Microelectronic Systems

DLX PRO

Academic Year 2020-2021

Professors:	Graziano Mariagrazia	
	Turvani Giovanna	
	Coluccio Andrea	
	Riente Fabrizio	
	Vacca Marco	
	Marchesin Andrea	
Students:	Pautasso Gianluca	282819
	Lanza Mauro	290766

Indice

1 Introduction **3**
1.1 Aim and goal of the work . . . 3
1.2 Brief description . . . 3

2 Control Unit **5**
2.1 Operation analysis . . . 5
2.2 Control Signals . . . 6

3 DataPath **7**
3.1 The Structure . . . 7
3.2 FETCH . . . 7
3.3 DECODE . . . 7
3.4 EXECUTE . . . 8
3.5 MEMORY . . . 8
3.6 WRITEBACK . . . 9
3.7 Chain of RD and RS2 registers . . . 9

4 ALU **9**
4.1 Operation analysis . . . 9
4.2 Shifter . . . 10
4.3 Logic . . . 10
4.4 P4 Adder . . . 11
4.5 Booth Multiplier . . . 11

5 Branch and Jump **12**
5.1 Main explanation . . . 12
5.2 Branch . . . 12
5.3 Jump . . . 12

6 Synthesis and Analysis **13**
6.1 Overview . . . 13
6.2 Power and Area Analysis . . . 14
6.3 Timing Analysis . . . 15

7 Physical Design **15**

1 Introduction

1.1 Aim and goal of the work

The aim of this project is to design a real version of the famous DLX processor, which was introduced in the 1994 by John L. Hennessy and David A. Patterson. The DLX is a RISC processor which implements a pipelined arhitecture. The latter is composed of 5 stages:

- Fetch: the instruction is taken from the Instruction Register
- Decode: the instruction is decoded and the value needed are taken from the registers
- Execute: the instruction is executed
- Memory: if we need to access the data memory, it is done in this stage
- Writeback: the results of the instructions are written in the register file

The DLX's instructions can be broken down into three types:

- R-type: pure registers instructions, with three register references contained in the 32-bit word
- I-type: specify two registers, and use 16 bits to hold an immediate value
- J-type: jumps containing a 26-bit address inside the 32-bit word

1.2 Brief description

We realized a complete version of the DLX and we added some commands and features to improve it. Our processor can execute the following instructions:

- add
- addi
- and
- andi
- beqz
- bnez
- jal
- lw
- nop
- or

- ori
- sge
- sgei
- sle
- slei
- sll
- slli
- sne
- snei
- srl
- srli
- sub
- subi
- sw
- xor
- xori

Moreover it can execute the following pro instructions:

- sra
- srai
- mult
- multi
- addu
- addui
- subu
- subui
- multu
- jalr
- sgeu

- sgeui
- seq
- seqi
- sgt
- sgti
- sgtu
- sgtui
- slt
- slti
- sltu
- sltui

The execution is pipelined according to the requirements and, thanks to a dedicated unit, it can manage the data dependencies.

2 Control Unit

2.1 Operation analysis

The control unit is in charge of handling the signals of the entire system. We designed an Hardwired Control Unit which manages 19 signals (Control Words). It takes the control word used for the signals from a microcode memory which contains all the possible combinations. The CU receives as input the code of the instruction to be executed. The input is 32-bit long and the bits from 31 to 26 are named OPcode, while the bits from 10 to 0 are named FUNC. These two groups of bits are implemented, by the CU, in order to determine which is the operation to be executed. Firstly, the OPcode is being analyzed to distinguish which kind of operation is (R-type,I-type or J-type). If the integer value of the OPcode is greater than 0, it is a R-type operation and the value indicates the exact instruction to be executed by the ALU as well. On the other hand, if the OPcode is equal to 0, the instruction to be executed is an I-type or J-type. In this case the CU needs to analyze the FUNC to discern between the list of the available operations. Once the operation in input has been identified, the CU sends to the ALU the operation to be carried out.

2.2 Control Signals

The Control Word is 19 bits long and it contains the value of the all signals to be forwarded to the data path. A brief description, which wants to clarify what are the main signals, is proposed below.

FETCH STAGE
In the Fetch stage the signals handle PC, NPC and the registers which are in charge of keeping the data of the instruction word.

- HW_PC_enable_fetch: enables the PC register;
- HW_NPC_enable_fetch: enables the NPC;
- HW_pre_immediate_enable: enables the set of registers present in this stage;

DECODE STAGE
In the Decode stage the signals mainly handle the Register File which needs to be enabled and set in read mode to obtain the data.

- HW_Jump_25: activates the jump instruction;
- HW_reg1_decode: enables the set of registers present in this stage;
- HW_RF_enable: enables the Register File;
- HW_RF_RS1: enables the output RS1 of the RF;
- HW_RF_RS2: enables the output RS2 of the RF;
- HW_RF_WR_RDneg: sets the write/read mode of the RF;

EXECUTE STAGE
In the execute stage the signals set what are the the necessary input for the ALU. They also ensure the proper storage of the ALU's output.

- HW_muxs1: configures the output of the MUXS1;
- HW_muxs2: configures the output of the MUXS2;
- HW_reg2_execute: enables the set of registers present in this stage;

MEMORY STAGE
In the memory stage the signals define the right settings for the memory depending on the instruction is being executed.

- HW_reg3_mem: enables the set of registers present in this stage;
- HW_CS_mem: enables the memory;
- HW_RD_WRN_mem: sets the write/read mode of the memory;
- HW_MUX_PC_mem: enables the PC in case of jump or branch;

WRITEBACK STAGE
In the WriteBack stage the signals set the address and the data which are going to be written in the Register File after being processed.

- HW_muxs3: configures the output of the MUXS3
- HW_MUX_RS2_WR_sel: configures the output of MUXRS2;
- HW_MUX_RD_WR_sel: configures the output of MUXWR;

3 DataPath

3.1 The Structure

The datapath is the whole set of hardware components which read, elaborate and store the data. During the design phase, we decided to keep it divided into different stages. This solution gives advantages in terms of clarity and readability. Moreover, it is easier to make an adjustment and to debug the single parts as well.

3.2 FETCH

In the Fetch stage the instruction to be carried out is extracted from the IRAM and it is decomposed and stored in dedicated registers.

- **PC:** It is a 32-bits register and it contains the address to be inserted in the IRAM. It is synchronous and it has three input signals: CLK,RESET,ENABLE. It takes as input a std_logic of 32 bits which will be available as output after one clock cycle.
- **NPC:** It is a hardware component which takes as input the output of PC and computes what will be the next instruction's number by adding 4 to the input. It has three input signals: CLK, RESET, ENABLE.
- **IR:** It is the IRAM of the processor, it contains the list of instructions belonging to the main program. It is a memory which is filled at the beginning of the simulation with the instructions present in the .asm file. It takes as input an address and outputs the matching instruction.

3.3 DECODE

In the DECODE stage the data stored in the Register File are picked up and stored in the dedicated registers. The "immediate" data are processed to obtain the correct value for the computation.

- **Register File:** It is one of the most important element of the processor. It has two mode : READ and WRITE. In the Read mode the RF outputs 2 data depending on what are the two addresses used to make the access. Instead, in WRITE mode, it writes in the requested addresses a data given to it as an

input. To summarize it has 3 input addresses, 2 for the read and 1 for the write. It has also 1 32-bits input and 2 32-bits outputs. The signals used to control it are

  - RF_enable: enables the RF;
  - RF_RS1: enables the first output;
  - RF_RS2: enables the second output;
  - RF_RD/WR: enables the mode.

  It is synchronous in both modes.

- **MUXJUMP25:** It is a multiplexer which, according to the signal provided by the CU, select whether to use the immediate 26-bits long or the immediate 16-bits long.
- **Register NPC:** It is a 32-bits register which contains the output of the NPC.
- **Register A:** It is a 32-bits register which contains the first output of the Register File.
- **Register B:** It is a 32-bits register which contains the second output of the Register File.
- **Register IMM:** It is a 32-bits register which contains the correct value of the Immmediate, since depending on the instruction it could be of different.

3.4 EXECUTE

In the EXECUTE stage the data are processed by the ALU and they are stored in a dedicated register.

- **MUXS1 and MUXS2:** This two multiplexers select the entries of the ALU according to the signals provided by the CU.
- **ALU:** It is the core of the processor, here are done the all arithmetic computation and much more. For a detailed description see the dedicated chapter.

3.5 MEMORY

In this stage the data which need to be written in the memory are stored in it. Here there is also the multiplexer in charge of selecting what will be the next input of the PC.

- **Memory:** Its name says it all. It can be accessed using an address and it is possible to read data from it or to store data in it. It has has 2 signals:

  - CS_mem: enables the memory;
  - RD_WRN_mem: selects the mode to run the memory with

Its dimension is 32 but can be easily extended. It is synchronous either in read or write mode.

- **MUXPC_OUT:** This multiplexer is responsible for selecting what is the signal that will be written in the PC. Indeed, there are some instructions which change the value of the PC, for instance the JUMP instructions. In our design the ALU computes the new possible value of the branch/jump and in case of a branch it determines if the branch is taken or not as well. Depending on this output, the PC will be written with the value stored in the NPC or with the value just computed by the ALU. Therefore, the aim of the MUXPC is to enable this mechanism when these kind of instructions are being executed.

3.6 WRITEBACK

In the WRIT-BACK the data and their respective addresses are selected. The design involves that in this stage the data can be from the memory or from the ALU. Likewise, the address indicating where the data is going to be stored, is selected.

- **MUXS3** It is the multiplexer which is responsible for choosing whether the data needed is from the memory or from the ALU.
- **MUX_RS2_WR** This multiplexer, in case of the instruction JAL is being executed, outputs the address corresponding to the register 32.
- **MUX_RD_WR** This multiplexer select which is the correct address to be used in the write operation. Indeed, depending on the type of the instruction, the writing address is in a different position of the instruction word. In this phase the right one is selected.

3.7 Chain of RD and RS2 registers

In the datapath there is a chain made of 2 32-bits register for each stage. It is necessary to propagate the writing address, which is read in the Fetch stage, until the WriteBack stage. The chain is composed of the registers RS2 and RD, they contain, respectively, the writing address of I-type instruction or R-type one. They are propagated through all the datapath until the WriteBack stage where is decided which is necessary to complete the operation.

4 ALU

4.1 Operation analysis

The Arithmetic Logic Unit is the part of the processor that is in charge of executing all arithmetic and logic operations. The core of the ALU is composed by four components: Shifter, Logic, P4 Adder, Booth Multiplier. The ALU is also in charge of calculating the branches and to output the signal that enables the right multiplexers and registers that executes the Branch/Jump operations. The ALU

receives the data to be used to compute the result and the name of the operation to be executed.

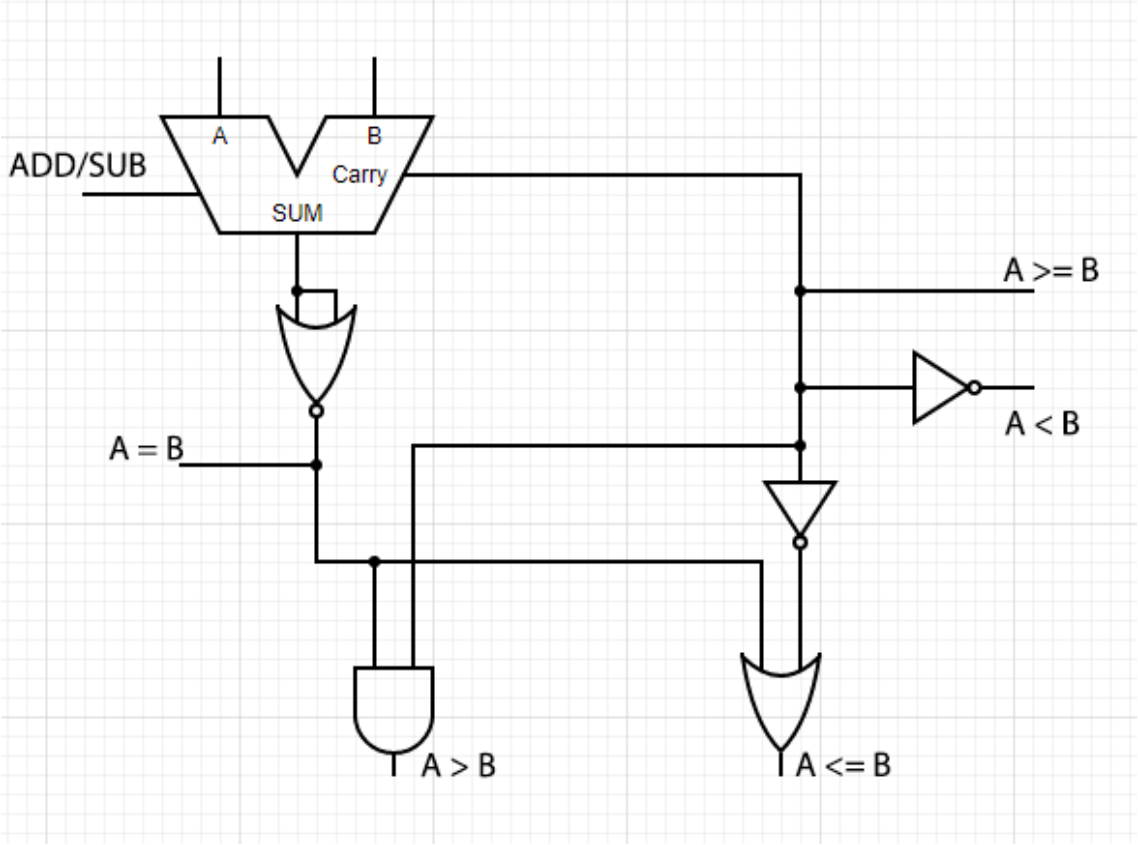
4.2 Shifter

The Shifter receives as input a 2-bit signal that tells it what kind of shift needs to be executed:

- 00: Shift Left
- 01: Shift Right
- others: Shift Right Arithmetic

4.3 Logic

The Logic components is in charge of understanding the relationship between the two data A and B given to it as input.



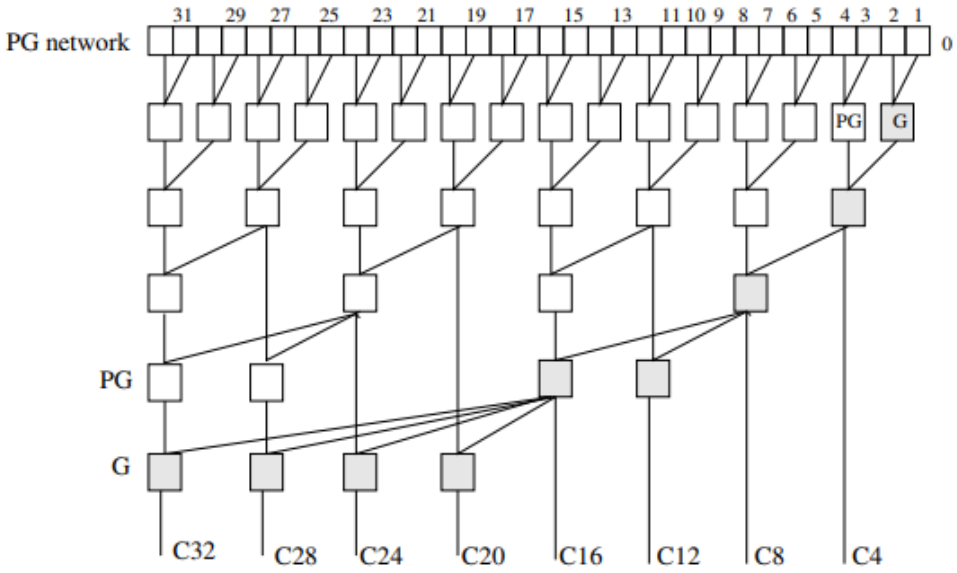
To get a result we first perform a sum/subtraction of the two input data and we then confront it with itself and with the carry it generated. The function described in the figure above are the following ones:

- A = B: we perform an NOR operation between all the bits of the result, if all the bits are 0 then the result of the NOR will be one, hence our signal that tells us whether A it's equal B goes to 1

- $A > B$: we take the result of the NOR operation cited above and we put it in an AND operator with the carry
- $A \leq B$: we do the same operation of A B but with an OR operator and with a negated carry
- $A < B$: to understand this we just need to check if the carry is 0
- $A \geq B$: we do the opposite of the A B operation

4.4 P4 Adder

Our P4 Adder has a Sum Generator composed by 8 blocks with 4 bit per block. The Carry Generator is a Sparse tree that can be adjusted to compute the carry of numbers that have a dimension of bit that is a power of 2 up to 64 bits. We use it to 32 bits, that's why the 8 blocks of 4 bit per block of the Sum Generator. In order to generate the carries our component analyses at what row of the sparse tree it is and depending of the input number of bits it generates dynamically the results. In the following image a representation of a sparse tree for a 32 bit adder

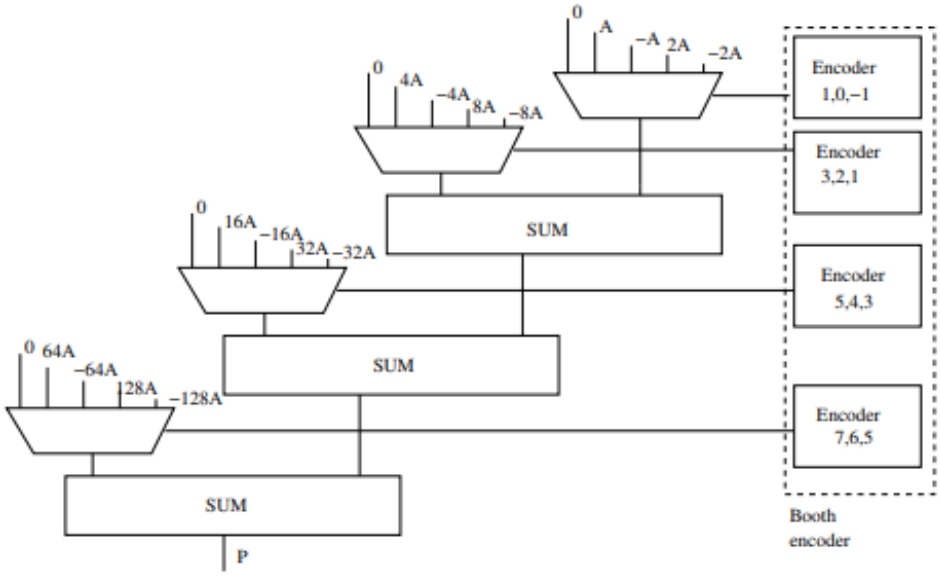


4.5 Booth Multiplier

The Booth Multiplier works with a series of Multiplexer that chooses one of the 2 inputs of the P4 Adder cited above. The select signal of the multiplexer is given by encoders that chooses one between five possible output of the adders. For a 32-bit numbers we will need a total of:

- Encoder and Multiplexer = $\log_2 32Bit = 4$
- Adders = $\log_2 32Bit$ - 1 = 3

Each multiplexer always has five inputs: one of those is always zero, two are a power of 2 of one of the numbers that we want to multiply and the last two are the negated version of the previous one. The second number that we want to multiply is used in the encoders to choose the output of the multiplexer. In the following figure a representation of the Booth multiplier



5 Branch and Jump

5.1 Main explanation

The branch and the jump are a kind of operations which modify the PC value, in order to change the order of the execution flow. They are fundamental in each program and they are needed to create loop and other constructs. In our architecture, after they have been fetched and identified, they are performed in the ALU. It is in charge to compute the next value of the PC.

5.2 Branch

In case of a Branch instruction, the ALU compute the value of the next PC instruction and, at the same time, it computes the logical comparison to understand if the branch is taken or not. It outputs a signal which will be used in the next stage to select which will be the correct input of the PC.

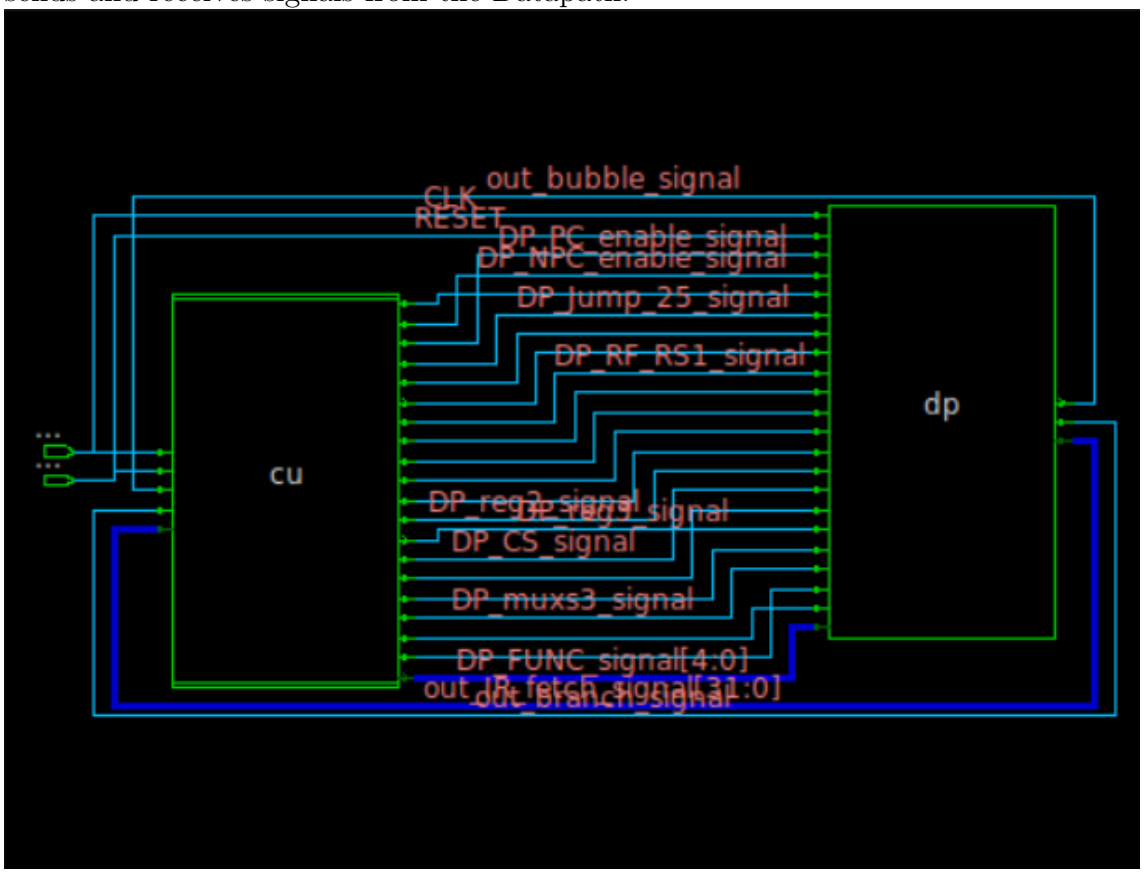
5.3 Jump

In case of a Jump instruction, the ALU compute the value of the next PC instruction and enable the multiplexer in charge of selecting the right input of the PC.

6 Synthesis and Analysis

6.1 Overview

The synthesised circuit appears to be like what we expected: we have the CU that sends and receives signals from the Datapath.



As we can see from the picture, all the control signals of the *Control Word* are from the CU to the DP and are recognizable by a *DP_* sign. The output of the Datapath are just 3:

- out_bubble_signal - Used to signal to the CU the presence of a STALL from the Hazard Unit
- out_IR_fetch_signal - Used to pass to the CU the value of the next instruction in order to generate the corresponding Control Word
- out_branch_signal - Used to signal to the CU the fact that a branch as been taken and so we need to flush the instructions that was being executed and to fetch the new one.

All the commands that have been used can be found in the folder */SYN/commands.tcl*. In that TCL script we, not only, reported all the commands for the clock settings and the required instructions for the synthesis, but also all the report commands. This is very helpful for reading in a clear manner all the Power, Area and Timing analysis results in a *.txt* file. These kind of analysis will be better explored later.

6.2 Power and Area Analysis

In order to do a complete analysis of the circuit we provide an analysis of the power and the area. The commands used to reach this purpose are :

- report_area
- report_power
- report_power -hier
- report_power -net

The value obtained at the beginning are the following:

- total area = 496.356 umm$\hat{2}$
- total power = 129.380 uW

To analyze which are the impact of the Clock frequency on the total power, we set 3 different clock value: 1MHz, 10 MHz, 100 MHz and 1 GHz. The retrieved result are shown below:

CLK	**Power**	**Power -hier**	**Power -net**	**Area**
1 MHz	9.55uW	9.52uW	7.06nW	496.35
10 MHz	13.87uW	13.07uW	71.52nW	496.35
100 MHz	52.20uW	48.241uW	706.27nW	496.35
1 GHz	439.85uW	400.25uW	7.06uW	496.36

Tabella 6.1: **pre optimization**

It is possible to notice the trend of the power, which is proportional to the increasing of the clock frequency.
Then, in order to optimize the circuit, we used the command:
compile -exact_map -map_effort high -area_effort high -power_effort high
and we obtain the following results :

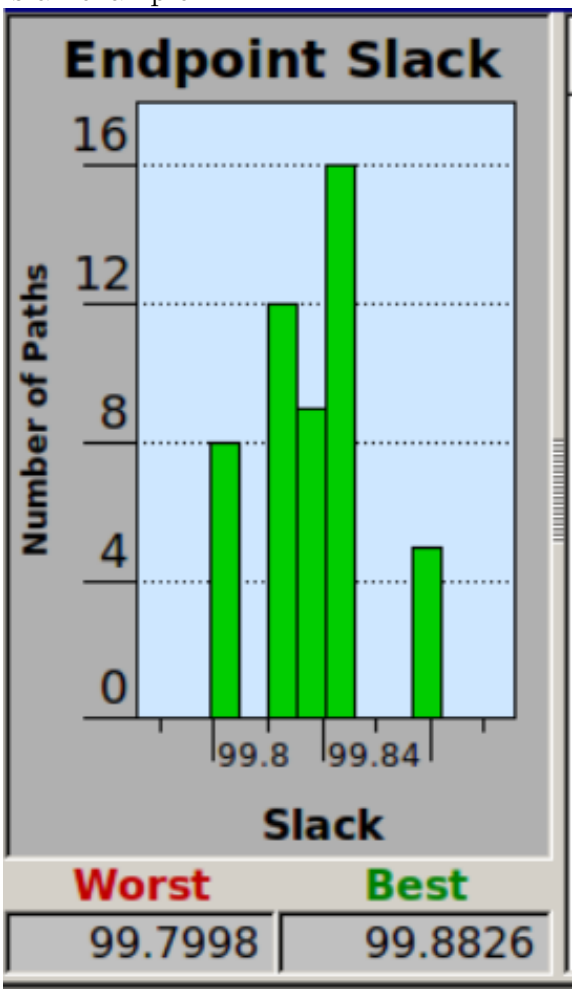
CLK	**Power**	**Power -hier**	**Power -net**	**Area**
1 MHz	28.82uW	9.90uW	8.37nW	1311.34
10 MHz	64.49uW	13.64uW	71.52nW	1309.95
100 MHz	120.05uW	867.68nW	706.27nW	1311.64
1 GHz	969.34uW	408.91uW	8.05uW	1450.49

Tabella 6.2: **after optimization**

It is possible to notice that the optimization doesn't work as we expect. This is due to the constrains which are difficult to comply with. Therefore, as a result, the circuit gets worse in terms of area and power. Each time we use a synthesizer, it tries to modify our circuit in different way. For instance it can add the "Clock Gating " technique, or it can change the order of some input to reduce the switching activity. The main problem is that, trying to achieve a better condition, in particular under specific condition, could bring in a worse condition, exactly as in our case.

6.3 Timing Analysis

To perform a Timing Analysis is necessary to set a clock. To do that we used to following command, the same we have used before:
create_clock -name "CLK" -period N {CLK}
After that, a clock of a specific period has been created. In our case the clock periods used are 1 ns, 10 ns, 100 ns, 1000 ns. Once we created a clock we were able to run the *report_timing* command to see what was the slack of the circuit. We also ran the *report_timing -nworst 10* to check on the 10 worst critical paths. After that we ran the optimization of the circuit and we checked on the Endpoint Slack. Here there is an example.



In the file attached is possible to find the 10 critical path before and after the optimization for the 4 different clock frequency.

7 Physical Design

In order to do the Physical Design of our DLX processor we followed the following steps:

1. Created a *DLX.globals* file in order to load the design inside of Innovus.

2. Added a Floorplan with a Core aspect ratio of 1.0 and an Utilization of 0.6. The Core Margins selected were *Core to die boundary* and they've been forced to 5μm.

3. Associated VDD and GND as the nets to power and ground, the ring type was *Around core boundary* and the metal associated with the top lines was M9 and for the bottom lines was M10

4. Added stripes with a *Set-to-set-distances pattern* of 20

5. Placed horizontal wires to prepare VDD and GND for the standard cells by creating a *Special Route* for VDD and GND

6. Created a *Placement Blockage* for all the layers from M1 up to M8 so that the cells would be placed out from the space occupied by the power and ground stripes. This is very important to avoid any congestion problems.

7. Used the *Pin Editor* to set both the CLK and RST to the TOP side of the design

8. Optimized the design both for *Setup* and *Hold* types with a *Post-Clock-Tree-Synthesis* optimization

9. Added a NanoRoute to connect the cells considering the available metal layers

10. Optimized the design both for *Setup* and *Hold* types with a *Post Routing* optimization

The following picture is the result at the end of the Post-Route optimization.