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_18

Montagna Marco, Pistilli Francesca
TOP-UIC

July 6, 2018

Contents

CHAPTER 1

Introduction

The report contains an accurate description of a 5 stages pipelined DLX processor presented as final project for the course of Microelectronic Systems.
This implementation is the BASIC one.
The DLX is a typical RISC architecture; with RISC - Reduced Instruction Set Computer - we mean an architecture characterized by a low number of instructions, fixed length instructions and simple addressing mode.
In this report we will present the main steps taken for the elaboration of the project, starting from the analysis of the main block, datapath and control unit analysis and ending with synthesis and physical design.

CHAPTER 2

Datapath

2.1 Presentation

Every DLX instruction can be implemented in at most five clock cycles, each of them represent a different step in the evolution of the processor:

1. Instruction Fetch - **IF**;
2. Instruction Decode - **ID**;
3. Execution - **EX**;
4. Memory Access - **MEM**;
5. Write-back - **WB**.

In the following pages all of these phases will be discussed in depth.
First of all we would like to start explaining the use of the colors in the datapath scheme reported in Figure 2.1.1:

- **Black**: components inside the datapath;
- **Blue x**: components outside the datapath, but inside the general DLX project;
- **Green x**: components outside the DLX, they are memories;

This subdivision it has been carried out following the instructions given in the project file (Figure 2.2 on DLX project file); memories are outside the DLX because they can not be synthesized. The processor interfaces to the memories through different signals like: DRAM_ADD, DRAM_DIN, DRAM_DOUT, etc.; same thing regarding the instruction ram.

2.2 Datapath Blocks

In all of the the five stages registers are exploited for the pipelining process of the datapath and muxes are used to select correct values based on the current instruction: muxes have a different number of inputs and registers have different parallelism, but all of them are described in a behavioral way.
Let's move now to a brief description of the main components of the designed datapath.

2.2.1 IRAM & DRAM

As we can see from the Figure 2.1, the *Instruction Ram* is pointed by the *Program Counter* and returns the *Instruction Register*; the IRAM contains all the instructions that have to be executed by the DP. This memory has not been write manually, but through a translated assembler program.
Regarding the *Data Ram* we can see that is addressed by the output of the ALU - that before pass through a register - and the data that has to be written inside is the value B coming from the *Register File*. The output of the DRAM goes to *LMD* register and could be skipped through the multiplexer, selecting the *Out_Alu_2* register.

2.2.2 Register File

Is an array of registers; the dimension is 32 registers per 32 bits each. The inputs of this component are two address port that are driven by a section of the IR, data port which value come from the output of the DP, enables for reading and writing and two output ports.

2.2.3 Sign Extender

This simple block has been created in order to extend on 32 bits the input word. We have two signals in input, the first one indicates if we are handling a jump immediate or not: in the first case the input is on 26 bits and in the second case we are initially working on 16 bits; the second input signal tell us if it is needed an extension of the MSB or of 0s.

2.2.4 Aluopcode Generator

This block of the datapath is a decoder. We have as input *OPCODE* (6 bits) and *FUNCTION* (11 bits), parts of the IR coming from the IRAM. This component computes the signal directed to manage the ALU: sum/sub operation, logic operations etc. It drives the multiplexer in the arithmetic unit.

2.2.5 ALU

This is the arithmetic unit of out processor, it executes the following operations:

- sum - P4 adder;
- subtraction - P4 adder;
- logic operations - behavioral VHDL description;
- comparison;

The outputs of the internal blocks are then selected by a multiplexer driven by the signals coming from the Aluopcode Generator block described before.

2.3 Pipeline Stages

Instruction pipelining is a technique for implementing instruction-level parallelism; as explained before in a typical RISC processor there are five phases (we have also talked about a "pre-phase") and in this section we will explain the functionality of each stage.

2.3.1 Pre-Fetch Stage

Even if is not a phase in the evolution of the pipelined datapath we decided to describe briefly the operation executed in this initial part: the PC register, in normal condition, is updated every clock cycle with the value PC + 4; if jump, jump to a register or branch operations are requested, the PC is updated with the value computed by the adder - sum of the NPC value + *immediate*. As we can see from the schematic of the DP, the multiplexer is driven by an OR port between the *jump_signal* and the signal coming from the ALU.

2.3.2 Instruction Fetch

This is the first stage of the pipeline: here the the normal condition value of the PC is computed and the IRAM is addressed by the Program Counter. The IRAM's output is sent to the IR. The result from the + 4 sum is saved in the NPC1_register (addressable byte memory, big endian).

2.3.3 Instruction Decode

In this second stage the instruction stored in the IR is decoded: the content of the register is a 32 bits instruction which contains different informations that are passed to the DP. Part of the IR address the register file, OPCODE and IMMEDIATE are passed to the *Aluopcode_generator* register and the immediate is also passed to the extender sign block. Another part of the IR contains the address in which we will have to save the data that will be computed in the execution stage; this operations will be executed in the write back stage so this value has to be delayed of three clock cycles - through the use of registers.

2.3.4 Execution Stage

In the third stage of the pipelined datapath is computed if and where to jump. The two multiplexer drive the ALU's input ports through two signals that come from the Control Unit; the Aluopcode_generator, as we said before, generate the controls for the ALU. The value of the port B of the register file, that has been saved in a proper register, go in the second multiplexer and in a B2 register also; this value will be useful in next stage. The value of RD1 has to be delayed of other two clock cycles so is connected to a second register. The ALU return the result of the selected operations and the return value.

2.3.5 Memory Access Stage

In this fourth phase the main part is the Data RAM previously described; it is addressed by the ALU result and the data that has to be written inside is the one stored in the B2 register.
The other part of the stage we are analyzing is the connection for delaying for the last time the write address and a multiplexer that, driven by a signal coming from the hardwired control unit, choose a signal between return and the ALU result.

2.3.6 Write Back Stage

In the last part of the pipelined process the value selected by the last multiplexer (between the one coming from the previous stage and the exit of the Data RAM) goes out from the DP and in connected to the data port of the register file; the address in which we have to store this value is the one delayed of three clock cycles or the value "11111" (used for *jal* instruction).

2.4 Comments

As we can see from the datapath scheme we do not connect any control signal because we have discussed them in the Control Unit chapter. Another difference is that in the VHDL we have the register NPC2 that is doubled outside the datapath, but for simplicity in the schematic we drive the signal from the NPC2 register in the DP to the DLX. The operations executed are the same.

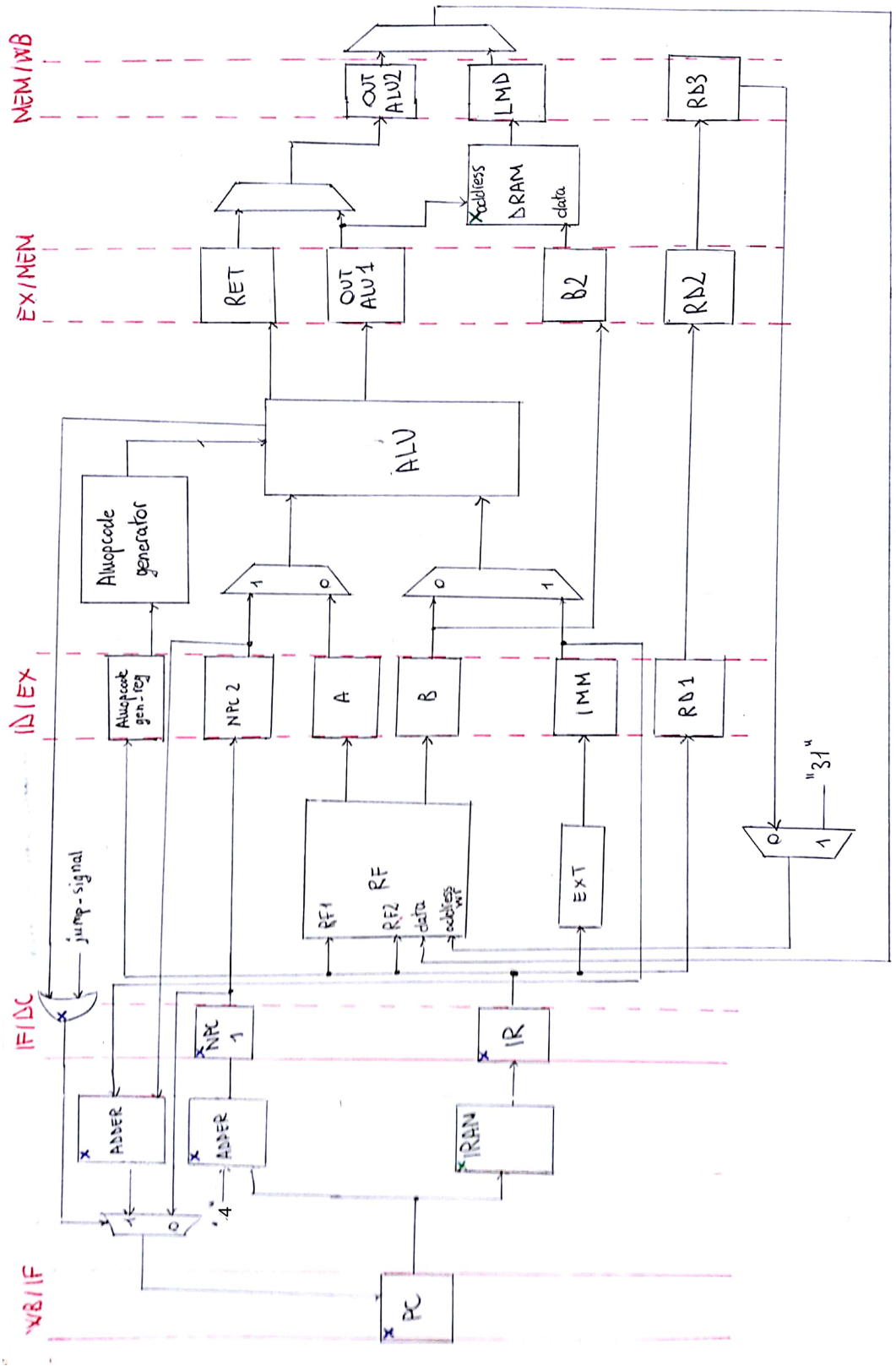

Figure 2.1: Datapath schematic

CHAPTER 3

Control Unit

To implement the control unit there are three classical prototype:

- HARDWIRED
- FSM
- MICROPROGRAMMED

We decided to develop an hardwired control unit, because its structure is quite simple and it is well designed in order to implement the pipeline. The hardwired control unit has in input only the clock, the reset, the instruction register and the signal "flush", the output are all the enable of the register, the selectors of the multiplexers and other useful signals. The control unit is made by a decoder that, in a combinational way, with respect to the "opcode" and "func" (two pieces of the instruction register used for determine the instruction) set a particular values at the vector "cw". This vector contains all the signals that in the various stage of the pipe will be set in order to implement this particular instruction. To set the signal correctly, according to the stage of the pipe, it is used a process were at each clock cycle the vector of the signals shift in a different register that connect the signal related to that stage of pipe to the correct output. Of course the size of these vectors is gradually smaller. In order to synchronize the IRAM, the IR and the PC we decided to update the CU on the falling edge of the clock. Only when a jump is verified, the signal flush will go to the logical value of 1, and on the rising edge the CU will set to zero the two vectors that stores the command of the instructions wrongly fetched.

The signal set by CU are the following, divided by the pipe stage:

- First stage of pipe: enable pc and ir
- Second stage of pipe: enable of registers A, B,IMMEDIATE, RegAluOpCodeGen, NPC2, rd1 and set the signal of read for the register file
- Third stage of pipe:enable of the registers B2, ALUOUTREG,RETURNREG, RegRD2, set the selectors of muxA and muxB and set the jump signal if there is a jump op
- Fourth stage of pipe: set the DRAm write signal, the selector of the output of the alu, and the registers rd3, aluout2, lmd and set the signal to stop the write in the register file in case of jump
- Fifth stage of pipe: sets the operations for the write back, so set the write signal of the register file and the selectors of the mux for the write address

CHAPTER 4

Verification

First of all, in order to test correctly the DLX, a complete_DLX file has been created; in this file the entity is composed only by the *Clock* and *Reset*. The component used in this file are the DLX (in which are instantiated the datapath and the control unit) and the two memories: DRAM and IRAM. Then a test-bench has been created: in this file a clock and reset signal are generated and a few instructions are simulated, in order to test the correct execution of the DLX processor. The program used for the the test is Modelsim in which, through the analysis of the waveforms, we verified that the CU was setting the correct signals and that the datapath work in a desired way. The program given by the professors used to generate the files that will fill the IRAM. The first test is made with the following pseudocode of instructions:

Listing 4.1: Pseudocode

```
addi r1,r1,#1  ;r1 <- 1
addi r2,r0,#2  ;r2 <- 2
addi r3,r0,#3  ;r3 <- 3
addi r4,r0,#4  ;r4 <- 4
addi r5,r0,#5  ;r5 <- 5
addi r6,r0,#6  ;r6 <- 5
addi r7,r0,#7  ;r7 <- 7
sub  r8,r3,r2  ;r8 <- 1
sgei r9,r1,#3  ;r9 <- 0
and  r10,r5,r6  ;r10 <- "00000000000000000000000000000100"
slli r11,r4,#1  ;r11 <- 8
```

In the comment after each instruction is reported the expected results. After the filling of the IRAM, the testbench explained before is used to simulate the machine with ModelSim. Following some capture of the simulations. In the first screen it is visible the beginning of the simulation, so the program counter (PC) that starts to count and the first operations.In the figure is shown the clock, the reset and to understand if the operations really worked the register files.

In the other capture all the operations can be visible.

It is reported a second simulation in order to verify the behavior of the branch instructions. The command of this simulation are the following one:

Listing 4.2: Pseudocode

```
addi r1,r1,#1
addi r2,r0,#2
addi r3,r0,#3
```

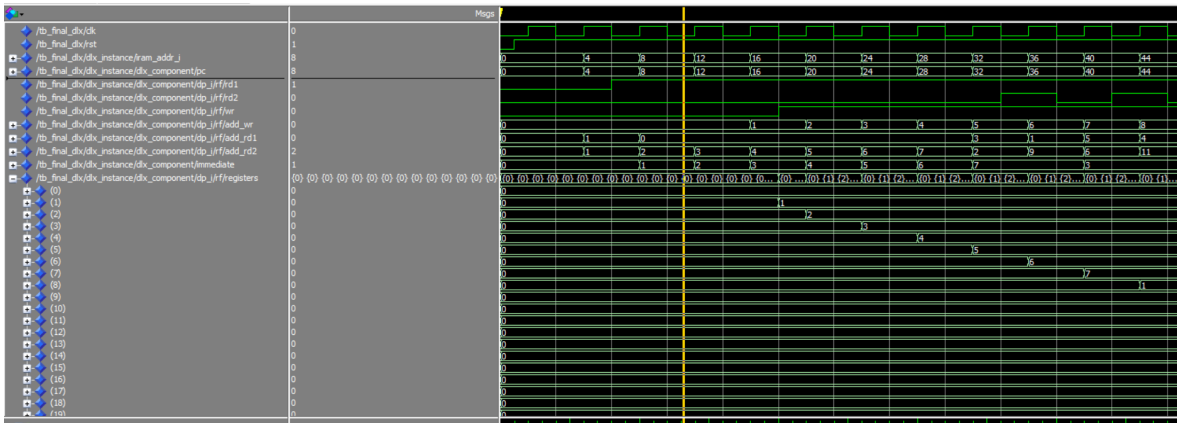

Figure 4.1: Simulation part one

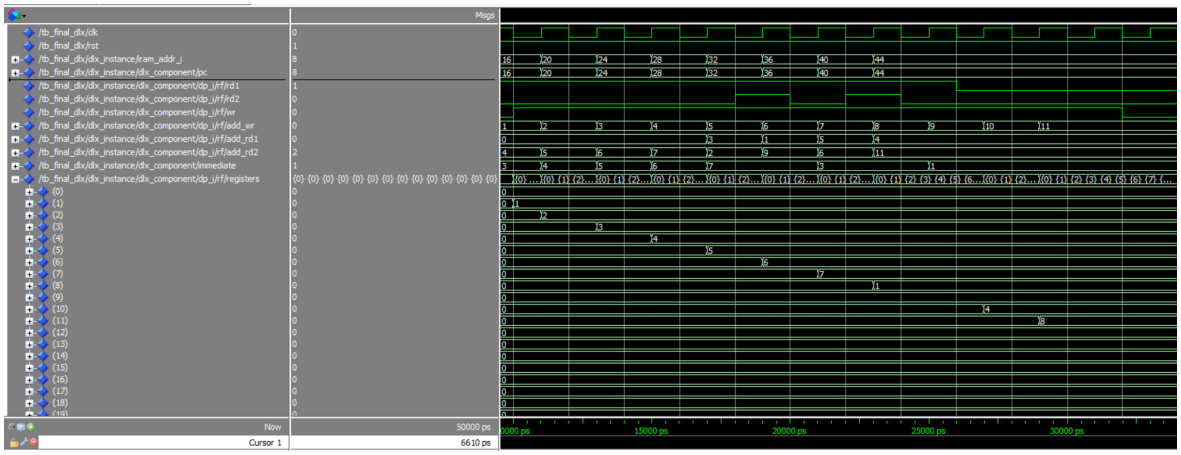

Figure 4.2: Simulation part two

```
addi r4,r0,#4
beqz r0, here
addi r5,r0,#5
addi r6,r0,#6
addi r7,r0,#7
sub  r8,r3,r2
here:
sgei r9,r1,#1
and  r10,r5,r6
slli r11,r2,#1
addi r12,r0,#12
addi r13,r0,#13
addi r14,r0,#14
```

In this case the branch is verified, so we suppose to branch at the address of IRAM of 36. It can be visible from the screens that the operation after the branch are not implemented (infact, register 5, 6, 7 and 8 remain to zero) and the normal flow of the instructions restart from the first instruction after the label of the branch.

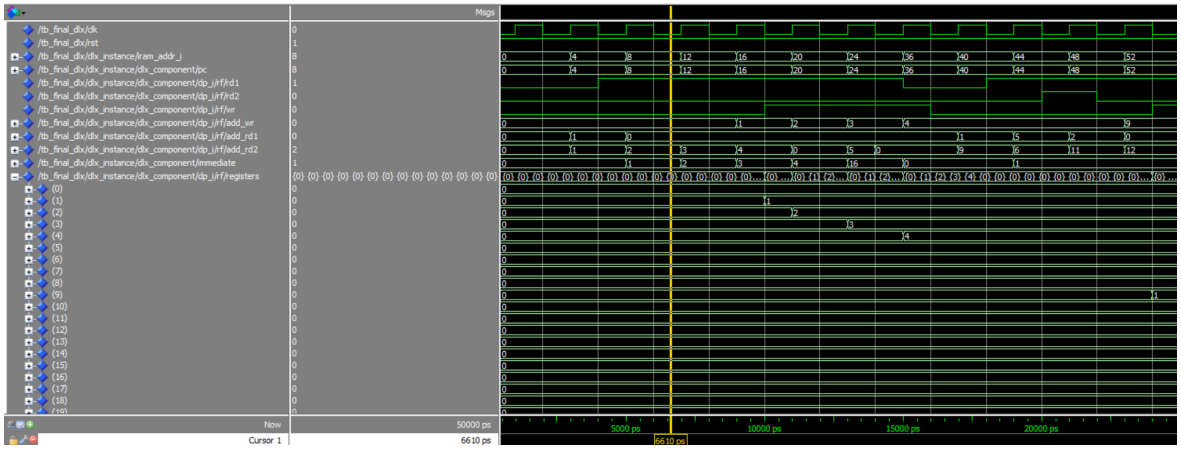

Figure 4.3: Simulation branch part one

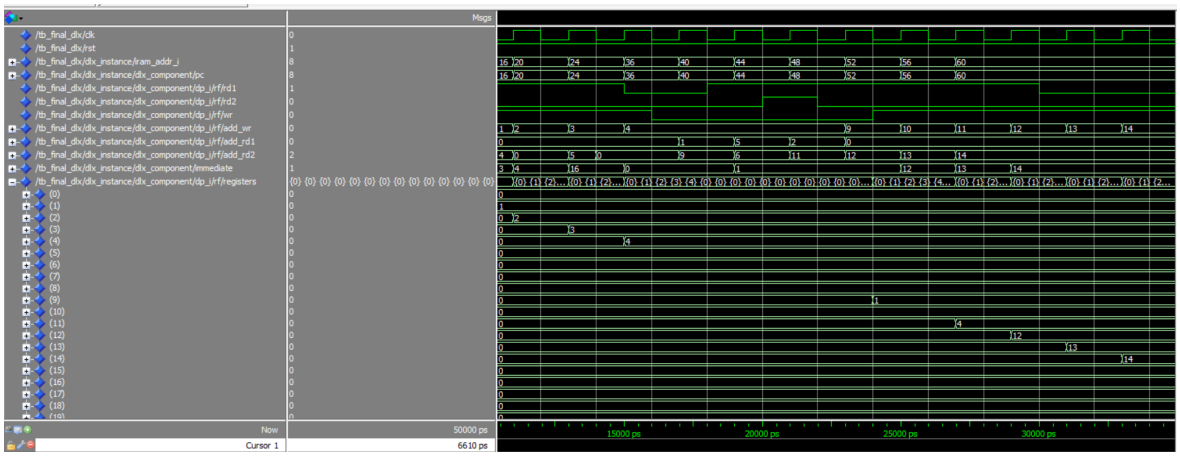

Figure 4.4: Simulation branch part two

CHAPTER 5

Synthesis

After the verification and testing phase, the DLX has been synthesized through Synopsys program. The DLX project is composed by the DP and the CU, while the RAM memories were left outside because they can not be synthesized. After analyzing every component a *.scr file was used in order to perform every step of our synthesis:

Listing 5.1: DLX.scr

```
elaborate DLX -architecture dlx_rtl -library WORK
set_wire_load_model -name 5K_hvratio_1_4
#Forces a clock of period Period connected to the input port CLK #
create_clock -name "Clk" -period 5 {"Clk"}
#forces a combinational max delay of 5 ns from each of the inputs
#to each of th output in case combinational paths are present
set_max_delay 5 -from [all_inputs] -to [all_outputs]
compile -map_effort high
report_power > dlx_timeopt_pw.txt
report_timing > dlx_timeopt_t.txt
report_area > dlx_timeopt_area.txt
write -hierarchy -f verilog -output dlx.v
write_sdc DLX.sdc
```

The clock period was chosen looking at the MAX_PATH_DELAY given by the timing report analysis performed before the optimization. Different timing constraints were tried in order to find the best solution that meets the timing specific (no slack violated). The commands reported below were used for the physical design: a Verilog file and a *sdc file was created and was later used with Cadence Innovus.

```
set_wire_load_model -name 5K_hvratio_1_4
write -hierarchy -f verilog -output dlx.v
write_sdc DLX.sdc
```

We report below the results of our optimized analysis performed on timing, area and power.

Listing 5.2: Timing report

```
Startpoint: CU_I/cw3_reg[15]
            (rising edge-triggered flip-flop clocked by Clk')
Endpoint: DP_I/reg_return/Q_reg[31]
          (rising edge-triggered flip-flop clocked by Clk)
Path Group: Clk
Path Type: max

Des/Clust/Port      Wire Load Model      Library
--------------------------------------------------------
DLX_IR_SIZE32_PC_SIZE32
                    5K_hvratio_1_4       NangateOpenCellLibrary

Point                                               Incr       Path
---------------------------------------------------------------------------
clock Clk' (rise edge)                              2.50       2.50
clock network delay (ideal)                         0.00       2.50
CU_I/cw3_reg[15]/CK (SDFFR_X1)                      0.00       2.50 r
CU_I/cw3_reg[15]/Q (SDFFR_X1)                       0.09       2.59 r
CU_I/MUXA_SEL (CU_HDW_MICROCODE_MEM_SIZE10_FUNC_SIZE11_OP_CODE_SIZE6_IR_SIZE32_CW_SIZE21
                                                    0.00       2.59 r
DP_I/S1 (datapath_reg_size32)                       0.00       2.59 r

-- Here some of the timing paths were removed --

DP_I/reg_return/Q_reg[31]/D (DFFR_X1)               0.01       4.96 f
data arrival time                                              4.96

clock Clk (rise edge)                               5.00       5.00
clock network delay (ideal)                         0.00       5.00
DP_I/reg_return/Q_reg[31]/CK (DFFR_X1)              0.00       5.00 r
library setup time                                 -0.04       4.96
data required time                                             4.96
---------------------------------------------------------------------------
data required time                                             4.96
data arrival time                                             -4.96
---------------------------------------------------------------------------
slack (MET)                                                    0.00
```

As we can see the slack is MET. We also tried to increase the frequency, but in that case slack was violated. This is the best compromise that we were able

Listing 5.3: Area report

```
Number of ports:                  164
Number of nets:                   793
Number of cells:                  383
Number of references:              17

Combinational area:       7451.191986
Noncombinational area:    5864.768177

Total cell area:          13315.959961
```

Listing 5.4: Power report

```
Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   1.2232 mW   (79%)
  Net Switching Power  = 321.8531 uW   (21%)
                         ---------
Total Dynamic Power    =   1.5450 mW  (100%)

Cell Leakage Power     = 259.5069 uW
```

CHAPTER 6

Physical design

The synthesized design has been placed and routed following the instructions given during the laboratories of the course; Cadence Innovus has been used for this goal. After the physical design a performance analysis has been performed; the following parameters has been computed and are reported in the sent files:

- delay;
- EM;
- thermal informations;
- power;
- ...

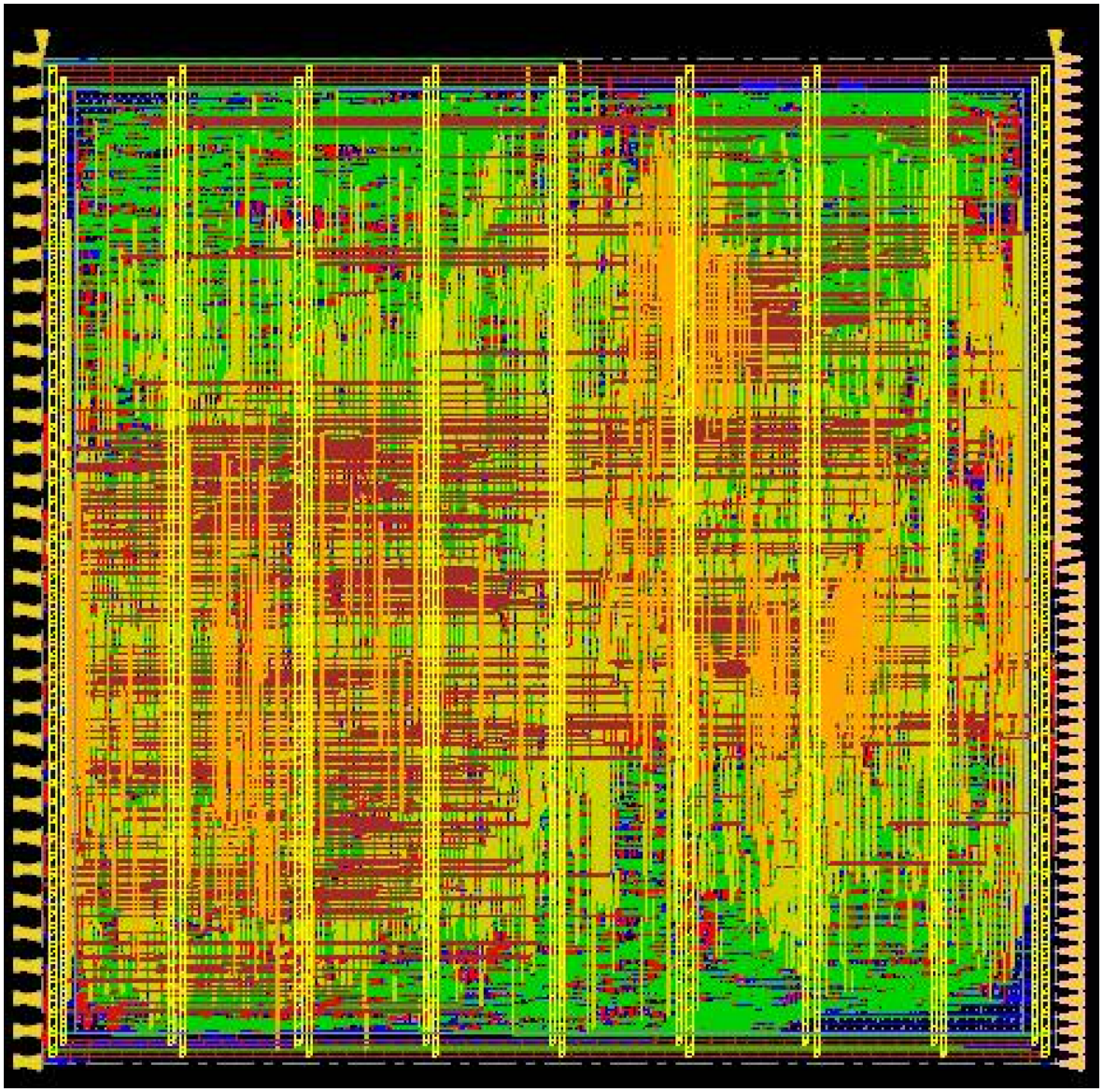

Figure 6.1: Physical Design

CHAPTER 7

Conclusion

If we had more time (TOP-UIC fault) we would have add a lot of more PRO features: the first one is without any doubt the forwarding unit, that is also pretty easy to implement, but due to different problems we did not find the time. One of the more important feature to add could have been the branch prediction block, or a component that could resolve the more critical hazards, as control or data hazards. Another interesting feature that could have been implemented would have been a floating point unit, although it would have been difficult because of our minor preparation in this argument (we always worked with integers). Another thing that could have been added it what we exploit our knowledge earned during the low power course, where we learned how to optimize the power consumption from an architectural point of view, using techniques as parallelizing, pipelining, clock gating and so on.
This project has been really useful because we started from the scratches, creating single blocks, and then assembled the complete DLX; this gave us a different vision regarding the process that stay beyond what we work on every day.
In our VHDL code there are a lot of components that work correctly, but are not implemented: this is because we started with the idea to implement more that a BASIC project. Unfortunately, due to lack of time (TOP-UIC faults again and last minutes presentation for exams), we were not able to implement them.

CHAPTER 8

References

- Microelectronic Systems course, Mariagrazia Graziano
- Google

APPENDIX A

Instruction Subset Implemented

Here we list the subset of instructions supported by our processor:

Instruction	**Opcode**	**Func**
add	0x00	0x20
addi	0.09	-
and	0x00	0x24
andi	0x0c	-
beqz	0x04	-
bnez	0x05	-
j	0x02	-
jal	0x03	-
lw	0x23	-
nop	0x15	-
or	0x00	0x25
ori	0x0d	-
sge	0x00	0x2d
sgei	0x1d	-
sle	0x00	0x2c
slei	0x1c	-
sll	0x00	0x04
slli	0x14	-
sne	0x00	0x29
snei	0x19	-
srl	0x00	0x06
srli	0x16	-
sub	0x00	0x22
subi	0x0a	-
sw	0x2b	-
xor	0x00	0x26
xori	0x0e	-

Table A.1: BASIC instructions