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_38

Alfredo Paolino, Vincenzo Petrolo, Diamante Simone Crescenzo

June 27, 2022

Contents

1 Introduction

In this technical report we will go through the entire process we followed in order to design our DLX microprocessor. The first step was to read up and gather information about the topic. We used as a reference the book "Computer Organization and Design: The Hardware/Software Interface" by Hennessy and Patterson. After that, having in mind what the expected outcome was, we divided the overall workflow in three main phases:

- VHDL design
- Synthesis
- Physical layout

Each of them will be analyzed in depth throughout this report.

2 VHDL Design

2.1 Datapath

The VHDL design started from the datapath, described in a structural style. A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions.

Datapath components

- Program Counter (PC), is a 32-bits register that holds the address of the next instruction the microprocessor will fetch from the Instruction Memory. At each and every clock cycle the PC is incremented by 4, offset needed to move from one instruction to the following one. In case of Jump/Branch instructions, the PC could be modified accordingly to the outcome of the Jump/Branch.
- PC Adder, is a ripple carry adder that increments by 4 the current PC value.
- PC MUX, is a 2:1 MUX that chooses between PC+4 and a modified PC due to Jump/Branch instructions. This selection is driven by a selection signal that will be analyzed later
- Register File (RF), is an array of 32 32-bits registers used to hold currently used data. Having a very simple design here, we use all the registers for the same purpose, except for R0 (hardcoded at logic 0) and R31 (used to maintain the return address in case of jal instructions).The two read ports are constantly active, while the write port is activated by an enable signal when needed
- Addr WR MUX, is a 2:1 MUX that chooses between the portion INSTRUCTION(20:16) and INSTRUCTION(15:11) as the write address for the RF. This selection is driven by a selection signal that will be analyzed later
- JAL R31 Address MUX, is a 2:1 MUX that chooses between the classic write address and the hardcoded address 31 as the write address for the RF. This selection is driven by a selection signal that will be analyzed later
- PC Branch Adder, is again a ripple carry adder used to compute which would be the next address in case of Jump/Branch instructions. Due to the lack of an enable, this address is always computed even when the instruction is not a Jump/Branch. However, the Control Unit will take care to forward the address to the PC only when needed
- Jump MUX, is a 2:1 MUX that chooses between the address needed for an unconditional branch and the address needed for a conditional branch. This selection is driven by a selection signal that will be analyzed later
- ALU, it is the core of the execute step, implementing several logic and arithmetic operations. In particular, the implemented operations are: ADD, SUB, AND, OR, SLL, SRL. Furthermore, ADD and SUB operations are implemented using a P4 adder to increase performance. On top of that, another 1-bit output signal has been inserted, which notifies when the result of an operation is equal to 0. This signal is used in case of beqz and bnez instructions as a sort of control signal for the Jump MUX. The selection of the correct operation is driven by a 4-bit signal that will be analyzed later.
- SE MUX, it is a 2:1 MUX that chooses between the sign extended version of the input immediate and the not sign extended one. This selection is driven by a selection signal that will be analyzed later
- ALU MUX, it is a 2:1 MUX that chooses between the data coming from the register file and the one coming from the SE MUX depending on the type of instruction (R-Type or I-Type). This selection is driven by a selection signal that will be analyzed later

- MEM MUX, is a 2:1 MUX that chooses between the data coming from the ALU and the one coming from the external Data Memory. It is used in case of Load instructions to forward the data from the memory to the RF. This selection is driven by a selection signal that will be analyzed later
- JAL MUX, is a 2:1 MUX that chooses between the data coming from MEM MUX and the address coming from the PC Branch Adder. This is needed in case of jal instruction to swap from saving a data in R31 to saving the return address of the new subroutine. This selection is driven by a selection signal that will be analyzed later

Instruction Set

These are all the instructions we have been able to implement in our DLX microprocessor thanks to the above mentioned datapath:
R-Type

- ADD R1, R2, R3 → R1 = R2 + R3 between signed integers
- SUB R1, R2, R3 → R1 = R2 - R3 between signed integers
- AND R1, R2, R3 → R1 = R2 AND R3
- OR R1, R2, R3 → R1 = R2 OR R3
- XOR R1, R2, R3 → R1 = R2 XOR R3
- SLL R1, R2, R3 → R1 = R2 << R3[27:31] using unsigned integers
- SRL R1, R2, R3 → R1 = R2 >> R3[27:31] using unsigned integers
- SGE R1, R2, R3 → R1 = 1 if (R2 >= R3) else R1 = 0 using signed integers
- SLE R1, R2, R3 → R1 = 1 if (R2 <= R3) else R1 = 0 using signed integers
- SNE R1, R2, R3 → R1 = 1 if (R2 ! = R3) else R1 = 0 using signed integers

I-Type

- ADDI R1, R2, #1 → R1 = R2 + IMM with signed integers. The immediate is on 16 bits
- SUBI R1, R2, #1 → R1 = R2 - IMM with signed integers. The immediate is on 16 bits
- ANDI R1, R2, #1 → R1 = R2 AND IMM. The immediate is on 16 bits, this means that the high 16 bits of R1 will always be set at logic 0
- ORI R1, R2, #1 → R1 = R2 ORI IMM. The immediate is on 16 bits
- XORI R1, R2, #1 → R1 = R2 XORI IMM. The immediate is on 16 bits
- SLLI R1, R2, #1 → R1 = R2 << IMM[27:31] using unsigned integers. The immediate is on 16 bits
- SRLI R1, R2, #1 → R1 = R2 >> IMM[27:31] using unsigned integers. The immediate is on 16 bits
- SGEI R1, R2, #1 → R1 = 1 if (R2 >= IMM) else R1 = 0 using signed integers. The immediate is on 16 bits
- SLEI R1, R2, #1 → R1 = 1 if (R2 <= IMM) else R1 = 0 using signed integers. The immediate is on 16 bits
- SNEI R1, R2, #1 → R1 = 1 if (R2 ! = IMM) else R1 = 0 using signed integers. The immediate is on 16 bits

- BEQZ R1, LABEL → PC = PC + IMM if (R1 == 0). The immediate is on 16 bits
- BNEZ R1, LABEL → PC = PC + IMM if (R1 ! = 0). The immediate is on 16 bits
- LW R1, #0(R2) → R1 = MEM[IMM + R2]. The immediate on 16 bits acts as on offset for the register value used as address
- SW #0(R2), R1 → MEM[IMM + R2] = R1. The immediate on 16 bits acts as on offset for the register value used as address
- NOP. It has been implemented as ADD R0, R0, #0 and simply idles the microprocessor for one cycle

J-Type

- J LABEL → PC = PC + IMM. The immediate is on 26 bits and is shifted left by 2 positions
- JAL LABEL → PC = PC + IMM, R31 = PC + 4. The immediate is on 26 bits and is shifted left by 2 positions

Pipeline

Like any other MIPS architecture, the datapath is organized in a 5 stages pipeline, which increases the overall performance of the entire microprocessor. Each stage is then separated from the following one by means of some pipeline registers. Due to the delay in terms of clock cycles needed for an instruction to complete, we had to insert a skewing network on the control signals too, in order to synchronize them according to the datapath pipeline. With the actual design we decided to implement, we ended up having a branch delay slot of 2 instructions. Due to this limitation, we had to insert some NOP instructions in our testbenches after Jump/Branch instructions in order to guarantee a correct execution of the program. However, by moving the new address computation to the ID stage we could reduce the branch delay slot to a single instruction.
In more details, the 5 pipeline stages are:

- Instruction Fetch cycle (IF), where the microprocessor sends out the PC and fetches the instruction from the Instruction Memory. On top of that, it increments the PC by 4 to address the next sequential instruction.
- Instruction Decode cycle (ID), where the microprocessor decodes the instruction and accesses the Register File to read the registers
- EXecution cycle (EX), where the ALU operates based on the type of instruction received. On top of that, the adder computes the next address in case of Jump/Branch instructions
- MEMory cycle (MEM), where the Data Memory is accessed. If the instruction is a load, the data is read from the memory and written on the next cycle in the RF, meanwhile, if the instruction is a store, the result computed in the ALU is directly written in memory.
- Write Back cycle (WB), where the microprocessor writes back the result into the Register File, whether it comes from the memory system or from ALU.

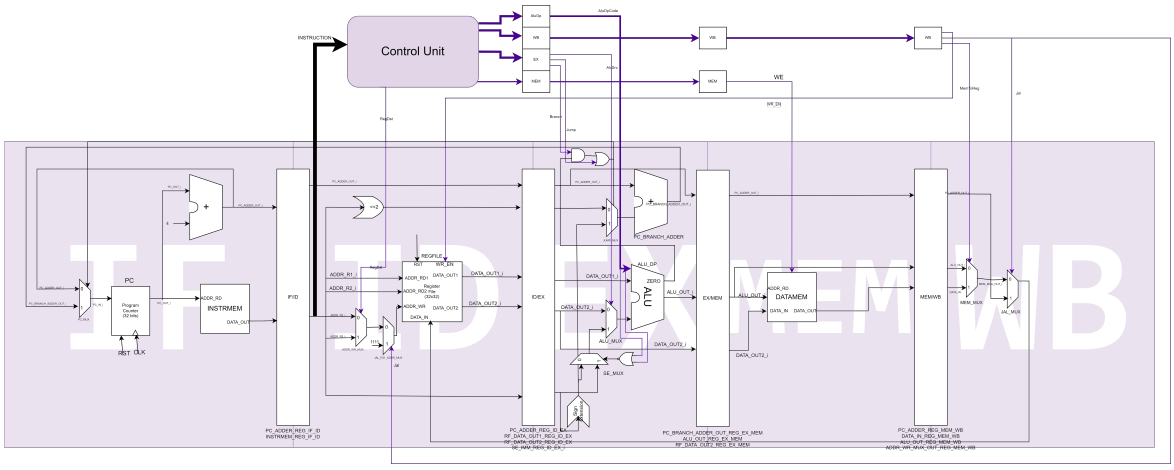

Figure 1: DLX design schematic

2.2 Instruction Memory

While not being part of DLX microprocessor itself, we included an Instruction Memory in the design so that exhaustive simulations could be done. The memory structure is very simple, but the process describing it is not the usual memory process. The process is in fact in charge of filling the Instruction Memory with the firmware. As soon as the memory is filled, the simulation can start.

2.3 Data Memory

As mentioned for the Instruction Memory, the Data Memory is outside the scope of the microprocessor, either. Still, we decided to insert it in the Top Level Entity design in order to test the correct functioning of Load/Store instructions. The process describing it is the simplest one, with a reset and two enables in order to activate read and write functions.

2.4 Control Unit

Among the three possible choices on the Control Unit approaches we decided to pick the Hardwired Control Unit. The reason behind this choice is driven by the fact that opcodes are encoded using the standard counting approach. Thus, an FSM-based approach is to be discarded in favor of the hardwired approach.

The control word

The control word is split into a sequence of bits for the control unit namely "aluOpcode" and the other part that is driving the selection signals of the multiplexers and the enables for both register file and data memory. An in-depth description of the control signals follows:

- AluSrc (ALU source) : Acts on the ALU MUX multiplexer. If at '1' then chooses the immediate, otherwise chooses the 2nd out port coming from the register file.
- Jump : If it is '1' then a Jump instruction is in the pipeline.
- Branch: If it is '1' then a Branch instruction is in the pipeline.
- AluOpCode : It is a 4-bits signal used for driving the ALU to perform the correct operation on the inputs. **This is not coming from the LUT, but it is coming from the decoder as shown in figure 2**.
- WE (Data memory write enable) : It is driving the memory to perform a write.
- RE (Data memory read enable) : It is driving the memory to perform a read.
- MemToReg (Memory to register) : Acts on the MEM MUX multiplexer, if it is '1' then the data to be written inside the register comes from the memory, else if it is '0' it comes from the ALU.
- RegDst (Register destination) : It is a signal used to drive the ADDR WR MUX in order to choose the register destination as mentioned in Datapath section.
- Jal (Jump and link) : It is once again a signal that is activated when a JAL instruction is performed. If it is '1' then the destination register becomes R31. On top of that, it also drives the JAL MUX which is in charge of propagating the address from the PC up to the write port of the register file.
- WR_EN (Register Write) : This signal is used to drive the write-enable signal of the register file.

Structure

The control unit is divided into 3 main parts (graphic representation is in Figure 2):

- The Lookup-table (LUT) : It is a memory containing the control word for each and every instruction. The size of a control word is 9 bits. The opcode is the address where the relative control word is stored.

- The ALU decoder : It's a simple $6 \rightarrow 4$ decoder mapping the OPCODE and FUNC code to a sequence of 4 bits describing the operation that ALU has to perform. If an R-type instruction is received, then the decoder takes the FUNC code and decodes it into the correct ALU operation. Alternatively, in case of I-type or J-type the decoder is in charge of generating the ALU opcode starting only from the OPCODE.
- Skewing network : This part is used in order to delay the arrival of control signals to the deeper stages of the pipeline. In fact, if not present, at each clock cycle the control signals would change reflecting only the one corresponding to the new decoded instruction causing unpredictable behaviors.

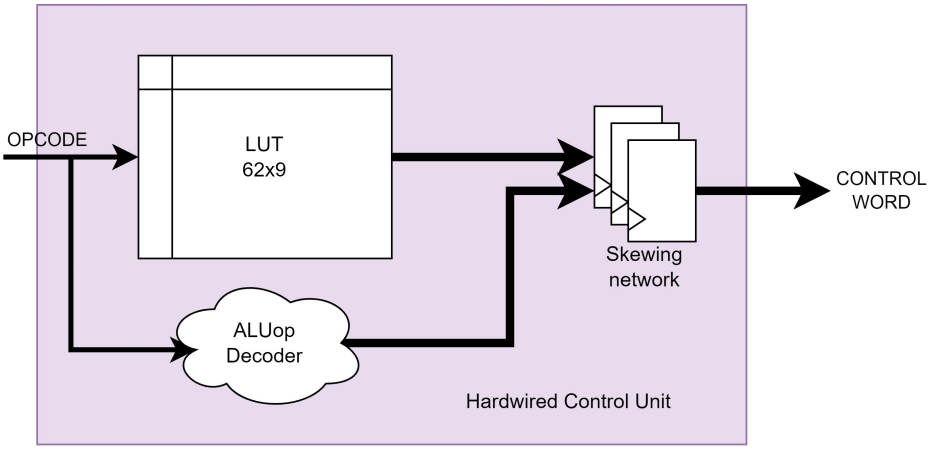

Figure 2: DLX Hardwired Control unit view

R-type example

This figure shows what happens to the control signals during the R-type. Note that not all the signals are activated at the same time, but they are activated gradually depending on the skewing network.

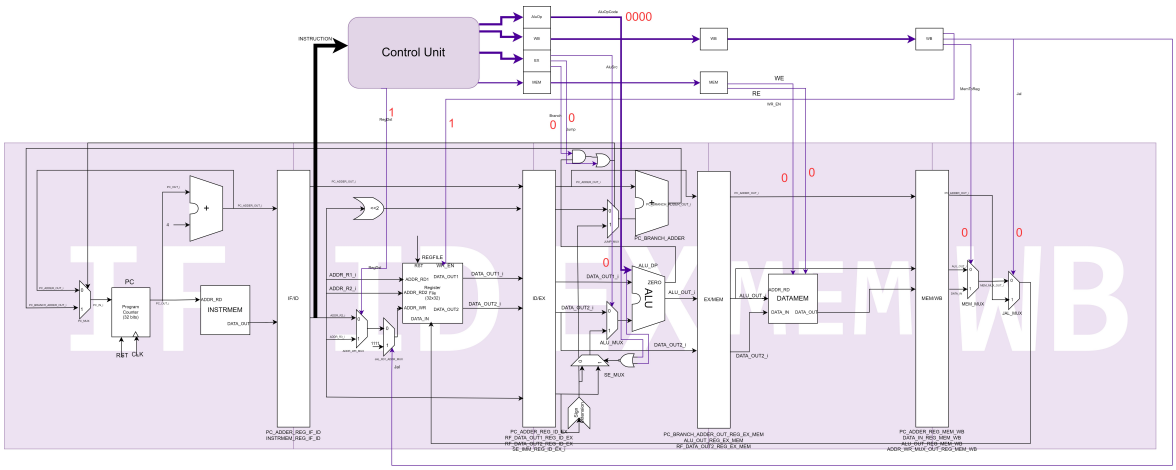

Figure 3: R-type ADD

I-type example

This figure shows what happens to the control signals during the I-type. Note that not all the signals are activated at the same time, but they are activated gradually depending on the skewing network.

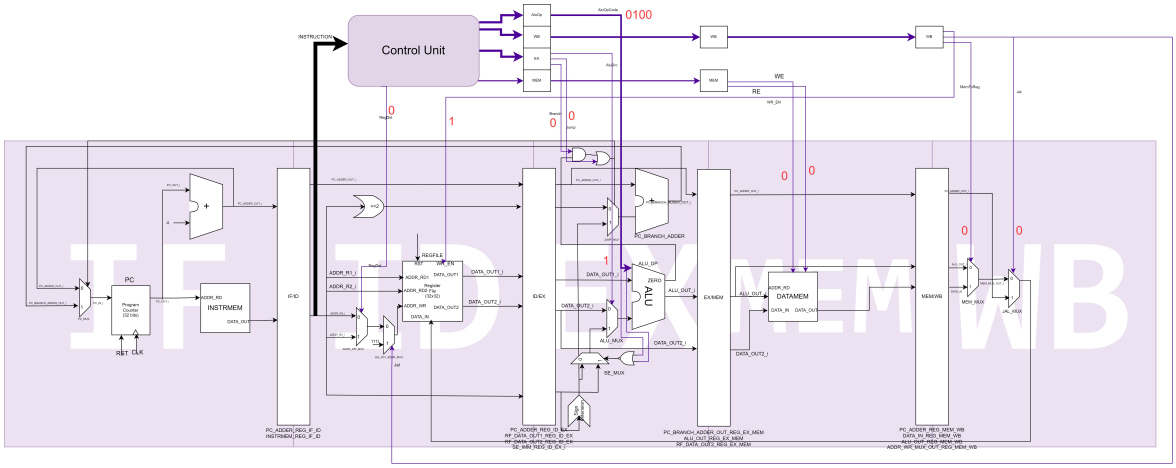

Figure 4: I-type ANDI

J-type example

This figure shows what happens to the control signals during the J-type. Note that not all the signals are activated at the same time, but they are activated gradually depending on the skewing network.

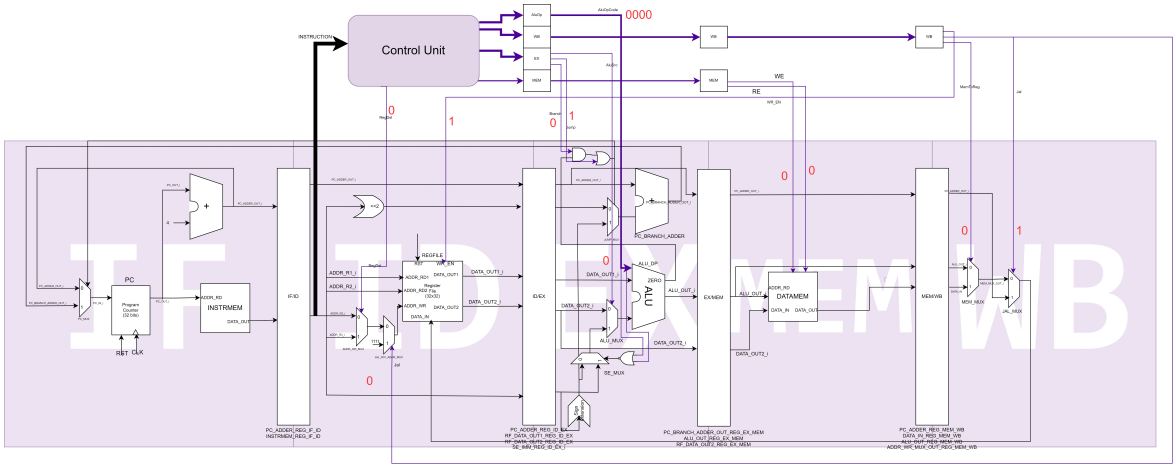

Figure 5: J-type JAL

LW example

This figure shows what happens to the control signals during the LW. Note that not all the signals are activated at the same time, but they are activated gradually depending on the skewing network.

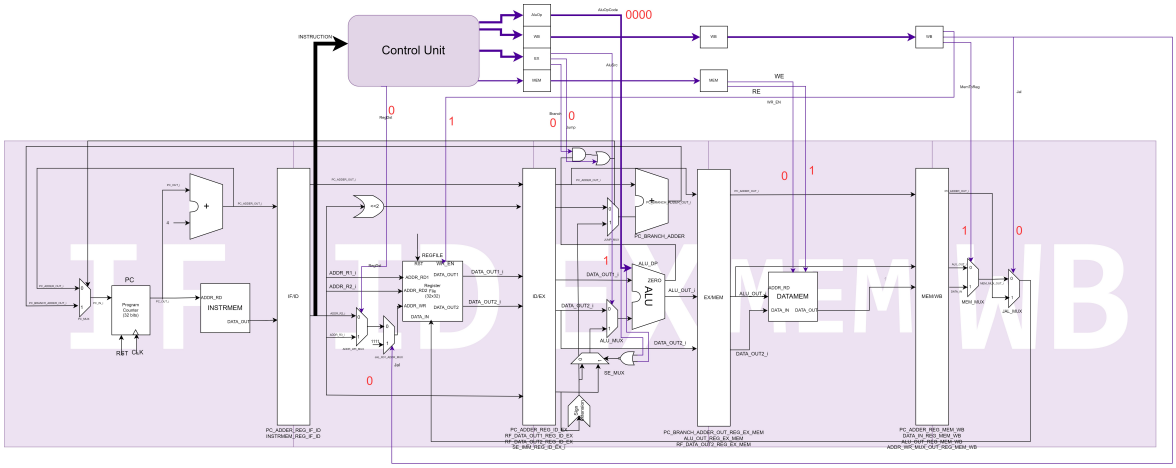

Figure 6: Load word

2.5 CPU

All the above mentioned components are connected together in a top level entity that simply acts as a wrapper.

2.6 Testbenches

The last step of our DLX design was to test the correct functioning of the microprocessor with each and every instruction of the implemented instruction set.

Assembler programs

We prepared 3 different test programs in order to test the DLX:

- I-Type testbench. The goal of this testbench is to demonstrate how the DLX behaves in case of I-Type instructions, managing the read/write from/to the RF, the sign extension of immediates and the correctness of ALU results
- R-Type testbench. Very similar to the I-Type testbench but this time with only R-Type instructions
- Division testbench. The goal of this testbench is to simulate a possible complete program running on our DLX. The procedure is an iterative division between two registers. On top of that, Load/Store and Jump/Branch instructions are tested to check the correctness of the PC update portion of the DLX.

Obtained results

I-type In this simulation we can observe that each I-type instruction is tested, at the beginning addition and subtraction is performed and after that a set of masks are applied to registers. Eventually, the program halts by calling a jump on the address of the jump itself.

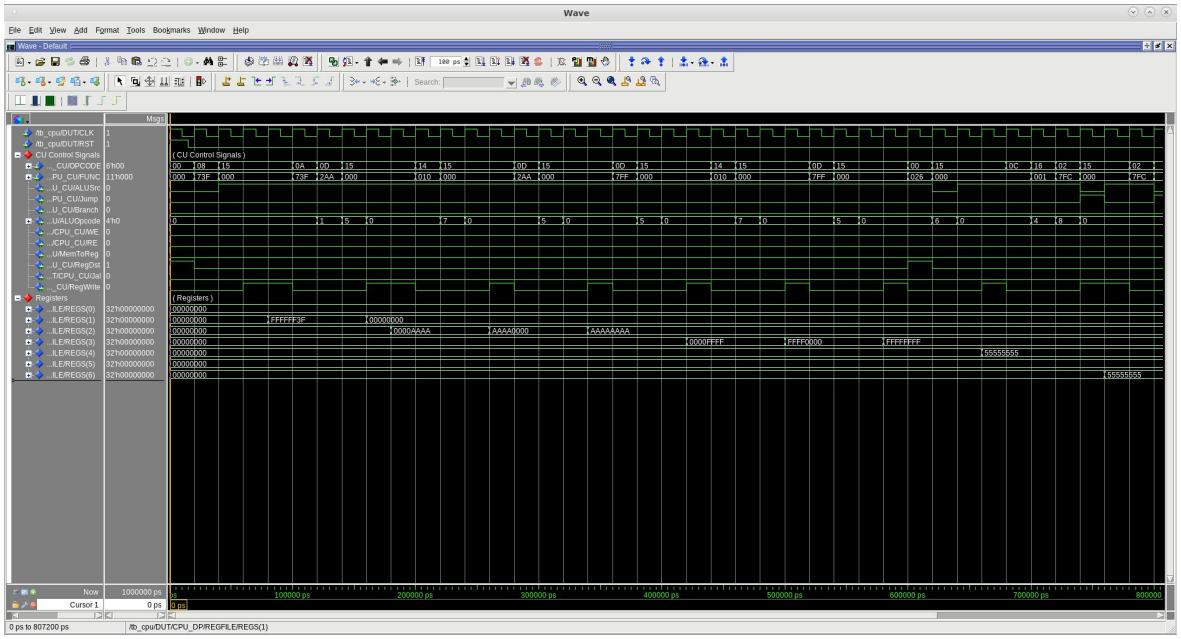

Figure 7: Waveforms taken from the I-type assembler program

R-type In this simulation we can observe that each R-type instruction is tested, at the beginning the registers are loaded with initial values, then some shift operations are performed using only R-type instructions. Eventually, the program halts by calling a jump on the address of the jump itself.

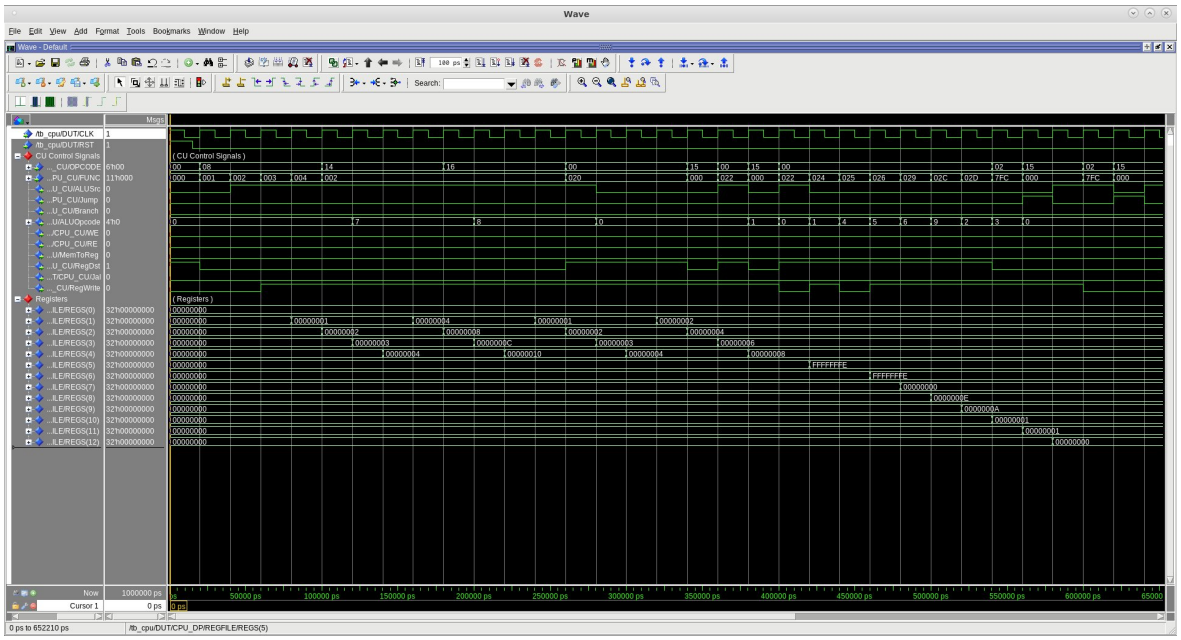

Figure 8: Waveforms taken from the R-type assembler program

Division In this simulation, we perform an iterative division algorithm. For brevity the operands for the division are 81/27. We proceed by performing a call to a procedure, using JAL instruction, and the end it returns back to the caller by using J because JR is not implemented in the basic set of operations. Before halting, the program executes a SW on the address 0 of the data memory, and right after it performs a LW on a register.

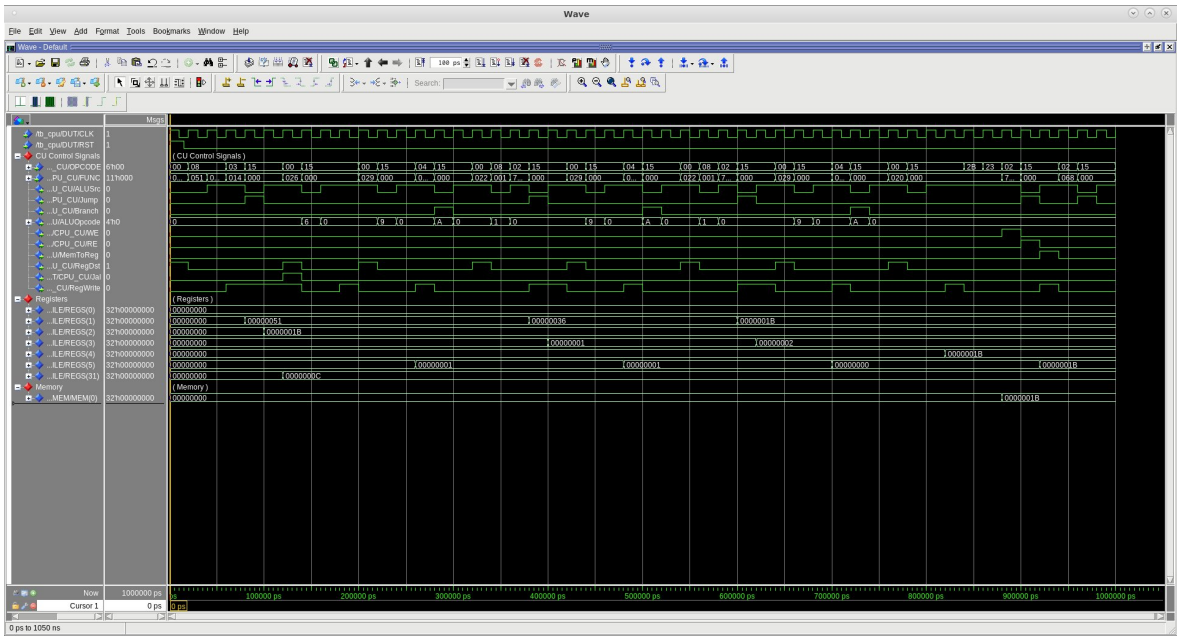

Figure 9: Waveforms taken from the Division assembler program, it shows 81/27 division

3 Synthesis

After verifying the simulated results, we wrote a simple synthesis script in order to be run by Design Compiler. In fact, the script creates the clock with a period of 1 ns and then starts compiling the design with a high map effort phase.

3.1 Flow

Initially the ALU was composed of a RCA to perform additions and subtractions, and this was not a problem until we started performing the synthesis with a lower clock period.
After several analysis over the critical path, we came up to the conclusion that a faster adder was necessary. Therefore, the Pentium-4 adder designed in the previous laboratories replaced the slow RCA. In order to respect the correct flow, we did again the simulation steps and begun again the synthesis phase, this time achieving a positive slack even at higher frequencies. However, we decided to maintain 1GHz as frequency in order not to have hold/setup violations during the layout phase.

3.2 Reports

Power report

From the static analysis it can be observed that the Total Power dissipated is 426 mW. This power could be lowered in future versions by applying clock gating techniques to the unused combinational modules composing the control unit and the datapath. In fact, a clear example is that of the P4 adder which could be gated during execution of Jump instructions and viceversa for the gating of the PC branch adder during R-type or I-type instructions.

	Internal	Switching	Leakage	Total	
Power Group	Power	Power	Power	Power	(%)
io_pad	0.0000	0.0000	0.0000	0.0000	(0.00%)
memory	0.0000	0.0000	0.0000	0.0000	(0.00%)
black_box	0.0000	0.0000	0.0000	0.0000	(0.00%)
clock_network	0.0000	0.0000	0.0000	0.0000	(0.00%)
register	3.2325e+03	35.6278	3.6556e+04	3.3047e+03	(77.54%)
sequential	306.2304	4.7966e-02	4.2248e+04	348.5265	(8.18%)
combinational	123.7337	346.0545	1.3904e+05	608.8256	(14.28%)
Total	3.6625e+03 uW	381.7302 uW	2.1784e+05 nW	4.2621e+03 uW	

Timing report

The timing report, shows the critical path obtained. As we can see, the critical path goes through the Pentium 4 adder and confirms that in our design the ALU is the bottleneck for the clock frequency. In order to mitigate this problem, we could use aggressive ungrouping techniques so that Design Compiler has more freedom to reorganize the gates and applying more efficient mapping techniques. Furthermore, "compile_ultra" can be used in order to have a more effective compilation.

Point	Incr	Path
clock CLK (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
CPU_DP/RF_DATA_OUT2_REG_ID_EX/O_reg[0]/CK (DFFR_X1)	0.00	0.00 r
CPU_DP/RF_DATA_OUT2_REG_ID_EX/O_reg[0]/Q (DFFR_X1)	0.09	0.09 f
CPU_DP/RF_DATA_OUT2_REG_ID_EX/O[0] (REG_BITS32_7)	0.00	0.09 f
CPU_DP/ALU_MUX/IN0[0] (MUX21_NBIT32_5)	0.00	0.09 f
CPU_DP/ALU_MUX/U12/Z (MUX2_X2)	0.08	0.17 f

```
CPU_DP/ALU_MUX/O[0] (MUX21_NBIT32_5)                    0.00       0.17 f
CPU_DP/ALU_DP/OP2[0] (ALU)                              0.00       0.17 f
CPU_DP/ALU_DP/ADDER/B[0] (adder_generic_NBIT32_NBIT_PER_BLOCK16)
                                                        0.00       0.17 f
CPU_DP/ALU_DP/ADDER/U44/ZN (XNOR2_X1)                   0.08       0.25 f
CPU_DP/ALU_DP/ADDER/carry/B[0] (CLA_NBIT32_NBIT_PER_BLOCK16)
                                                        0.00       0.25 f
CPU_DP/ALU_DP/ADDER/carry/PG_0_0/bi (pgb_0)             0.00       0.25 f
CPU_DP/ALU_DP/ADDER/carry/PG_0_0/U2/Z (XOR2_X1)         0.08       0.33 f
CPU_DP/ALU_DP/ADDER/carry/PG_0_0/p (pgb_0)              0.00       0.33 f
CPU_DP/ALU_DP/ADDER/carry/G_0_0/Pik (G_0)               0.00       0.33 f
CPU_DP/ALU_DP/ADDER/carry/G_0_0/U2/ZN (AOI21_X1)        0.04       0.37 r
CPU_DP/ALU_DP/ADDER/carry/G_0_0/U1/ZN (INV_X1)          0.03       0.40 f
CPU_DP/ALU_DP/ADDER/carry/G_0_0/Gij (G_0)               0.00       0.40 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_1/Gdkj (G_5)      0.00       0.40 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_1/U2/ZN (AOI21_X1)
                                                        0.04       0.44 r
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_1/U1/ZN (INV_X1)
                                                        0.03       0.47 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_1/Gij (G_5)       0.00       0.47 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_2/Gdkj (G_4)      0.00       0.47 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_2/U2/ZN (AOI21_X1)
                                                        0.04       0.51 r
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_2/U1/ZN (INV_X1)
                                                        0.03       0.53 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_2/Gij (G_4)       0.00       0.53 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_3/Gdkj (G_3)      0.00       0.53 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_3/U2/ZN (AOI21_X1)
                                                        0.04       0.57 r
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_3/U1/ZN (INV_X1)
                                                        0.03       0.60 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_3/Gij (G_3)       0.00       0.60 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_4/Gdkj (G_2)      0.00       0.60 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_4/U3/ZN (NAND2_X1)
                                                        0.03       0.63 r
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_4/U2/ZN (NAND2_X1)
                                                        0.03       0.66 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_4/U1/Z (BUF_X4)
                                                        0.07       0.73 f
CPU_DP/ALU_DP/ADDER/carry/INITIAL_G_i_4/Gij (G_2)       0.00       0.73 f
CPU_DP/ALU_DP/ADDER/carry/Co[0] (CLA_NBIT32_NBIT_PER_BLOCK16)
                                                        0.00       0.73 f
CPU_DP/ALU_DP/ADDER/sum/Ci[1] (SUM_GENERATOR_GENERIC_NBIT_PER_BLOCK16_NBLOCKS2)
                                                        0.00       0.73 f
CPU_DP/ALU_DP/ADDER/sum/CARRYSELBLOCK_i_2/C_IN (CARRY_SELECT_BLOCK_GENERIC_NBIT16_1)
                                                        0.00       0.73 f
CPU_DP/ALU_DP/ADDER/sum/CARRYSELBLOCK_i_2/MUX/SEL (MUX21_GENERIC_NBIT16_1)
                                                        0.00       0.73 f
CPU_DP/ALU_DP/ADDER/sum/CARRYSELBLOCK_i_2/MUX/U31/Z (MUX2_X1)
                                                        0.09       0.82 f
CPU_DP/ALU_DP/ADDER/sum/CARRYSELBLOCK_i_2/MUX/Y[14] (MUX21_GENERIC_NBIT16_1)
                                                        0.00       0.82 f
CPU_DP/ALU_DP/ADDER/sum/CARRYSELBLOCK_i_2/S[14] (CARRY_SELECT_BLOCK_GENERIC_NBIT16_1)
                                                        0.00       0.82 f
```

```
CPU_DP/ALU_DP/ADDER/sum/S[30] (SUM_GENERATOR_GENERIC_NBIT_PER_BLOCK16_NBLOCKS2)
                                                        0.00       0.82 f
CPU_DP/ALU_DP/ADDER/S[30] (adder_generic_NBIT32_NBIT_PER_BLOCK16)
                                                        0.00       0.82 f
CPU_DP/ALU_DP/U109/ZN (AOI22_X1)                        0.06       0.88 r
CPU_DP/ALU_DP/U110/ZN (OAI221_X1)                       0.06       0.93 f
CPU_DP/ALU_DP/RES[30] (ALU)                             0.00       0.93 f
CPU_DP/ALU_OUT_REG_EX_MEM/I[30] (REG_BITS32_5)          0.00       0.93 f
CPU_DP/ALU_OUT_REG_EX_MEM/O_reg[30]/D (DFFR_X1)         0.01       0.95 f
data arrival time                                                  0.95

clock CLK (rise edge)                                   1.00       1.00
clock network delay (ideal)                             0.00       1.00
CPU_DP/ALU_OUT_REG_EX_MEM/O_reg[30]/CK (DFFR_X1)        0.00       1.00 r
library setup time                                     -0.05       0.95
data required time                                                 0.95
-----------------------------------------------------------------------------
data required time                                                 0.95
data arrival time                                                 -0.95
-----------------------------------------------------------------------------
slack (MET)                                                        0.00
```

4 Physical Layout

The final step of our project has been the layout of the DLX microprocessor. It is important to note that the steps in the following paragraphs have been extracted from the Place&Route LAB with some adaptations. This has been carried out simply as a demonstrative step of the design. Of course, this layout is not sufficient to make the processor actually work in a real chip implementation as an accurate case study would be needed but at the moment it is outside the scope of this project.

4.1 Flow

Innovus environment and files As first step of the procedure, some files need to be prepared. In particular:

- **DLX_postsyn.v**: the Verilog netlist of the DLX processor created by Design Compiler after the synthesis procedure.
- **DLX.sdc**: file containing design constraints extracted with Design Compiler after the synthesis procedure.
- **Default.view**: another environment file, together with other parameters, of particular interest for us is the reference to the *sdc* to link necessary design constraints.
- **DLX.globals**: this file contains basic project information and references for Innovus, such as libraries to be used, working directories, power nets reference (*vdd* and *gnd* in our case) and the reference to circuit netlist. Moreover, it is crucial since the actual design will be loaded into Innovus environment by reading this specific file.

Structuring the floorplan In this step we provided parameters to impose a squared shape to the chip and to reserve the physical space necessary for the main power lines. The needed area is automatically computed by Innovus by reading environment files (mainly the netlist).

Power Rings At this point, *vdd* and *gnd* rings are placed along the perimeter of the chip, exploiting the highest metal layers (i.e. *M9* and *M10*) since a high amount of current needs to be delivered/collected from/by the power lines.

Power Stripes Power rings alone are not enough, so stripes have to be placed as well for correct power delivery. Care must be taken in distancing the stripes: not too few or they would not be able to supply enough power, not too many (too close) or they will be problematic in successive placement stages. For reference, an issue we faced with close power stripes is exceeding the design density with consequent failures at post-CTS optimization time.

Std Cell Power routing This is a very simple step which can be carried out automatically, in which power wiring for the cells is provided in the lower metal layer.

Placement Just before running the placement procedure, some details need to be adjusted. In particular, we let Innovus free of placing cells among *M1-M8* metal layers, so that layers *M9* and *M10* would be reserved for *vdd* and *gnd* in order to avoid congestion problems.

I/O Pins Placement Coming to pin placement, our approach has been to spread them around the chip in the most even configuration. We started from *clk*, *rst*, and control signals, which are single wires, and placed them at the bottom of the chip spreading them from the center. The input and output data pins for the Data Memory (two 32-bit buses) have been spread alongside on the left side of the chip. An identical approach for the buses (two of 32-bits each) going to the Instruction Memory has been used, besides placing them on the right side of the chip. The top side of the chip has been occupied by one 32-bit bus which is reserved for the Data Memory Address.

Post Clock-Tree-Synthesis (CTS) optimization This procedure attempts to optimize the design before the routing process, and it helps achieving the required timing constraints. It has been carried out for both *setup* and *hold* time conditions.

Fillers Another crucial step before routing is placing the fillers within the actual design. Fillers have the key role of improving the mechanical strength of the chip, since silicon alone would be very weak and its cave in would cause mechanical stress over other components (mainly connection lines) and eventually lead to chip failure. For this purpose, it is good practice to select all available type of fillers available in the libraries which are being used, so that Innovus will be free to choose the most appropriate ones. It has to be noted that fillers won't have any impact on the designed circuit since they will be connected so that they never activate and are not part of the circuit logic.

Routing The routing process has been treated as a completely automated procedure, but it must be underlined that it is not so straightforward. However potential issues have not been considered as outside the scope of this project.

Post routing optimization This step is again attempting to optimize the design in order to respect the required timing constraints, and it has been carried out once more for both *setup* and *hold* timings.

Reports and data extraction Being now the design complete, we proceeded with key information extraction:

- Parasitic resistance and capacitance
- Timing reports, analysis carried out for both *setup* and *hold* types
- Gate count (after verification)
- Post-route netlist in verilog format (after verification)
- Delay annotation in SDF (after verification).

Moreover, we took care of verifying that no violated constraints appeared in the design.

Design analysis and verification Finally, we launched the verification tools provided by Innovus to verify both *Connectivity* and *Geometry*, in order to verify respectively the absence of floating wires and of wrong constraints on geometric features.

4.2 Results

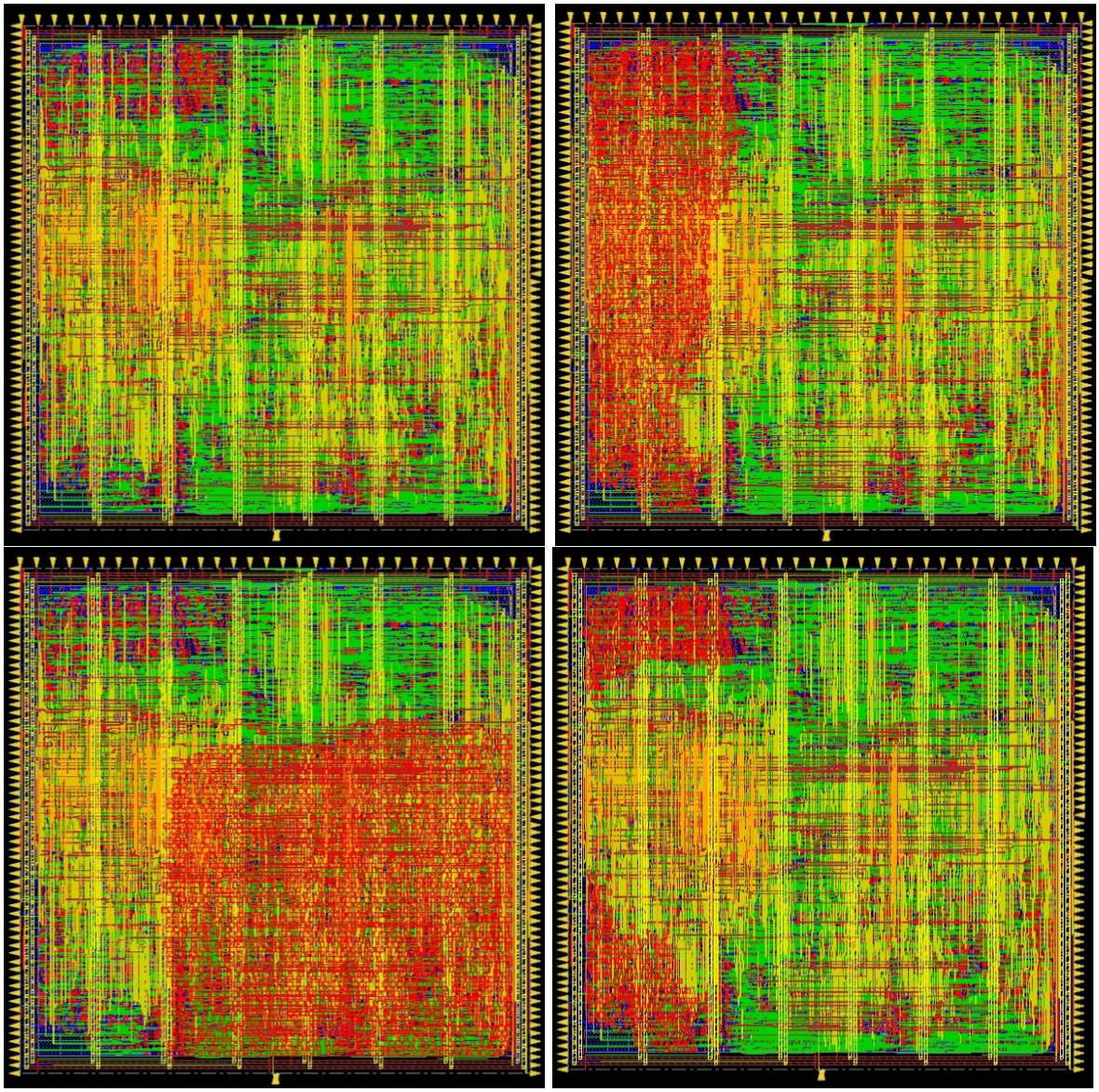

Figure 10: CU (top-left) - ALU (top-right) - RF (bottom-left) - p4 Adder (bottom-right)

5 Possible Improvements

Among the future improvements on our DLX version, there are:

- Data Hazard unit
- Branch Prediction Unit
- Floating point unit
- Instruction/Data cache
- Extended Instruction set
- Reducing to 1 the Branch delay slot
- Verification using UVM

6 Group Workflow

In order to keep track of the file versioning, we decided to use the Git versioning system. Furthermore, the in-presence meetings were divided in a brainstorming phase and after the coding phase during which we adopted a pair programming technique (i.e. one member writes the VHDL code and the other two members check the consistency of the code and the design, swapping roles on the go)