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

Authors: Federico Pozzana
group_44

October 27, 2019

Contents

1 Introduction **1**
- 1.1 Project goal . . . 1
  - 1.1.1 Premise . . . 1

2 DLX components **2**
- 2.1 Combinational blocks . . . 2
  - 2.1.1 Sign extension . . . 2
  - 2.1.2 Arithmetic logic unit . . . 3
  - 2.1.3 Branch prediction unit . . . 3
- 2.2 Clocked logic . . . 4
  - 2.2.1 Hazard detection unit . . . 4

3 DLX introduction **6**
- 3.1 Main components . . . 6
  - 3.1.1 Control Unit . . . 6
- 3.2 Datapth . . . 7
  - 3.2.1 Fetch and decode stage . . . 8
  - 3.2.2 Execution stage . . . 8
  - 3.2.3 Memory stage . . . 9
  - 3.2.4 Write back stage . . . 10

4 DLX synthesis **11**
- 4.1 Synthesis goal . . . 11
- 4.2 Results comparison . . . 13
  - 4.2.1 Synthesis reports . . . 14

5 DLX placement and routing **15**
- 5.1 Place and Route goal . . . 15

A Further improvements **16**
- A.1 Expansion of instruction set . . . 16
- A.2 Improvement of the hazard detection unit . . . 17

CHAPTER 1

Introduction

1.1 Project goal

This project focused on the design, simulation and synthesis of a processor based on a DLX architecture. The report is responsible of introducing the various components used in the design and to explain the interconnections between them. The first chapter introduces to the reader the various components used in the DLX design, the second chapter gives an higher level view of the processor focusing on the connections between every elementary block described in the previous chapter in order to form the datapath and the connection of it with the control unit. The third chapter will focus on the synthesis of the DLX processor on a 45 nm library of gates. The fourth chapter will focus on the routing of the synthesized processor. An appendix is given to explore further improvements possible for this implementation of a DLX processor.

1.1.1 Premise

In this report is given as an initial premise that the reader is familiar with the standard DLX architecture (5 stage pipeline made of fetch, decide, execute, memory and write back) and its relative instruction set decoding (I-type, R-type and J-type instructions with their relative use of OPCODE and FUNC fields). Moreover the knowledge of standard components such as multiplexers, registers, latches etc.. is taken for granted.

CHAPTER 2

DLX components

The aim of this chapter is to familiarize the reader with every block ,both combinational and clocked, used in the final DLX design. As specified before only characteristic components will be analyzed since it has been taken for granted the familiarity of the reader with common components such as multiplexers, latches, registers etc...

2.1 Combinational blocks

RD multiplexer

The RD multiplexer is a component used to address the right destination register whether we have an I-type or R-type instruction. The components has

	in/out	type
I-type_RD	input	std_logic_vector(4 downto 0)
R-type_RD	input	std_logic_vector(4 downto 0)
OPCODE	input	std_logic_vector(5 downto 0)
RD	output	std_logic_vector(4 downto 0)

The RD multiplexer looks at the OPCODE it is fed given by the instruction register; if is is "000000" then we fetched an R-type instruction and the RD output receives the destination register according to the R-type representation, otherwise the RD output receives the destination register according to the I-type representation.

2.1.1 Sign extension

The sign extension module takes the last 16 bits of the instruction register output and extend it to 32 bits. The component has

	in/out	type
imm_in	input	std_logic_vector(15 downto 0)
imm_out	output	std_logic_vector(31 downto 0)

The input is extended to 32 bits as a signed number. The component is used to extend the immediate portion of every I-type instruction to 32 bits in order to be used in the arithmetic logic unit.

2.1.2 Arithmetic logic unit

The arithmetic logic unit his the foundation of the execution stage. My arithmetic logic unit is capable of executing the following operations

- SLL
- SRL
- ADD
- SUB
- AND
- OR
- XOR
- SEQ
- SNE
- SLT
- SGT
- SLE
- SGE

The component has

	in/out	type
FUNC	input	std_logic_vector(3 downto 0)
DATA1	input	std_logic_vector(31 downto 0)
DATA2	input	std_logic_vector(31 downto 0)
DATA_OUT	output	std_logic_vector(31 downto 0)

The FUNC field is tied to the control unit which instruct the arithmetic logic unit to execute the right operation between DATA1 and DATA2. The result is also a 32 bit vector.

2.1.3 Branch prediction unit

The branch prediction unit is the component in charge of deciding the next value for the program counter. The component has

	in/out	type
OPCODE	input	std_logic_vector(5 downto 0)
imm_branch	input	std_logic_vector(15 downto 0)
imm_jump	input	std_logic_vector(26 downto 0)
data_one	input	std_logic_vector(31 downto 0)
data_two u	input	std_logic_vector(31 downto 0)
npc_in u	input	std_logic_vector(31 downto 0)
npc_out	output	std_logic_vector(31 downto 0)

Data_one and data_two are the data coming from the register file as if the register file was addressed by an R-type instruction. The component looks at the OPCODE that receives from the instruction register and perform the correct operation according to it; it can ,if no jump or branch instruction is fetched, propagate the next program counter to the output, otherwise it can perform the following operations

- J
- JAL
- JALR
- JR
- BHI
- BLO

The branch prediction unit is a combinational component, this means that it is not necessary to wait a clock cycle to know the value of the next program counter. The vhdl code for this component can be found in the simulation folder.

2.2 Clocked logic

2.2.1 Hazard detection unit

The hazard detection unit is in charge of detecting RAW data hazard and bypass the correct data to the arithmetic logic unit in order to compute the correct result in presence of I-type and R-type instructions. The component has

	in/out	type
clk	input	std_logic
reset	input	std_logic
OPCODE	input	std_logic_vector(5 downto 0)
RD_IN_R-type	input	std_logic_vector(4 downto 0)
RD_IN_I-type	input	std_logic_vector(4 downto 0)
RS1	input	std_logic_vector(4 downto 0)
RS2	input	std_logic_vector(4 downto 0)
alu_forwarding_one	output	std_logic
mem_forwarding_one	output	std_logic
alu_forwarding_two	output	std_logic
mem_forwarding_two	output	std_logic
RD_out	output	std_logic_vector(4 downto 0)

This component works on both edges of the clock. On the falling edge it samples the OPCODE, sources registers and destination register while on the rising edge it computes the outputs. If the reset signal is high the component stalls. In case of a RAW hazard depending whether the hazard happens at the following clock cycle or the next one the component raises the correct output following this scheme

	clock cycle difference	source register	alu_forward_1	mem_forward_1	alu_forward_2	mem_forward_2
RAW	one	RS1	1	0	0	0
RAW	one	RS2	0	0	1	0
RAW	one	RS1 and RS2	1	0	1	0
RAW	two	RS1	0	1	0	0
RAW	two	RS2	0	0	0	1
RAW	two	RS1 and RS 2	0	1	0	1

The values forwarded to the input of the arithmetic logic unit are the value of the output register of the execution stage and a delayed version of it with the use of another register. This allows the execution of programs with RAW stalls without inserting NOP operations between instructions in order to delay the need of the result of a specific register for both I-type and R-type instructions.

CHAPTER 3

DLX introduction

3.1 Main components

The main components of a DLX processor are

- Datapath
- Control unit
- Data memory
- Instruction memory

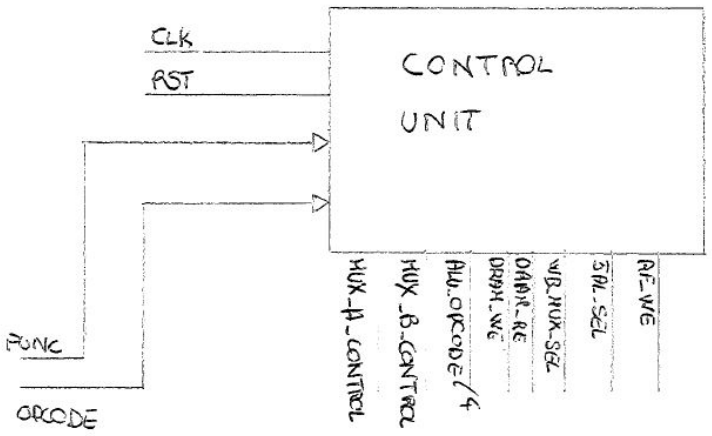
The DLX , in fact, follows the Harvard architecture ; it has two distinct memories, one for data (DRAM) and one for instructions (IRAM). The control unit is the component that control the datapath in order to produce the right results. The datapath is the component that is in charge of executing the instructions placed in the IRAM. Since IRAM and DRAM are pretty standard components hereinafter will be analized only the control unit and the datapath.

3.1.1 Control Unit

The control unit for my DLX architecture implementation is an hardwired one; there are two memories that store the implementation for R-type and non R-type instruction that are addressed by the control unit after having fetched each operation. Depending on the OPCODE an instruction is decoded and the outputs of the control unit are set ath the right clock cycle according to the instruction. Namely these outputs are

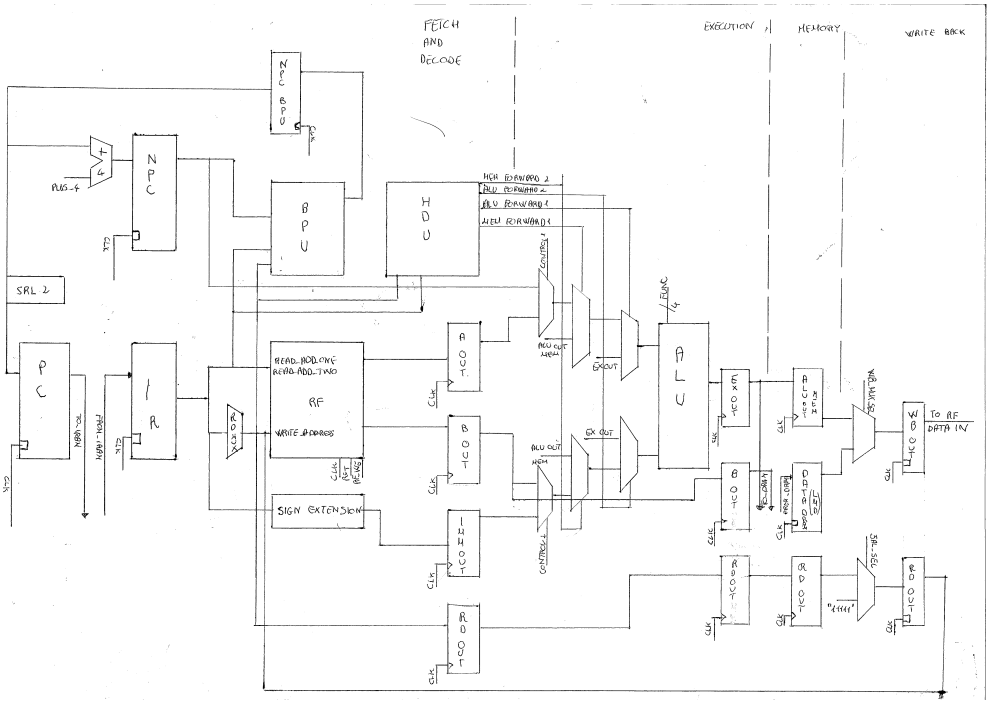
- MUXA_CONTROL at cc 1
- MUXB_CONTROL at cc 1
- ALU_OPCODE at cc 1
- DRAM_WE at cc 2
- DRAM_RE at cc 2
- WB_MUX_SEL at cc 3
- JAL_SEL at cc 3
- RF_WE at cc 3

A schematic representation of the control unit is the following



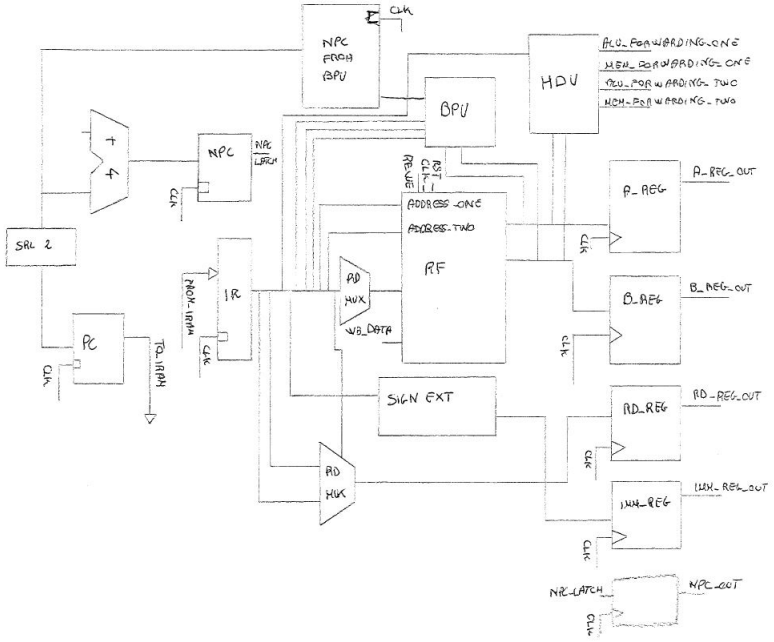
3.2 Datapth

The datapath for this implementation doesn't follow the standard 5 stage pipeline (FDEMW); instead ,in order to reduce the latency by one clock cycle, the fetch and decode stage were put together in order to achieve a smaller latency without affecting too much the throughput. In fact in the following chapter for an implementation at 32MHz (a typical clock frequency for low power applications) the slack is not only met, but is met with a margin of more than 15 ns. An overall schematic representation of the datapath is the following



3.2.1 Fetch and decode stage

The fetch and decode stage is a stage created modifying the standard fetch stage and decode stage; this pipeline stage is obtained by the original one by transforming the IR and NPC from register to latches. This lets one instruction fetched at clock cycle 1 to be decoded during the same clock cycle. This pipeline stage is also responsible of deciding the next program counter, task that is managed by the branch prediction unit (explanation in chapter 2); the npc computed by the branch prediction unit is fed to a register in order to shift the result by one clock cycle and allowing to fetch and decode one instruction every clock cycle. The fetch and decode stage also has embedded the hazard detection unit (explanation in chapter 2) that detect RAW hazards in consecutive and non consecutive R-type and I-type operation. By having the hazard detection unit in the pipeline lighten the workload of the control unit. A schematic representation of the control unit is the following



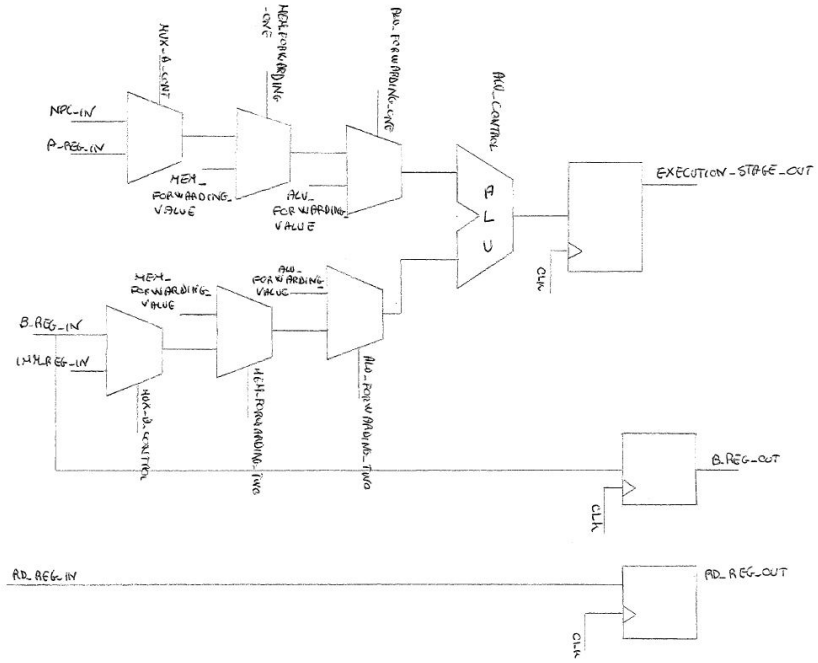
3.2.2 Execution stage

The execution stage is organized with six multiplexers, three registers and one arithmetic logic unit. The first register takes as input the rd_reg output coming from the fetch and decode unit, thus shifting it in time by one clock cycle. The second register takes as input the data coming from the second output of the register file; this shift the data by one clock cycle, data which will be used in LOAD/STORE instructions since it is fed to the data in of the DRAM. The six multiplexers are used to feed the arithmetic logic unit with the right data between

- first output of the register file
- second output of the register file
- immediate register coming from the fetch and decode stage
- npc_in coming from the fetch and decode stage
- alu_out coming from the execution stage register output

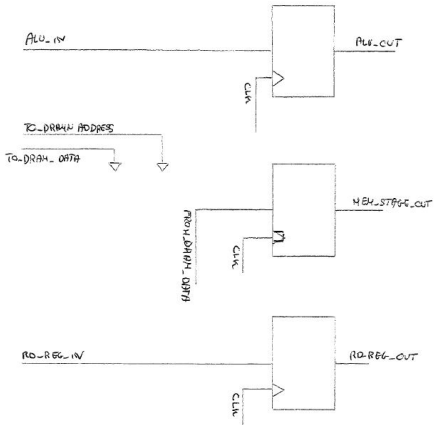
- alu_out coming from the shifted execution stage register output placed in the memory stage

A schematic representation of the control unit is the following



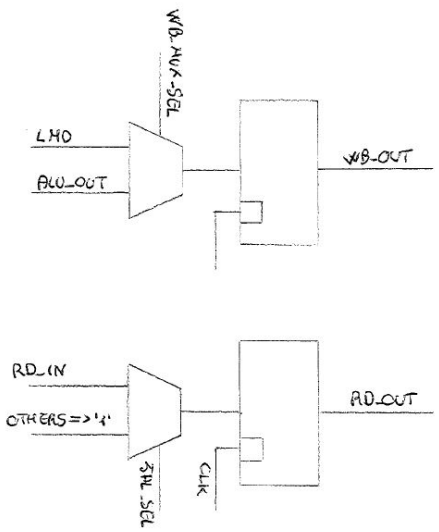
3.2.3 Memory stage

The memory stage is organized with two registers and one latch. The latch receives as input the data coming from the DRAM. The first register takes as input the data coming from the execution unit out; this means that holds the alu_out result shifted in time by one clock cycle. Similarly the last register takes as input the data coming from the rd_reg_out coming from the execution unit. A schematic representation of the control unit is the following



3.2.4 Write back stage

The write back stage is organized with two multiplexers and two latches. The first multiplexer takes as input the data coming from the DRAM and the alu output coming from the memory stage; the selection signal for this multiplexer is WB_MUX_SEL, which decides whether the instruction was a LOAD/STORE or else. This signal is routed to a latch to hold the information and the output of the latch is routed to the data in of the register file. Similarly the second multiplexer has as inputs the RD register coming from the memory stage and (others =¿ '1'); the selection signal is JAL_SEL which desides if the instruction is a JALR/JAL or not. A schematic representation of the control unit is the following



CHAPTER 4

DLX synthesis

4.1 Synthesis goal

The synthesis goal is to map the DLX designed to a real 45 nm library. With the synthesis process I had two goals in mind; to find a fast implementation with a high clock frequency and one low power implementation, where clock speed and total power parameters were taken into account. The result displayed hereinafter are the result of multiple synthesis processes to yield an acceptable result considering the two initial goals.

Here there is the script used in the synthesis part.

```
############################################################
#SCRIPT FOR SPEEDING UP and RECORDING the DLX SYNTHESIS#
# analyzing and checking vhdl netlist#
# here the analyze command is used for each file from bottom to top #
############################################################
analyze -library WORK -format vhdl {log_pkg.vhd}
analyze -library WORK -format vhdl {000-globals.vhd}
analyze -library WORK -format vhdl {myTypes.vhd}
analyze -library WORK -format vhdl {inverter.vhd}
analyze -library WORK -format vhdl {nand2.vhd}
analyze -library WORK -format vhdl {mux21_generic.vhd}
analyze -library WORK -format vhdl {alu.vhd}
analyze -library WORK -format vhdl {RDSelect.vhd}
analyze -library WORK -format vhdl {RdMux.vhd}
analyze -library WORK -format vhdl {Boffset.vhd}
analyze -library WORK -format vhdl {Joffset.vhd}
analyze -library WORK -format vhdl {BranchPredictionUnit.vhd}
analyze -library WORK -format vhdl {RDSelect.vhd}
analyze -library WORK -format vhdl {RdMux.vhd}
analyze -library WORK -format vhdl {cu_hw.vhd}
analyze -library WORK -format vhdl {R1HazardDetection.vhd}
analyze -library WORK -format vhdl {R2HazardDetection.vhd}
analyze -library WORK -format vhdl {hazard_detection.vhd}
analyze -library WORK -format vhdl {latch.vhd}
```

```
analyze -library WORK -format vhdl {latch_generic.vhd}
analyze -library WORK -format vhdl {fd.vhd}
analyze -library WORK -format vhdl {register_generic.vhd}
analyze -library WORK -format vhdl {registerfile.vhd}
analyze -library WORK -format vhdl {sign_extention_register_generic.vhd}
analyze -library WORK -format vhdl {fetch_stage.vhd}
analyze -library WORK -format vhdl {decode_stage.vhd}
analyze -library WORK -format vhdl {execution_stage.vhd}
analyze -library WORK -format vhdl {memory_stage.vhd}
analyze -library WORK -format vhdl {write_back_stage.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.vhd}
analyze -library WORK -format vhdl {a-DLX.vhd}

link
################################################################
# elaborating the top entity
# choose the architecture you want
elaborate DLX -architecture dlx_rtl
################################################################
link

# Define a new variable Period force the value you want
set Period 3
#Forces a clock of period Period connected to the input port CLK #
create_clock -name "clk" -period $Period {"clk"}
#forces a combinational max delay of Period ns from each of the inputs
#to each of th output in case combinational paths are present
set_max_delay $Period -from [all_inputs] -to [all_outputs]

#compile the design
compile -map_effort high
# save report
report_timing > reports/DLX/DLX_3ns_timing.txt
report_area > reports/DLX/DLX_3ns_area.txt
report_power > reports/DLX/DLX_3ns_power.txt

# Define a new variable Period force the value you want
set Period 5
#Forces a clock of period Period connected to the input port CLK #
create_clock -name "clk" -period $Period {"clk"}
#forces a combinational max delay of Period ns from each of the inputs
#to each of th output in case combinational paths are present
set_max_delay $Period -from [all_inputs] -to [all_outputs]

#compile the design
compile -map_effort high
# save report
report_timing > reports/DLX/DLX_5ns_timing.txt
report_area > reports/DLX/DLX_5ns_area.txt
```

```
report_power > reports/DLX/DLX_5ns_power.txt

# Define a new variable Period force the value you want
set Period 62.5
#Forces a clock of period Period connected to the input port CLK #
create_clock -name "clk" -period $Period {"clk"}
#forces a combinational max delay of Period ns from each of the inputs
#to each of th output in case combinational paths are present
set_max_delay $Period -from [all_inputs] -to [all_outputs]
set_max_dynamic_power 1 mW

#compile the design
compile -map_effort high
# save report
report_timing > reports/DLX/DLX_62,5ns_timing.txt
report_area > reports/DLX/DLX_62,5ns_area.txt
report_power > reports/DLX/DLX_62,5ns_power.txt

# Define a new variable Period force the value you want
set Period 31.25
#Forces a clock of period Period connected to the input port CLK #
create_clock -name "clk" -period $Period {"clk"}
#forces a combinational max delay of Period ns from each of the inputs
#to each of th output in case combinational paths are present
set_max_delay $Period -from [all_inputs] -to [all_outputs]
set_max_dynamic_power 700 uW

#compile the design
compile -map_effort high
# save report
report_timing > reports/DLX/DLX_31,25ns_timing.txt
report_area > reports/DLX/DLX_31,25ns_area.txt
report_power > reports/DLX/DLX_31,25ns_power.txt


# saving files
write -hierarchy -format ddc -output DLX-structural-topt.ddc
write -hierarchy -format vhdl -output DLX-structural-topt.vhdl
write -hierarchy -format verilog -output DLX-structural-topt.v
```

4.2 Results comparison

Here is a table highlighting the power, timing and area parameters for the four implementations.

	timing	power	area
3 ns	slack violated	3.7006e+03 uW	17947.552062
5 ns	0.44 ns	2.3085e+03 uW	15047.088159
62.5 ns	30.92 ns	533.3143 uW	16234.778248
31.25 ns	15.291 ns	663.3711 uW	16201.528247

For the high speed implementations we can see that the 3 ns clock period parameter was too strict of a constraint which led the implementation to violate the slack. For this reason the clock period parameter was increased to 5 ns and this new high speed implementation (clock frequency is 200 MHz) present a power consumption of 2.3085 mW.

For the low power implementations two of the most popular clock for low power microprocessors ,namely 16 MHz and 32 MHz, were taken into account. The first low power implementation a clock frequency constraint of 16 MHz resulted in a positive slack of 30.92 ns and a power consumption of 533.3143 uW. The indication of such a positive slack gave away the fact that the design space could be explored for a new implementation that presented a faster clock frequency with a marginal increment in power consumption; the clock frequency chosen for this second low power implementation is 32 MHz which gave me a power consumption of 663.3711 uW. Relative to the first low power implementation a doubling of clock frequency increased the power consumption of roughly 20%.

4.2.1 Synthesis reports

To see the complete result reports refer to the files in the folder ../reports/DLX

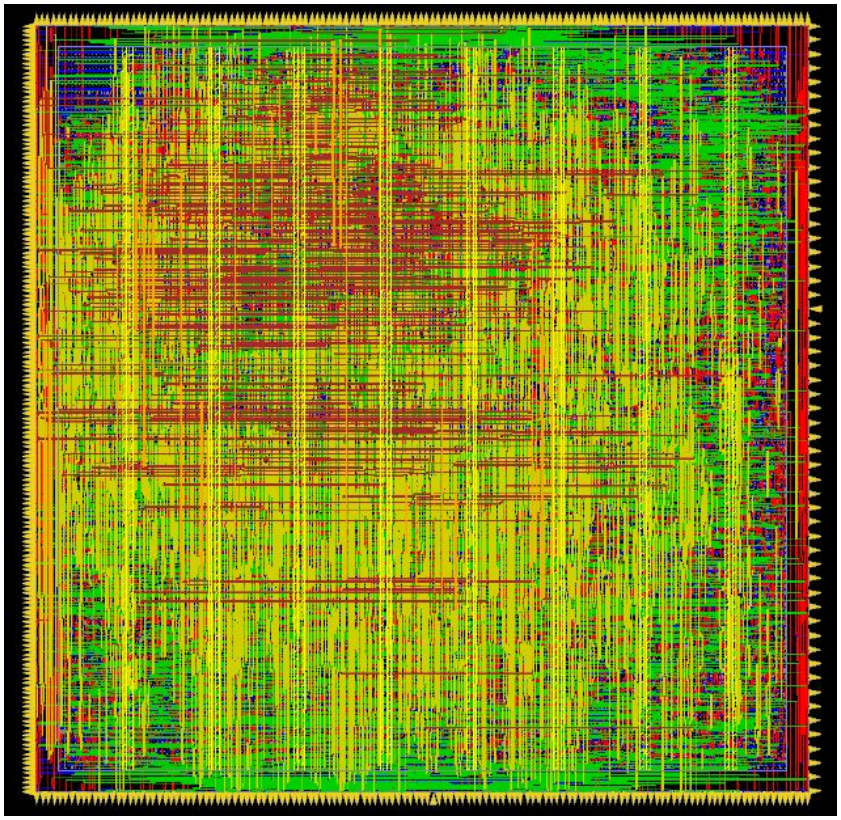
CHAPTER 5

DLX placement and routing

5.1 Place and Route goal

The goal of the place and route part was to take the post synthesis design in verilog of our design (in my case was the 32 MHz implementation) and placing and routing all the cells that are part of the design itself. Furthermore both timing reports, parasitics extraction and design analysis and verification were performed. For the complete reports refer to the folder ../placeandroute.

The final product of the place and route step is the following.



APPENDIX A

Further improvements

In this section will be presented tips on how to improve the current design.

A.1 Expansion of instruction set

Currently this DLX design can perform the following instructions

- SLL
- SRL
- ADD
- SUB
- AND
- OR
- XOR
- SEQ
- SNE
- SLT
- SGT
- SLE
- SGE
- J
- JAL
- BEQZ
- BNEZ
- JR

- JALR
- NOP
- SW
- LW

A first improvement of my DLX design could be to expand the instruction set; this could be done by expanding the input FUNC of the ALU in order to have access to 16 more possible operations. These operations could be

- SRA
- MULT
- etc...

Another way of expanding the instruction set would be to modify the DRAM in order to be able to store and load not only words but also half-words and bytes. This change would result in adding control signals coming from the control unit to the DRAM in order to distinguish between a LW/SW to a LB/SB etc...

A further improvement of the instruction set would be to be able to have BHI and BLO (branch if higher and branch if lower) available. This improvement has been taken into consideration, in fact the branch prediction unit (refer to chapter2) has an additional input (unused for now) tied to the second output of the register file. This improvement would just require to implement the function inside the branch prediction unit top entity and modify the .asm to .mem script to accomodate these two J-type instructions.

A.2 Improvement of the hazard detection unit

As explained in the second chapter the HDU (hazard detection unit) is in charge of dealing with RAW stalls for both I-type and R-type instructions. Another improvement that could be achieved for this DLX design is to expand the HDU functionality with respect to LOAD instructions.
In the current design after a LOAD instruction there has to be two clock cycles where there can't be a RAW stall, penalty obtaining the wrong result. Ideally after having fetched the value from the DRAM to be placed into the register file it is possible to feed that to the arithmetic logic unit. This upgrade require the insertion of two more multiplexers in the execution unit and modifying the HDU to handle the LOAD instruction. This upgrade would resolve a RAW stall between two non consecutive instructions. Due to the nature of the pipeline a RAW stall of two consecutive instructions involving a LOAD can not be solved if not by stalling the pipeline.