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: group_09
Davide Fogliato, Loris Panaro, Matteo Panigati

July 13, 2022

Contents

1 Summary **1**

2 Introduction **2**

3 Functional schema **3**
3.1 Datapath . . . 3
3.1.1 Fetch unit . . . 4
3.1.2 Decode unit . . . 4
3.1.3 Execution unit . . . 5
3.1.4 Memory Unit . . . 7
3.1.5 Write back unit . . . 8
3.2 Control unit . . . 8
3.2.1 Instruction fetch control signals . . . 9
3.2.2 Instruction decode control signals . . . 9
3.2.3 Execution control signals . . . 9
3.2.4 Memory control signals . . . 9
3.2.5 Write back control signals . . . 10

4 Implementation **11**
4.1 Synthesis results . . . 11
4.1.1 Synthesis without constraints . . . 11
4.1.2 Synthesis with constraints . . . 11
4.1.3 Place and route . . . 12

5 Conclusions **14**

6 References **16**

A ASM programs **17**
A.1 Basic instruction set without jumps test program . . . 17
A.2 Basic instruction set with jumps test program . . . 18

CHAPTER 1

Summary

In this project we refined the DLX processor from RTL level down to the synthesis and physical design.
All the modules in the hierarchy are connected using structural modeling style and the scheme is as follows: a main entity called DLX, which instantiates the control unit and the datapath and connect them together. The datapath in turn instantiates and connects together five components corresponding to the five stages of the pipeline: fetch unit, decode unit, execution unit, memory unit, write back unit. The DLX exploits the basic version functionalities: basic instruction set and basic pipeline with the introduction of stalls to prevent hazards. The synthesis has been performed with Synopsys Design Vision, consisting in various attempts leading to a rise of frequency and a reduction of power consumption.
Finally, the place and routing has been performed using Innovus: we created the floorplan, the power grid and power stripes for the power supply and then placed the standard cells; after that we inserted fillers to ensure N+ and P+ wells continuity among the different layers. In the end, we extracted timing reports to be sure that no paths could violate our constraints and also verification on the geometry and connections to guarantee that our design can be realized and built.

CHAPTER 2

Introduction

The DLX is a conceptual architecture implementing a RISC processor with a 32-bit load/store architecture. There are 32 32-bit wide general purpose registers, from R0 to R31, where R0 always contains the constant value 0 and R31 is used as link register. The arithmetical and logical operations are between 32-bit integer values. The memory is byte addressable with Big Endian mode and 32-bit wide addresses. The basic instruction set consists in a set of 32-bit long instruction split up into 3 groups: I-type (immediate instructions), R-type(registers instructions), J-type(jump instructions). The pipeline consists in five stages: IF (instruction fetch), ID (instruction decode), EX (execution), MEM (memory), WB (write back).

CHAPTER 3

Functional schema

The DLX module is the top entity that instantiates and connect using structural modeling style the two main elements of the processor: the datapath and the control unit. The two flip-flops in 3.1 are needed to activate the control unit two clock cycles after the reset operation, when the first instruction reaches the decode stage.

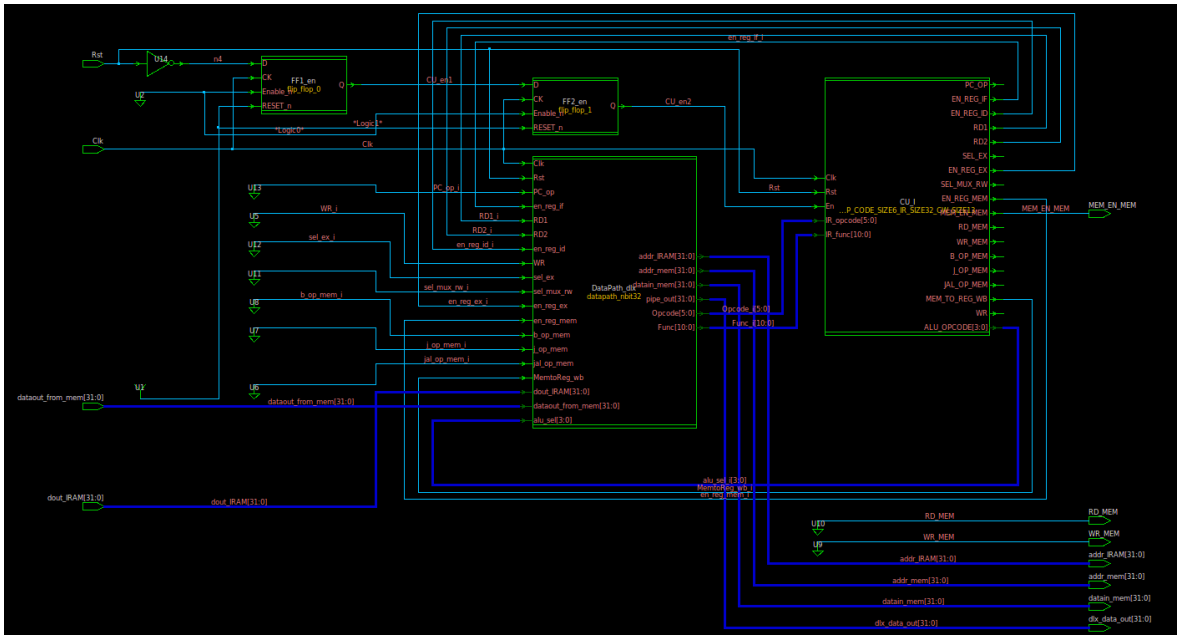

Figure 3.1: Schematic symbol of the DLX.

3.1 Datapath

The datapath module is composed of five sub module connected using structural modeling style, corresponding to each stage of the pipeline; this choice has been made in order to guarantee modularity and easier debugging phases of the project.

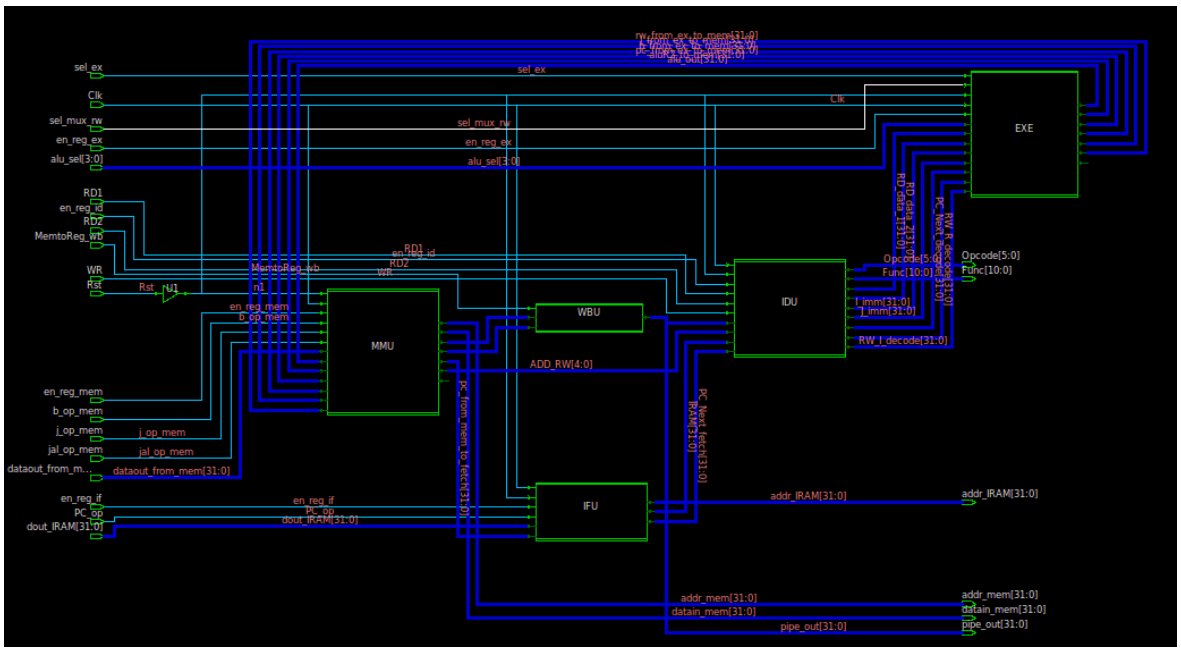

Figure 3.2: Schematic symbol of the datapath.

3.1.1 Fetch unit

The fetch unit is composed of:

- **PC_block:** program counter, a register containing the address pointing to the next instruction to execute in the instruction memory.
- **PC_ADD:** adder performing the addition of 4 to the current program counter in order to compute the next program counter (which is correct in case of no jump instruction). The number required to add is 4 because we operate on a byte addressable memory and we need to move on the next 32-bit long instruction.
- **mux_PC:** multiplexer to select the correct program counter to update at the next clock cycle choosing between the regular program counter generated by the addition of four or a different program counter computed by the execution unit.
- **IF_reg_PC_next:** pipeline register to store the program counter.
- **IF_reg_IRAM:** pipeline register to store the effective instruction (IR).

3.1.2 Decode unit

The decode unit is composed of:

- **registers:** asynchronous register file with 32 32-bit wide general purpose registers (from R0 to R31), two read ports and one write port, active low reset and active low enable.
- **SIGN_EXT_immediate_I:** component for sign extend on 32 bit the immediate (bits 15-0) of an I-type instruction.

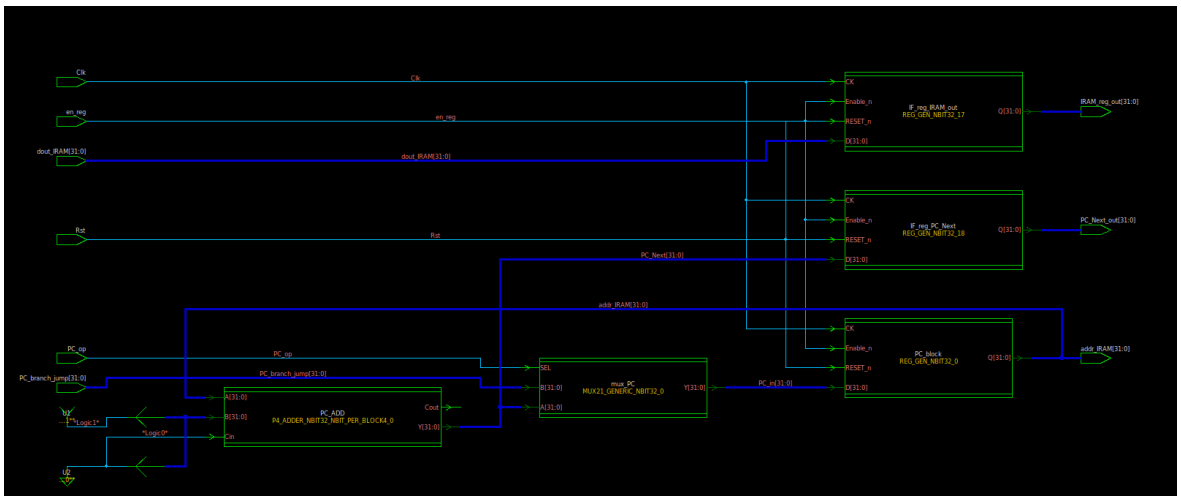

Figure 3.3: Schematic symbol of the instruction fetch unit.

- **SIGN_EXT_immediate_J:** component for sign extend on 32 bit the immediate (bits 25-0) of a J-type instruction.
- **ID_reg_PC_next:** pipeline register to propagate the program counter.
- **ID_reg_data_1:** pipeline register to store the output of the first register read by an instruction.
- **ID_reg_data_2:** pipeline register to store the output of the second register read by an instruction.
- **ID_reg_reg_I_imm:** pipeline register to store the sign-extended immediate value of I-type instruction.
- **ID_reg_reg_J_imm:** pipeline register to store the sign-extended immediate value of J-type instruction.
- **ID_reg_RW_I:** pipeline register to store the write register in case of I-type instruction (bits 20-16).
- **ID_reg_RW_R:** pipeline register to store the write register in case of R-type instruction (bits 15-11).

3.1.3 Execution unit

The execution unit is composed of:

- **alu_dut**: ALU performing the following operations: logical AND, OR, XOR, addition and subtraction using P4 adder where the subtraction is performed computing two's complement of the second operand through a logical NOT on the second operand and setting the external carry in to '1'. Shift operations are performed by a barrel shifter implemented using behavioral modeling style. Compare operations are implemented through a comparator described in behavioral modeling style.
- **pc_br:** adder to sum the propagated program counter to the immediate value corresponding to the needed offset jump instruction.

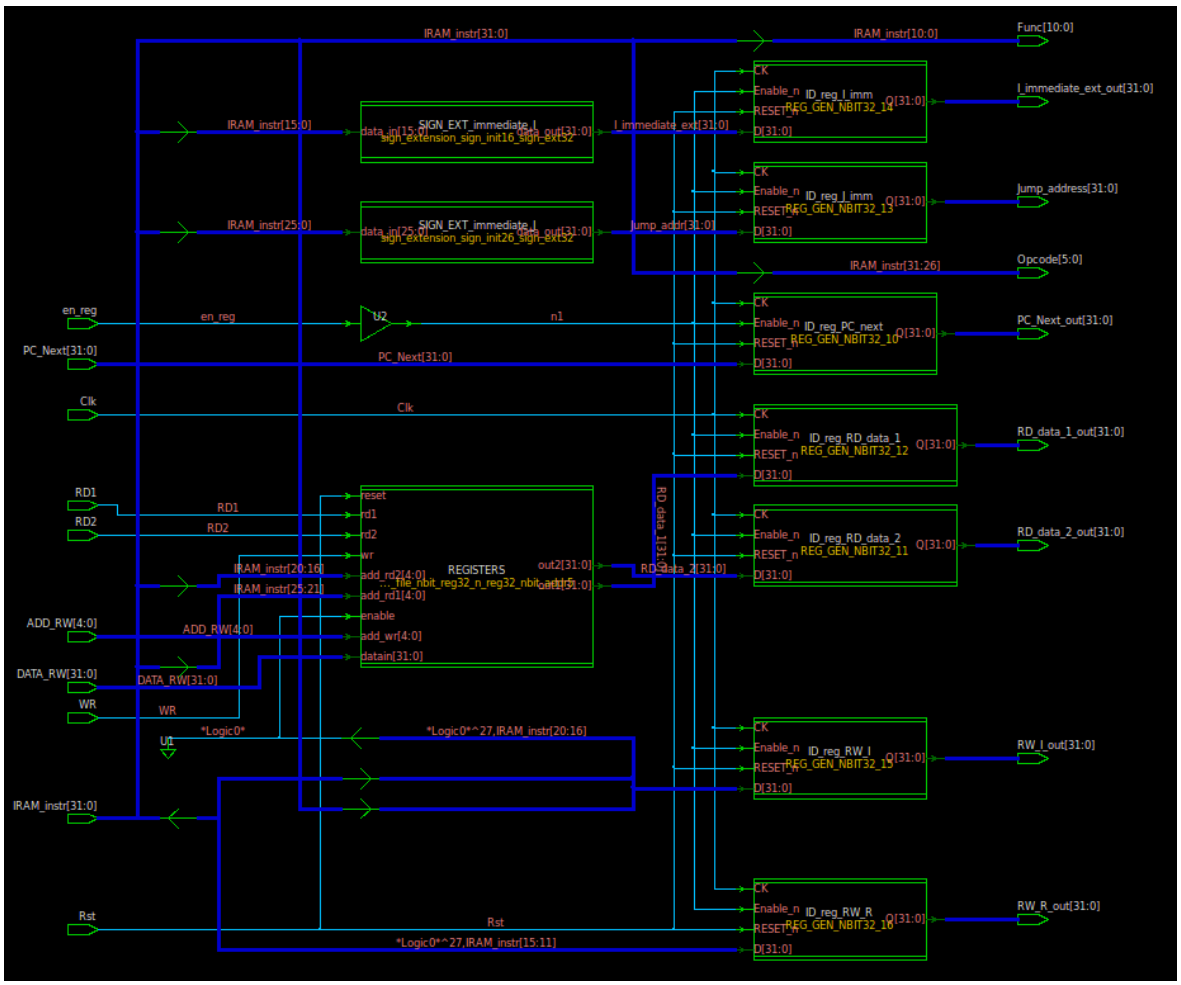

Figure 3.4: Schematic symbol of the instruction decode unit.

- **pc_jp:** adder to sum the propagated program counter to the immediate value corresponding to the offset needed by a branch instruction.
- **lo_mux:** multiplexer to select the second operand for the ALU between the second read register or the immediate value.
- **RW_mux:** multiplexer to select the correct write register in case of R-type or I-type instructions.
- **pc_reg:** pipeline register to propagate the program counter.
- **pc_reg_jp:** pipeline register to store the calculated program counter in case of jump instruction.
- **pc_reg_br:** pipeline register to store the calculated program counter in case of branch instruction.
- **reg_alu:** pipeline register to store the output of the ALU.
- **ls_reg:** pipeline register to store the content of read register 2 containing data to be stored in memory with store instruction.
- **rw_reg:** pipeline register to propagate the write register.

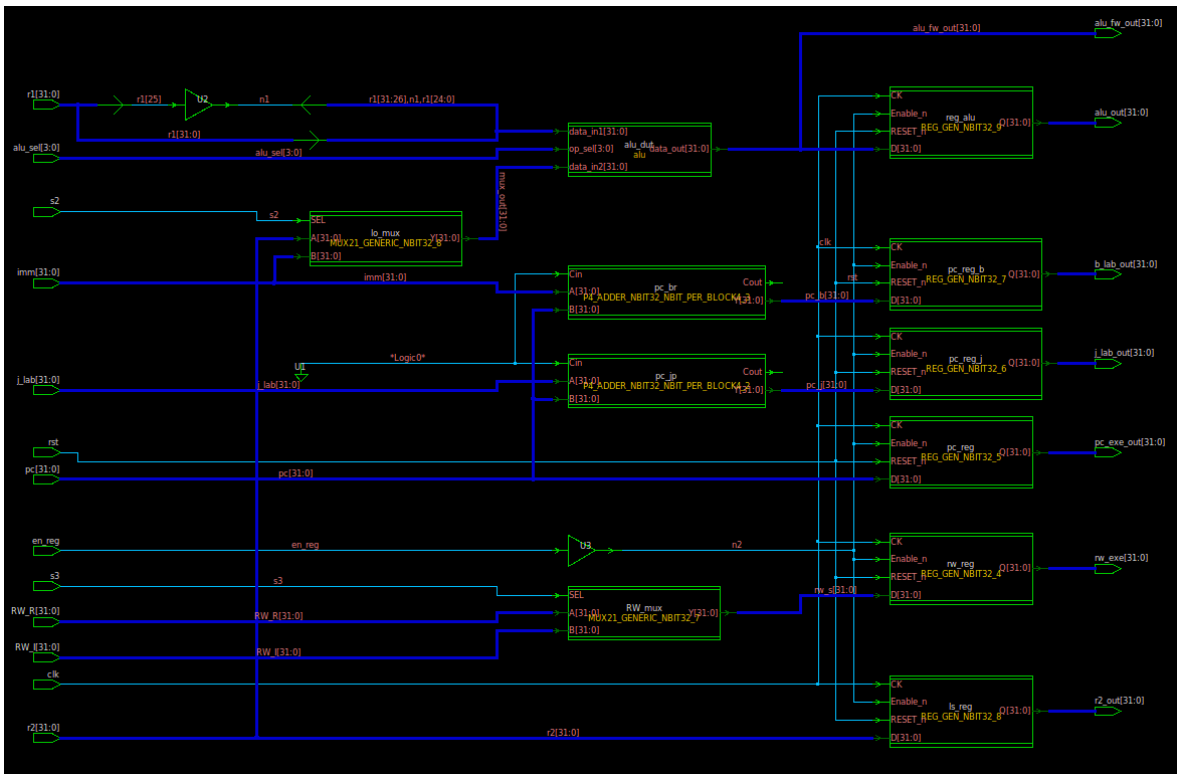

Figure 3.5: Schematic symbol of the execution unit.

3.1.4 Memory Unit

The memory unit is composed of:

- **DataMemory:** an asynchronous memory used only for simulation with active low reset and active low enable with one write port and one read port. The read address is the output of the ALU.
- **branch_block:** component that receives as input the regular program counter, the program counter specified by a branch instruction, B_OP_MEM control signal from the control unit that specifies if a branch instruction is the current one in execution and the output of the ALU comparator; after an AND operation between the control signal and the output of the comparator to check if the instruction is a branch with the valid condition to perform the branch, the result of the AND operations is used as the selection signal for a multiplexer that chooses between the regular program counter and the program counter needed for the branch instruction.
- **jmp_unit:** multiplexer to select between the program counter out of the branch block and the program counter for a jump instruction coming from the execution unit, with J_OP_MEM as selection signal coming from the control unit to select the correct one.
- **mux_jal_pc:** multiplexer with the control signal JAL_OP_MEM as selection signal for selecting between storing in the mem_reg pipeline register the output from a memory read or the value of the program counter to store in the register 31 in case of JAL instruction.
- **mux_jal_r31:** multiplexer with the control signal JAL_OP_MEM as selection signal to select between setting R31 as write register in case of a JAL instruction or the regular propagated write register in case of an instruction different from a JAL.

- **alu_reg:** pipeline register to propagate the ALU output for the successive write back operation.
- **rw_reg:** pipeline register to propagate the write register.
- **mem_reg:** pipeline register to store the memory read output or the program counter value in case of JAL instruction.

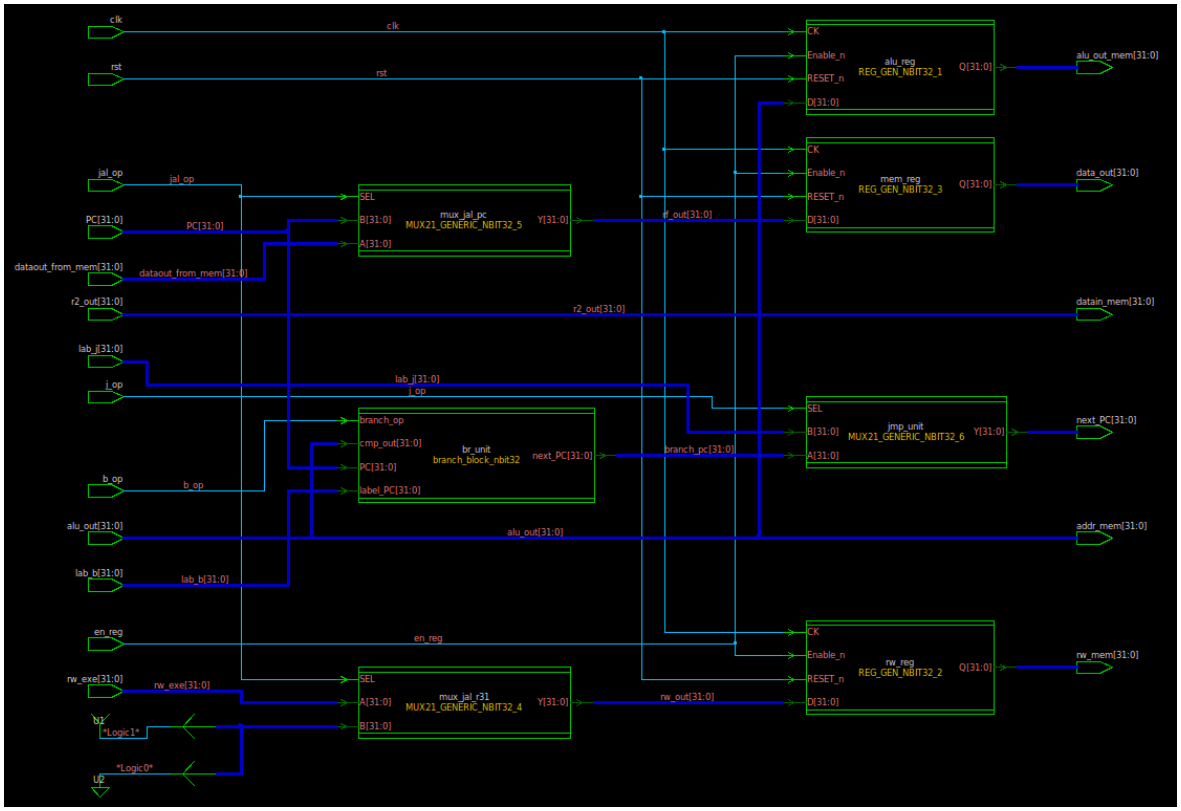

Figure 3.6: Schematic symbol of the memory unit.

3.1.5 Write back unit

The write back unit is composed of:

- **mux_writeBack:** multiplexer to select between the memory read output in case of load operations or the ALU output in case of R-type instruction.

3.2 Control unit

The control unit is modeled according to the hardwired approach. It has an asynchronus active low reset and a look-up table containing the information on how to set the control signal depending on the current instruction. The control unit takes as input the opcode and the **func** from the instruction. The opcode is the address to access the look up table. A process computes the **ALU_OPCODE** depending on both the **opcode** and **func** fields, while another process exploits the pipeline feature granting that each stage receives the correct control signals belonging to different instructions.

The control unit generates these outputs for each of the stages:

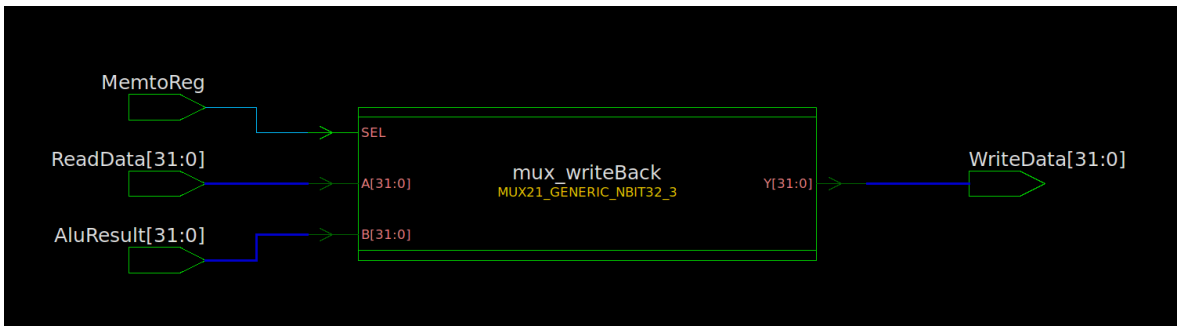

Figure 3.7: Schematic symbol of the write back unit.

3.2.1 Instruction fetch control signals

- **PC_OP:** signal that selects the regular program counter or that one coming from a branch or jump instruction.
- **EN_REG_IF:** signal to enable the pipeline registers.

3.2.2 Instruction decode control signals

- **RD1:** signal to enable read port 1 of the register file.
- **RD2:** signal to enable read port 2 of the register file.
- **EN_REG_ID:** signal to enable the pipeline registers.

3.2.3 Execution control signals

- **SEL_EX:** signal to select between immediate or read data 2 as second operand for the ALU.
- **SEL_MUX_RW:** signal to select the correct write register in case of R-type or I-type instruction.
- **ALU_OPCODE:** signal to select the ALU arithmetical operation.
- **EN_REG_EX:** signal to enable the pipeline registers.

3.2.4 Memory control signals

- **MEM_EN_MEM:** signal to enable all operations on the data memory.
- **RD_MEM:** signal to enable read operation on the data memory.
- **WR_MEM:** signal to enable write operation on the data memory.
- **B_OP_MEM:** signal used to inform the branch block that a branch instruction is in execution.
- **J_OP_MEM:** signal used as selection signal for the jmp_unit to select the program counter computed for a jump instruction.
- **JAL_OP_MEM:** signal used as selection signal for mux_jal_pc and mux_jal_r31 units in case of JAL instruction.
- **EN_REG_MEM:** signal to enable the pipeline registers.

3.2.5 Write back control signals

- **MEM_TO_REG:** signal to select between the memory read output or ALU output for writing into write register.
- **WR:** signal to enable register file write operation.

CHAPTER 4

Implementation

The synthesis of the DLX has been performed by Synopsys Design Vision. The library used is a 45 nm. Instruction memory and data memory has not been included in the synthesis process.

4.1 Synthesis results

4.1.1 Synthesis without constraints

The first synthesis without constraints is here reported:

Timing with no constraints:

- worst path data arrival time = 0.32 ns

Area with no constraints:

- combinational area = 8289.624
- non combinational area = 6348.356206
- total area = 14637.980206

Power with no constraints:

- leakage power = 273.4814 uW
- switching power = 158.5459 uW
- total power = 1.0762 mW

4.1.2 Synthesis with constraints

The second synthesis with constraints reported:

Timing with clock constraint:

- worst path = 1.46 ns
- clock period - library setup time = 1.46 ns

- slack = 0.0 ns

Area with clock constraint:

- combinational area = 9080.974051
- non combinational area = 6333.460204
- total area = 15414.434255

Power with clock constraint:

- leakage power = 3.2207e+05 nW
- switching power = 515.3112 uW
- total power = 4.2007e+03 uW

4.1.3 Place and route

Place and route has been realized with a Core Aspect Ratio = 1.0 and Utilization = 0.6 for the floorplanning and through different successive steps: insertion of power rings and stripes, standard cell power routing, placement, I/O pins placing, Post Clock-Tree-Synthesis optimization (CTS), fillers insertion, routing, post routing optimization and timing analysis.

Power rings: high metal layers had been used to avoid congestion, in particular M9 for horizontal lines and M10 for vertical lines; for power stripes M10 has been reused.

Standard Cell Power Routing: this operation has been adopted in order to place horizontal wires to distribute correctly power wires among standard cells.

Placement: this is the real placement of standard cells of the circuit.

I/O pins placing: using the pin editor of the tool, I/O pins of the DIE are placed along the edges; the spread type is "Along Side" for busses or "From Center" for single wires.

CTS: this is a design optimization in order to achieve the required timing constraints.

Fillers insertion: fillers cells are inserted to ensure wells continuity among the standard cells.

Routing: phase in which the connection of standard cells is done taking into account the available metal layers; this operation is under the Nano Route command on Innovus GUI.

Post routing optimization and timing analysis: in this last step, the design is again optimized and parasitics and delay analysis is performed; during this phase, it was important to verify that all paths could satisfy timing constraints with no negative slacks.

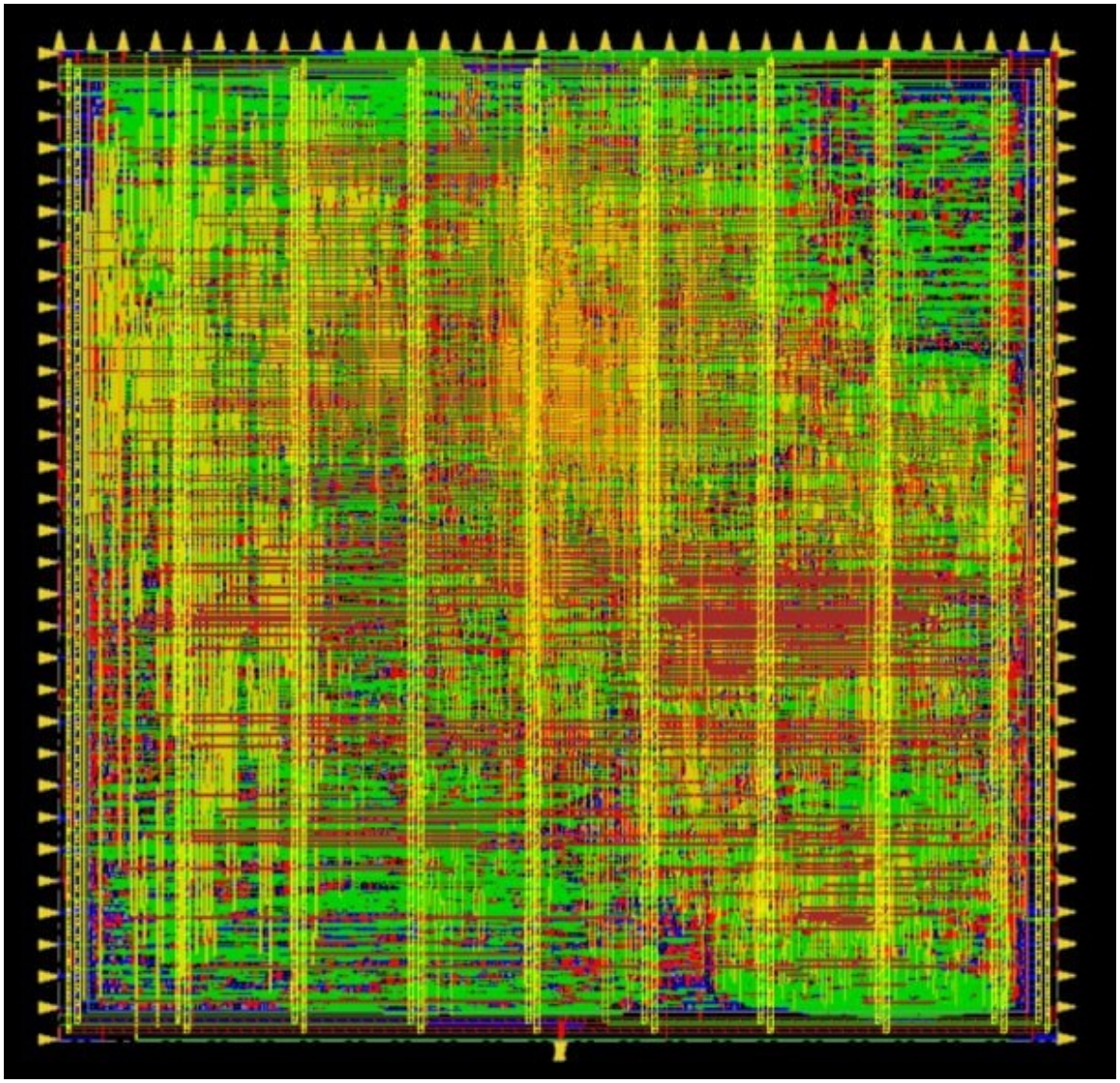

Figure 4.1: Floorplanning of the DLX.

CHAPTER 5

Conclusions

After the first synthesis without constraints we tried to improve DLX performance by applying different clock constraints. We started with a clock period of 2 ns without optimizations on the combinational paths, but pushing to low values we have seen that the synthetizer tries to remove some fundamentals components of the design, like the pipeline registers (the synthetizer does not directly remove them from the design but it is not able to synthetize them); the best solution that we have found is pushing the clock period down to 1.5 ns with no optimizations on the combinational path.
The DLX has been tested with two .asm file. The first assembler program aims to test the correct behaviour of all the instructions of the basic set; each instruction is executed one after the other in a pipelined way, with the introduction of stalls every time needed to avoid hazards. The simulation result is in figure 5.1 and 5.2. The second assembler program aims to test branch and jump instruction, with a specific instruction that must not be executed at all to show if the branch and jumps behaves correctly. The simulation result is in figure 5.3. Both the assembler programs showed correct results. The full code of the two test .asm files is in appendix A.

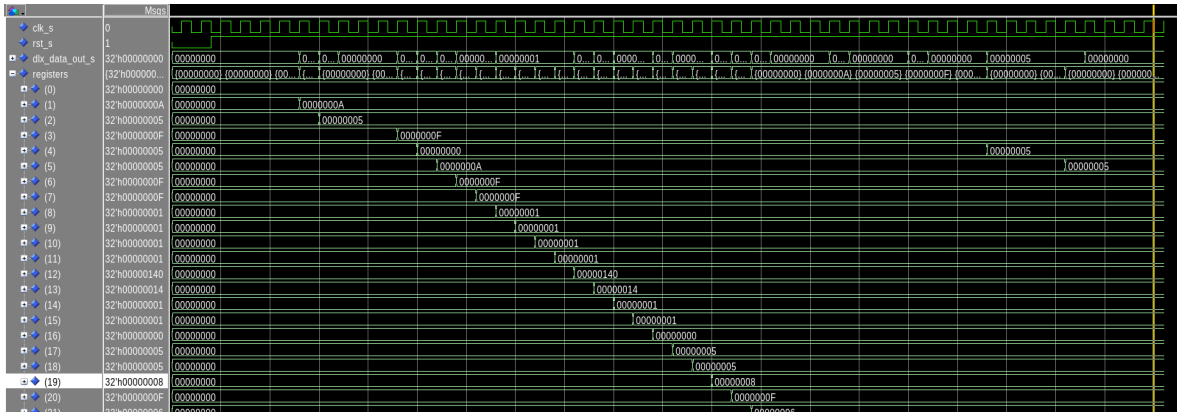

Figure 5.1: View of register file update during basic instruction set execution (jump instruction excluded.)

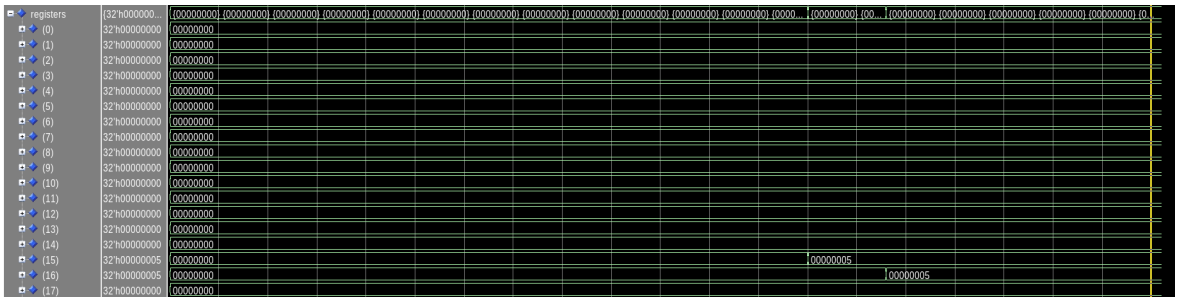

Figure 5.2: View of memory update during store instructions execution.

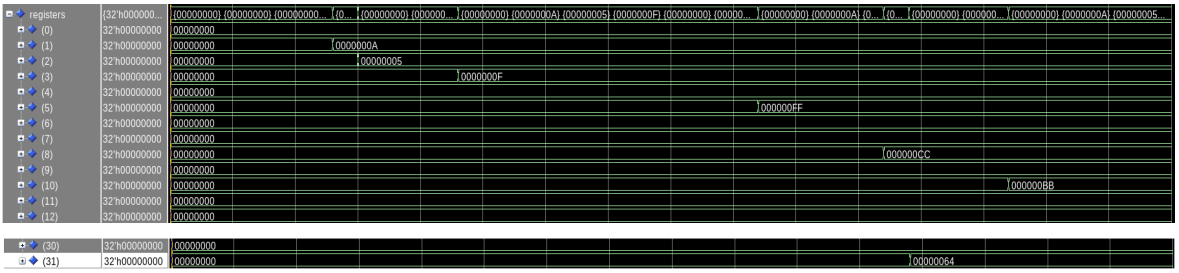

Figure 5.3: View of register file update during jump instruction execution.

CHAPTER 6

References

- Microelectronic Systems lecture notes, Mariagrazia Graziano
- Computer organization and design, David A.Patterson, John L.Hennessy

APPENDIX A

ASM programs

A.1 Basic instruction set without jumps test program

```
addi r1, r1, #10
addi r2, r2, #5
nop
nop
nop
add r3, r1, r2
and r4, r1, r2
andi r5, r1, #0xFFFF
or r6, r1, r2
ori r7, r1, #0x0005
sge r8, r1, r2
sgei r9, r1, #0x0009
sle r10, r2, r1
slei r11, r1, #0x000B
sll r12, r1, r2
slli r13, r1, #0x0001
sne r14, r1, r2
snei r15, r1, #0xFFFF
srl r16, r1, r2
srli r17, r1, #0x0001
sub r18, r1, r2
subi r19, r1, #0x0002
xor r20, r1, r2
xori r21, r1, #0x000C
nop
nop
nop
sw 0(r3), r2
nop
nop
nop
sw 1(r3), r2
```

```
nop
nop
nop
lw r4, 1(r3)
nop
nop
nop
lw r5, 0(r3)
```

A.2 Basic instruction set with jumps test program

```
addi r1, r1, #10
addi r2, r2, #5
nop
nop
nop
add r3, r1, r2
nop
nop
nop
bnez r3, forward
nop
nop
nop
addi r4, r4, #0x0002

again:
addi r5, r5, #0x00FF
beqz r6, step
nop
nop
nop

forward:
j again
nop
nop
nop


step:
addi r8, r8, #0x00CC
jal ending
nop
nop
nop
addi r9, r9, #0x00AA

ending:
```

```
addi r10, r10 #0x00BB

;the program must write "0x00FF" in r5, "0x00CC" in r8 and "0x00BB" in
;r10 and must not write "0x0002" in r4 and "0x00AA" in r9
```