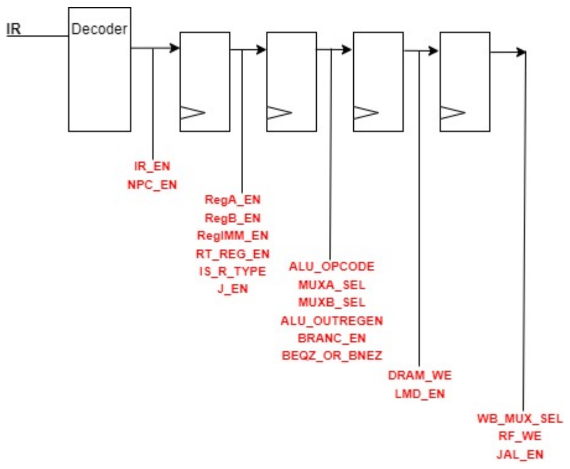


Figure 3.3: CU diagram

3.3 Memories

The structure has two main memories, plus the register file. There is one Instruction memory used only to hold the code instructions and one DRAM that holds data and computation results. This implementation was chosen for efficiency purposes. They area both composed of 32-bits registers. The DRAM has 128 registers (512 byte), while the IRAM has 48 registers (192 byte).
The IRAM receives in input an address that corresponds to the Program Counter and produces in output the Instruction Register. This is an hexadecimal value formed by groups of bits with different purposes and meanings. This memory is used only for read operations (it is not possible to access in write mode).
The DRAM is the main memory that Stores the temporary and persistent data that the program uses and manipulates during execution.
It can be accessed both to read and write. It has a signal WRITEnotREAD to manage these operations. When its high (write mode) the load operation is performed, while when the signal is low (read mode) the content is saved into a register (LDM register) that serves as a buffer.

3.4 Implementation

We chose to implement every single component and then match them together in order to create the datapath stages. Then we created a top level entity for the DLX that unified everything.
This choice was made in order to simplify the design phase and find bugs easier. We reused some of the components already implemented for the labs (with modifications). Most of the components were described in a structural way.
First stage is made by the register that stores the PC, the IRAM that contains all instructions and a module that adds 4 to produce the next PC. The next stage has the register file to store operands and a module to extend the bits of the immediates from 16 to 32 (dimension of each register). The multiplexers are used in case of a different program flow (jump cases). The execution is computed by the ALU (Arithmetic-Logic Unit), which results are stored back to the RF, in the DRAM (in case of load/store instructions) or to the PC (in case of branches).
We insert a lot of buffer registers for pipelining purposes.
Red signals are the control unit inputs to manage the datapath.

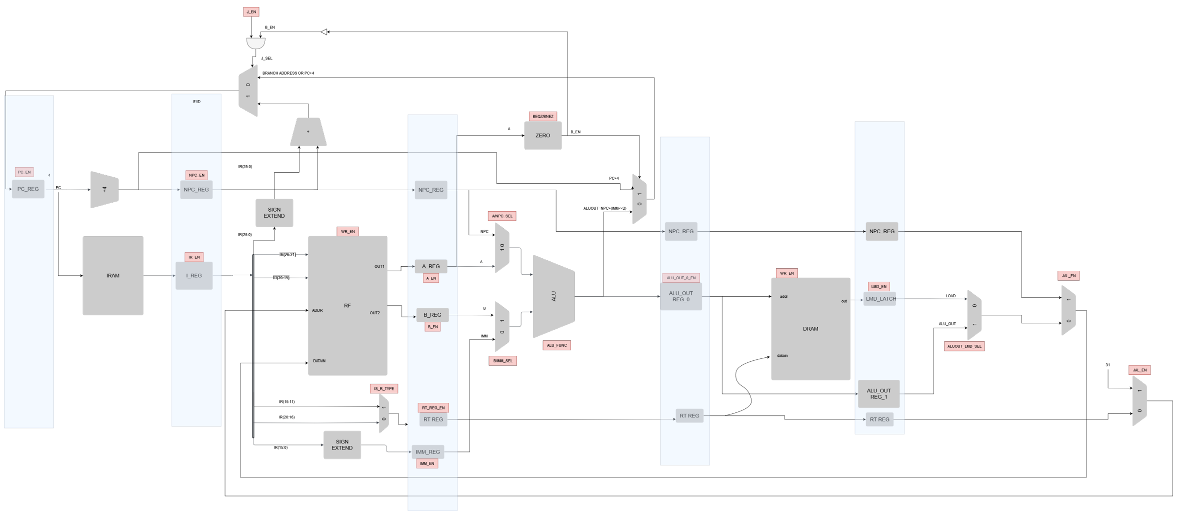


Figure 3.4: DLX schema

3.5 Synthesis

The architecture used for the synthesis is composed of only CU and DATAPATH since the memories IRAM and DRAM are not synthesizable. Figure 3.5 shows the schematic produced after the compilation of the whole system.

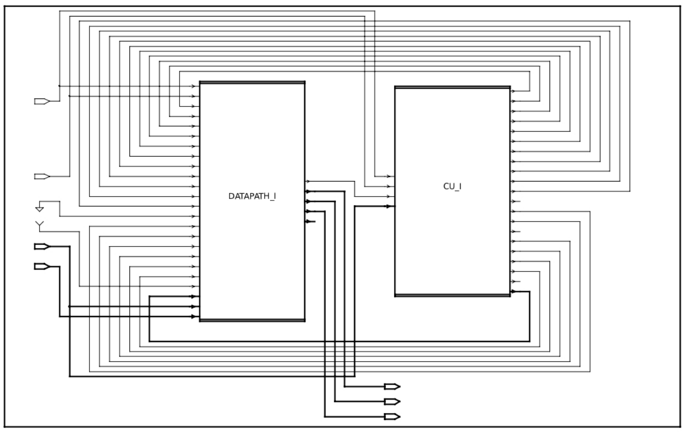


Figure 3.5: Post-synthesis DLX

The synthesis procedure is composed by these steps:

1. The source files are analyzed by the tool
2. Elaboration of the top-level entity (DLX)
3. Map the schematic of the design
4. Analyze the circuit performance in terms of timing, area and power

The synthesis phase translates the parsed VHDL into a flattened RTL (Register Transfer Level) representation. The RTL describes the design logic at a higher abstraction level than gates.

First we computed the synthesis without constraints, and saved the schematics and the results. Since it is a sequential circuit the constraints are applied to the clock of the system. Originally, without constraints, the required time is 0.54ns. The required time is the time that the combinational circuit takes for complete the path between input-output, input-register, register-output or register-register. In our case the critical path is negligible with respect to the delay of the registers, so it is characterized by the register on output of the control word for the fetch-decode part, as shown below:

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : DLX
Version: S-2021.06-SP4
Date   : Tue Oct 15 22:33:39 2024
****************************************

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

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

  Startpoint: CU_I/cw_FU_DU_reg[11]
              (rising edge-triggered flip-flop clocked by Clk)
  Endpoint: CU_I/cw_FU_DU_reg[1]
            (rising edge-triggered flip-flop clocked by Clk)
  Path Group: Clk
  Path Type: max

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

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock Clk (rise edge)                    0.00       0.00
  clock network delay (ideal)              0.00       0.00
  CU_I/cw_FU_DU_reg[11]/CK (DFFR_X1)       0.00 #     0.00 r
  CU_I/cw_FU_DU_reg[11]/QN (DFFR_X1)       0.07       0.07 r
  CU_I/U89/ZN (INV_X1)                     0.03       0.09 f
  CU_I/U103/ZN (NOR2_X1)                   0.05       0.15 r
  CU_I/U100/Z (BUF_X1)                     0.05       0.20 r
  CU_I/U15/ZN (NAND2_X1)                   0.04       0.23 f
  CU_I/U6/Z (BUF_X1)                       0.04       0.27 f
  CU_I/U12/ZN (AND2_X1)                    0.04       0.31 f
  CU_I/U98/ZN (OAI21_X1)                   0.03       0.35 r
  CU_I/cw_FU_DU_reg[1]/D (DFFR_X1)         0.01       0.36 r
  data arrival time                                    0.36

  clock Clk (rise edge)                     0.40       0.40
  clock network delay (ideal)               0.00       0.40
  CU_I/cw_FU_DU_reg[1]/CK (DFFR_X1)         0.00       0.40 r
  library setup time                       -0.04       0.36
  data required time                                   0.36
  ---------------------------------------------------------
  data required time                                   0.36
  data arrival time                                   -0.36
  ---------------------------------------------------------
  slack (MET)                                          0.01

1
```

It was then optimized by reducing the clock period to 0.4ns. In this case the required time of the combinational part is 0.36ns with a slack of 0ns since the library setup time is set to 0.4ns.
Afterward the limit of optimization was reached with a clock period of 0.3ns causing a negative slack, hence a violation, shown below.

```
    ****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : DLX
Version: S-2021.06-SP4
Date   : Tue Oct 15 22:35:10 2024
****************************************

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

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

  Startpoint: CU_I/cw_FU_DU_reg[11]
              (rising edge-triggered flip-flop clocked by Clk)
  Endpoint: CU_I/cw_FU_DU_reg[8]
            (rising edge-triggered flip-flop clocked by Clk)
  Path Group: Clk
  Path Type: max

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

  Point                                     Incr       Path
  ---------------------------------------------------------
  clock Clk (rise edge)                     0.00       0.00
  clock network delay (ideal)                0.00       0.00
  CU_I/cw_FU_DU_reg[11]/CK (SDFFR_X1)        0.00 #     0.00 r
  CU_I/cw_FU_DU_reg[11]/Q (SDFFR_X1)         0.08       0.08 r
  CU_I/U78/ZN (NOR2_X1)                      0.03       0.11 f
  CU_I/U74/ZN (NAND2_X1)                     0.04       0.15 r
  CU_I/U71/ZN (AND2_X1)                      0.05       0.20 r
  CU_I/U70/ZN (NAND2_X1)                     0.03       0.23 f
  CU_I/U126/ZN (OR2_X1)                      0.05       0.28 f
  CU_I/cw_FU_DU_reg[8]/D (DFFR_X1)           0.01       0.29 f
  data arrival time                                     0.29

  clock Clk (rise edge)                      0.30       0.30
  clock network delay (ideal)                0.00       0.30
  CU_I/cw_FU_DU_reg[8]/CK (DFFR_X1)          0.00       0.30 r
  library setup time                        -0.04       0.26
  data required time                                    0.26
  ----------------------------------------------------------------
  data required time                                    0.26
  data arrival time                                    -0.29
  ----------------------------------------------------------------
  slack (VIOLATED)                                     -0.03
```

1

As for the total cell area is 13985.481927μm^2 and for the total dynamic power used is 25.6678 mW and the total power leakage is 274.8364 μW

Note: the main problem to this part was the unreferenced pins because of the absence of IRAM and DRAM. Before we tried to do DATAPATH and CU separately but ultimately we chose to synthesize the together.

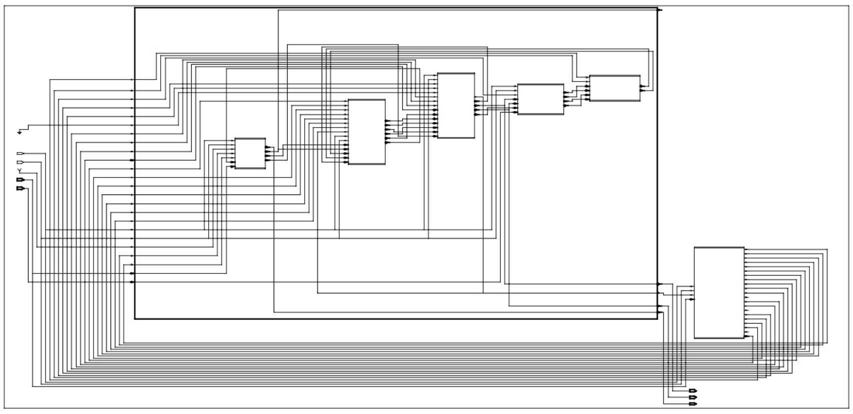


Figure 3.6: Post synthesis datapath

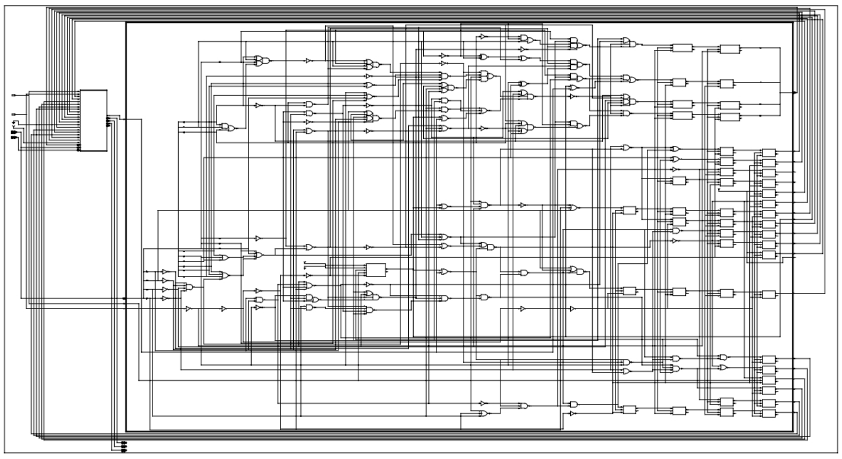


Figure 3.7: Post synthesis control unit

CHAPTER 4

Results

4.1 Physical Design

These general steps were followed:
1. floorplan structuring
2. power rings insertion
3. stripe insertion
4. standard cell power routing
5. placement of standard cell
6. pre-clock tree optimization
7. clock tree generation
8. post clock tree optimization
9. filler placement
10. routing
11. post routing optimization
12. design rule check
13. design rule optimization
14. extraction of timing parameters and parasitics

To perform these phases Innovus tool is used.

In the step 6 it is possible to notice that by analyzing the pre optimized version (just by doing timeDesign -preCTS) 13 paths have violation of slack. After doing the optimization (optDesign -preCTS) no path has any violation.

Then, for clock equalization a clock tree is introduced in step 7. After implementing the tree and doing the timing analysis on the optimized version, it is shown an evident reduce in density in terms of area: from 57.885% to 7.492%.

Below the report post routing is shown for both setup and hold time:

```
    ------------------------------------------------------------
      optDesign Final Non-SI Timing Summary
------------------------------------------------------------


Setup views included:
 default
Hold views included:
 default
```

```
+--------------------+----------+----------+----------+
|     Setup mode     |   all    | reg2reg  | default  |
+--------------------+----------+----------+----------+
|           WNS (ns):|  0.005   |  0.135   |  0.005   |
|           TNS (ns):|  0.000   |  0.000   |  0.000   |
|    Violating Paths:|    0     |    0     |    0     |
|          All Paths:|   26     |   13     |   19     |
+--------------------+----------+----------+----------+

+--------------------+----------+----------+----------+
|     Hold mode      |   all    | reg2reg  | default  |
+--------------------+----------+----------+----------+
|           WNS (ns):| -0.180   |  0.078   | -0.180   |
|           TNS (ns):| -1.910   |  0.000   | -1.910   |
|    Violating Paths:|   13     |    0     |   13     |
|          All Paths:|   26     |   13     |   19     |
+--------------------+----------+----------+----------+

+----------------+------------------------------+--------------------+
|                |             Real             |       Total        |
|    DRVs        +------------------+-----------+--------------------|
|                |  Nr nets(terms)  | Worst Vio |  Nr nets(terms)    |
+----------------+------------------+-----------+--------------------+
|   max_cap      |     0 (0)        |   0.000   |      0 (0)         |
|   max_tran     |     0 (0)        |   0.000   |      0 (0)         |
|   max_fanout   |     0 (0)        |     0     |      0 (0)         |
|   max_length   |     0 (0)        |     0     |      0 (0)         |
+----------------+------------------+-----------+--------------------+

Density: 7.492%
       (100.000% with Fillers)
------------------------------------------------------------------
**optDesign ... cpu = 0:00:11, real = 0:00:14, mem = 1245.1M, totSessionCpu=0:10:40 **
*** Finished optDesign ***
```

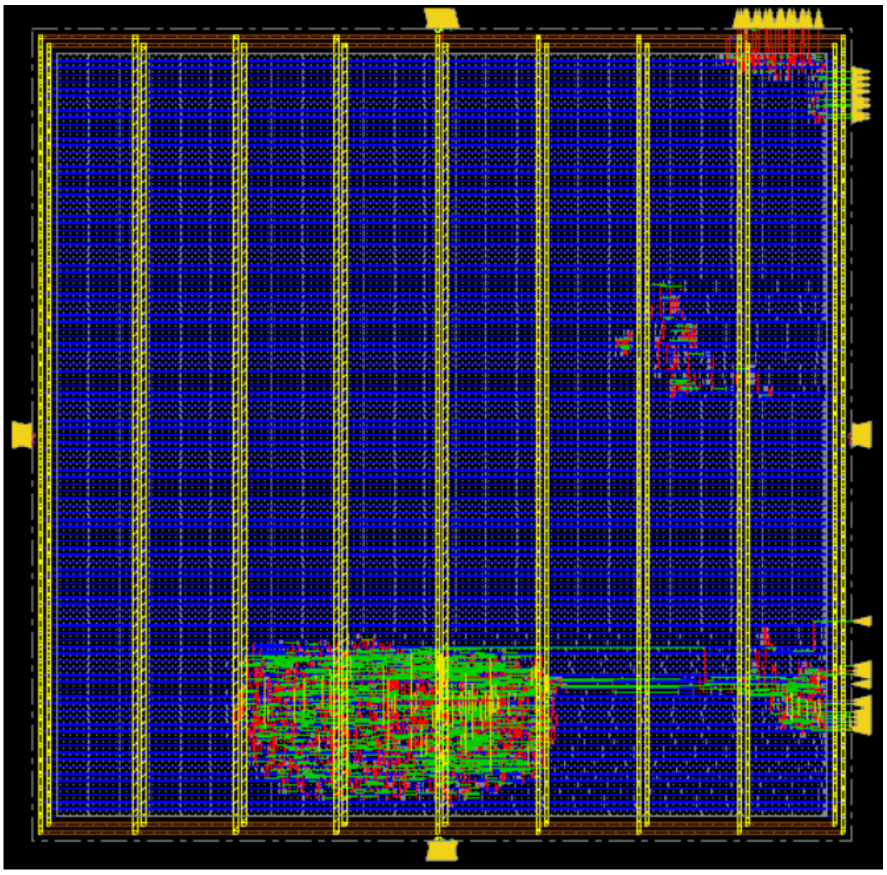


Figure 4.1: Physical design result after optimizations

CHAPTER 5

Conclusions

The development of the project was very useful to understand the characteristics of the DLX and to improve our skills in VHDL. Infact it required to our group to sum all our knowledge to reach the final result.

The tests conducted on the sample programs confirmed the correct operation of the DLX processor, with efficient execution times and expected results. Furthermore, the implementation of the pipeline improved the overall efficiency of the processor, reducing the clock cycles needed to complete instructions.

During development, one of the main challenges was managing the jump and branch instructions, because of the computation of the address. Initially we made it by concatenating NPC-IMMEDIATE-"00", but in this way we obtained wrong addresses, so we implemented ad adder to calculate the right one. Another critical part was to adapt the access the memory to be managed in a right way, infact the calculus of the PC was divided by 4 to not skip any data.

The simulation of all instructions implemented were successful as well as the synthesis. The physical design was the most difficult to understand and to decode since there were a lot of reports to analyze.

To further improve the performance of the DLX processor, future extensions could include the integration of a cache system to optimize memory access and the handling of more complex instructions.

In conclusion, the implementation of the DLX processor has allowed for an in-depth understanding of the fundamental principles of a RISC architecture and the instruction cycle of a processor. The results obtained demonstrated the effectiveness of the optimizations made, and the project provided a solid foundation for future explorations and improvements in the field of hardware architectures.