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_04
Dalmasso Luca, Gai Omar

22 Settembre 2021

CONTENTS

Chapter 1

DLX Microprocessor

Design of a 32 bit RISC processor architecture made of an Hardwired Control Unit and a 4 stages Harvard architecture Datapath:

1) Fetch Unit
2) Decode Unit
3) Execution Unit
4) Memory/Write Back Unit

The following picture is the top level view of the DLX:

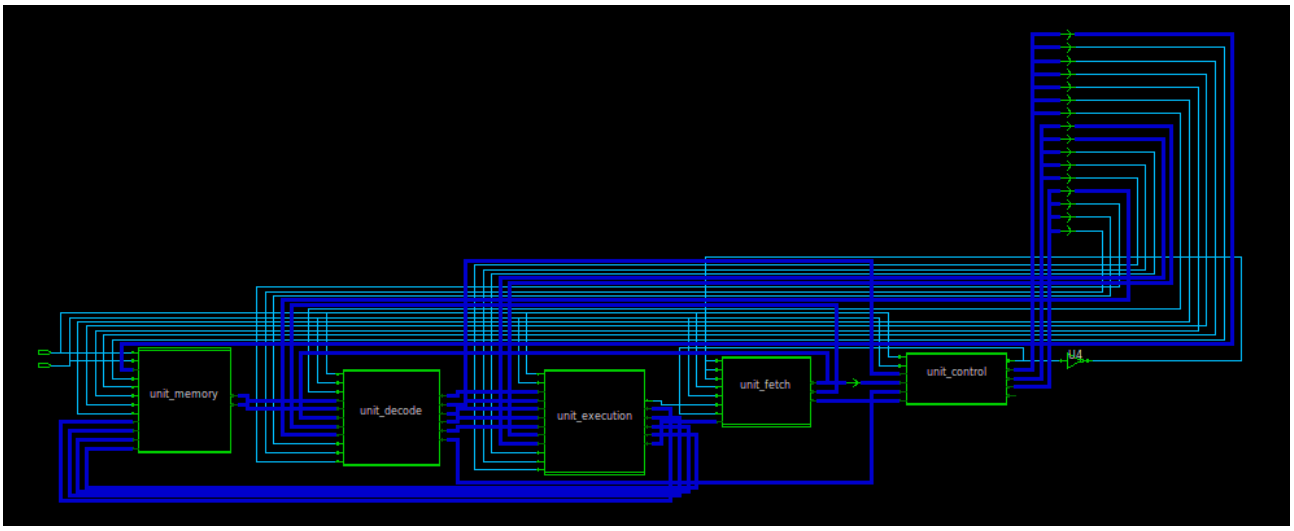


Figure 1.1 Top Level view of DLX

The version we designed offers the following functionalities:

1) **Basic instruction set**: `J, JAL, BEQZ, BNEZ, ADDI, SUBI, ANDI, ORI, XORI, SLLI, NOP, SRLI, SNEI, SLEI, SGEI, LW, SW, SLL, SRL, ADD, SUB, AND, OR, XOR, SNE, SLE, SGE.`
2) **Advanced instruction subset:** `ADDUI, SUBUI, JR, JALR, SRAI, SEQI, SLTI, SGTI, LB, LH, LBU, LHU, SB, SH, SLTUI, SGUI, SLEUI, SGEUI, SRA, ADDU, SUBU, SEQ, SLT, SGT.`
3) **Optimized ALU:** designed using arithmetical structures seen during the lectures.
4) **Basic hazard detection for solving RAW hazards.**
5) **Scripts for compilation and synthesis.**

Instructions are on 32 bits and are encoded as hexadecimal numbers. Are of different type:

- Immediate type instructions I-TYPE
- Register-Register instructions R-TYPE (OPCODE is always “000000”, FUNC field defines the single instruction).
- Jump type instructions J-TYPE

In chapter 3, section 3.1 the distinction is clearer.

In figure 1.2 is shown the complete Datapath schematic, in figure 1.3 the Hardwired Control Unit.

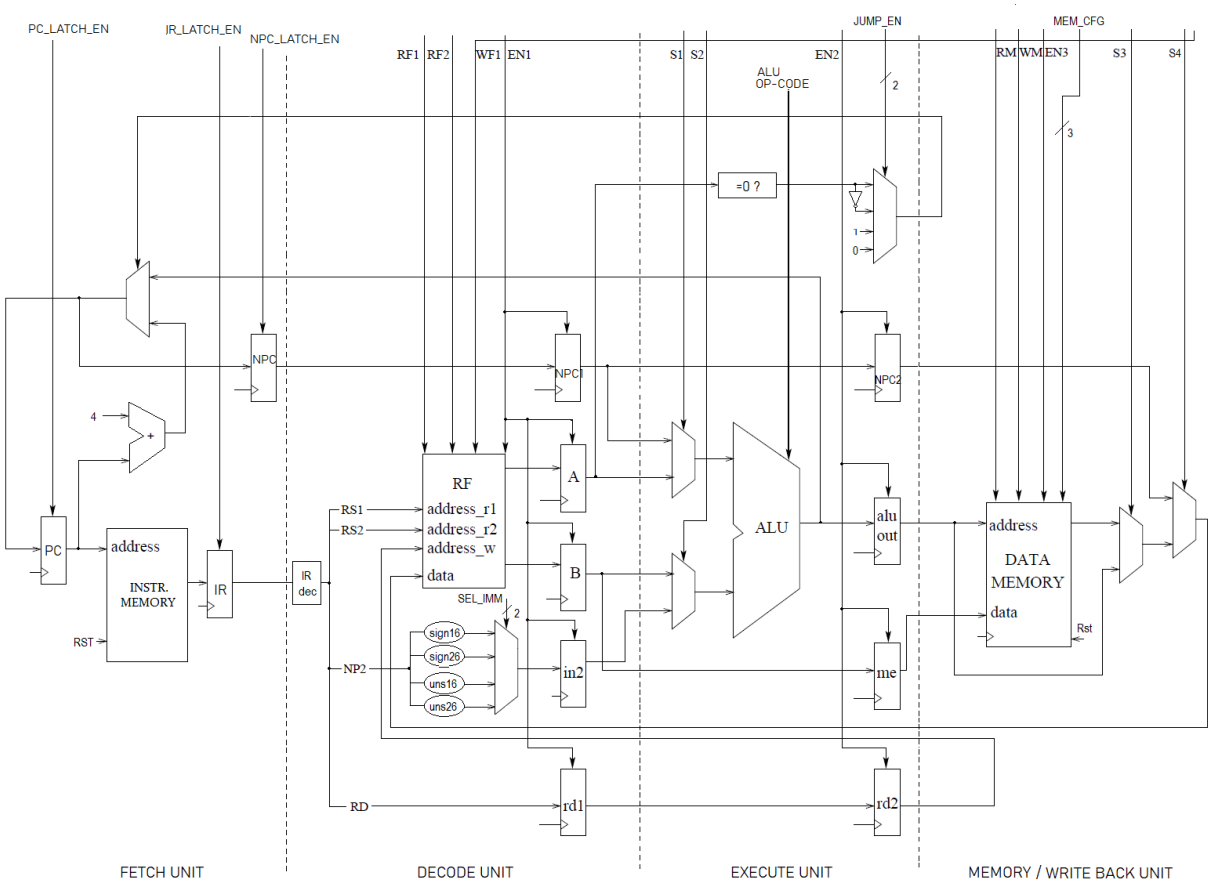


Figure 1.2 Datapath

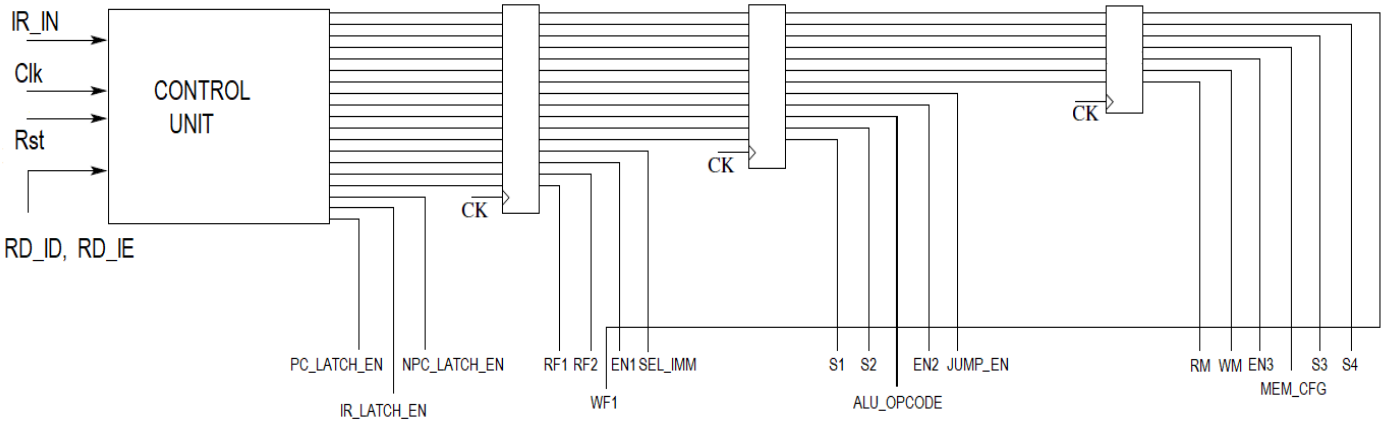


Figure 1.3 Hardwired Control Unit

Chapter 2

Fetch Unit

Fetch Unit is the first stage of Datapath. From this unit instructions are fetched (loaded) from Instruction Memory and passed to the next stage.

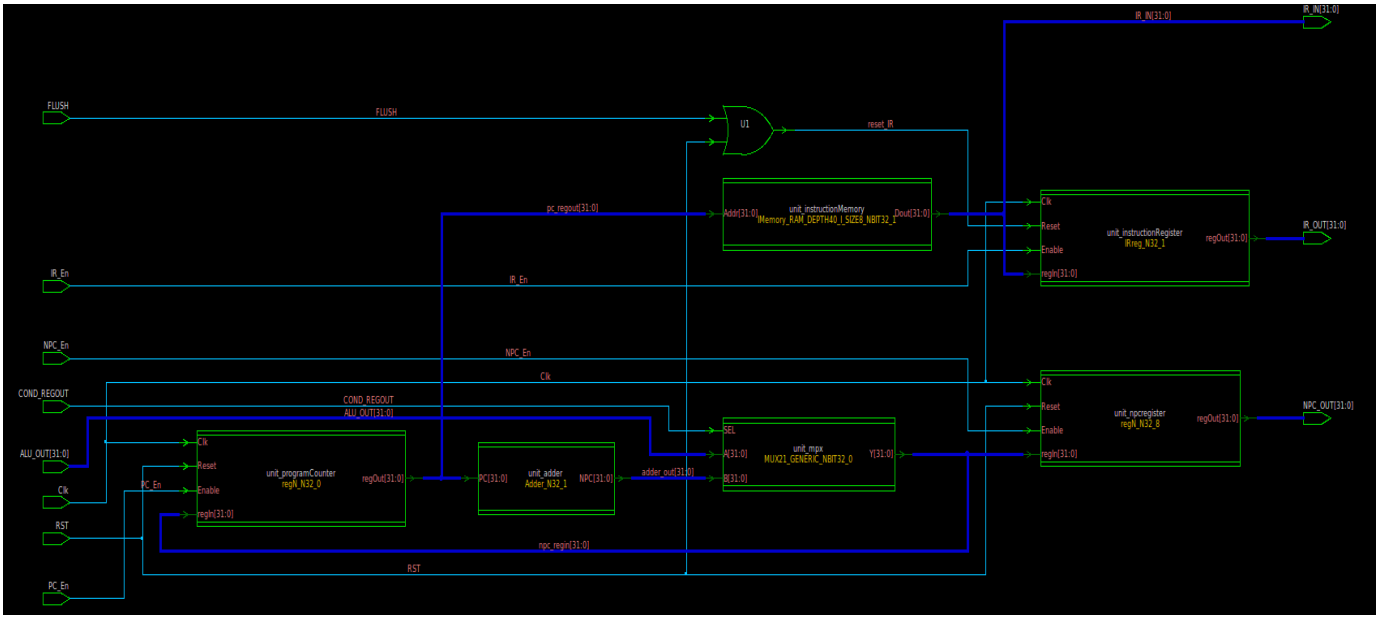


Figure 2.1 Fetch Unit top level view

As shown in the Figure 2.1 the fetch stage consists of:

1) **Program Counter Register PC:** 32 bit register that stores the address for accessing the instruction memory.
2) **Instruction Register IR:** 32 bit pipeline register that stores the instruction fetched.
3) **Next Program Counter Register NPC**: 32 bit register that is used for the propagation of the PC through the pipeline, mandatory component for all jumps and branches.
4) **Adder:** computes the PC+4 value, address of the next instruction to be fetched.
5) **Multiplexer:** allows the selection between PC+4 and the address coming from the execution unit.

When the selection signal ("COND_REGOUT" in the figure 2.2), coming as well from the execution unit, is low it means that normal execution flow needs to be maintained and so the multiplexer selects PC+4.

There's another important signal, coming from the control unit, called "FLUSH". When FLUSH signal is high, basically forces the fetch unit to stall, by disabling enable signals (XOR gate in figure 2.1) , and to reset the IR register.

When the Reset signal is active (high), the output of the IR register is **a NOP instruction**.

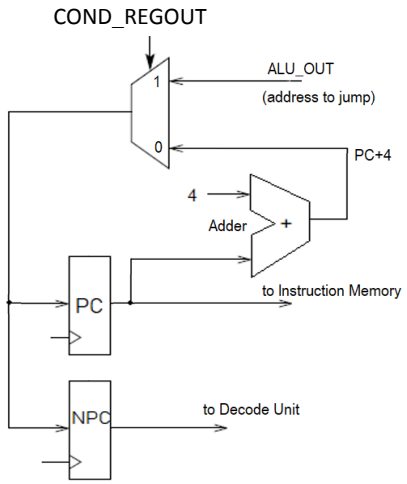


Figure 2.2 PC + NPC + Adder

When selection signal COND_REGOUT is:

- equal to '1' -> selects ALU_OUT , make jump

- equal to '0' -> selects PC+4, normal flow

2.1 Instruction Memory

The Instruction Memory is a RAM memory made of rows of 8 bits each. A set of 4 consecutive bytes corresponds to a unique instruction, which is on 32 bits.
There are two main processes, one for reading from a file (and filling the memory) and another for writing data on the output port if a new address is present.
The address signal comes from the PC register and the output goes to the IR register.
A Reset signal is also present, that is activated when the system is turned ON. When active, memory is filled with the content of a file.

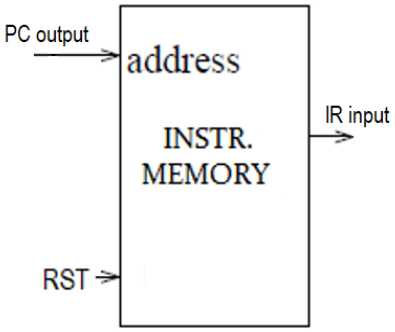


Figure 2.2 Instruction Memory

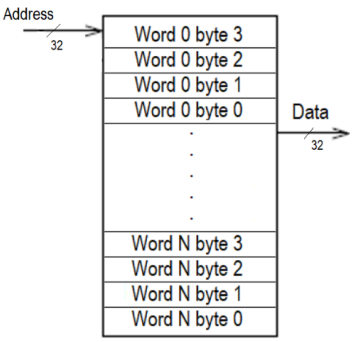


Figure 2.3 Internal structure of Instruction Memory

2.1.1 Read and Write Processes

The read process starts when the Reset signal is equal to 1, loads instructions from the file **test.asm.mem**, row by row, and saves the content inside memory. Every single row of the file contains a single instruction, on 32 bits, that is divided into four bytes before storing.

Write process is asynchronous and provides new data on 32 bits when the address changes. Four bytes are read at the same time (the Most Significant Byte is at the address coming from PC).

Chapter 3

Decode Unit

Decode Unit is the second stage of the pipeline.

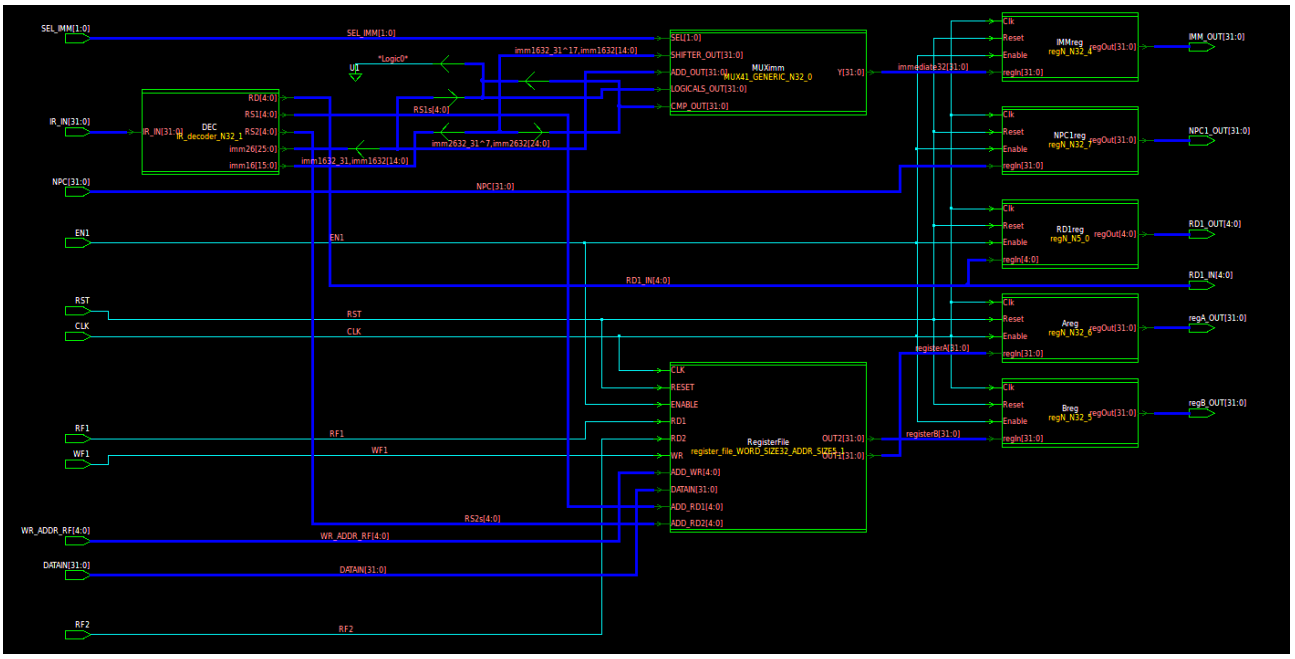


Figure 3.1 Decode Unit top level view

The Figure 3.1 shows the main components:

1) IR DECODER
2) Immediate Management: a block for sign extension (immediate on 16/26 bits are extended to 32 bits and can be treated as signed or unsigned)
3) Register File
4) Five registers to pipeline the data: NPC1, A,B,in2 and Rd1.

Every clock cycle arrives a new instruction from the Fetch Unit through IR signal and at the output of DU all data (operands for ALU, immediates) are ready for the next stage.

3.1 IR decoder

IRdecoder is a component that receives the current fetched instruction as input, on 32 bits. Every instruction is composed of bit-fields that assume different meanings, depending on the type of instruction. In figure 3.2 is clear the division in fields.

RS1 and RS2 are composed of 5 bits and contain the address_r1 and address_r2, to read from Register File operand A and operand B respectively. RD is on 5 bits and contains the address_w, to write on Register File a data coming from the last stage of Datapath.

3.1.1 Immediate management

The purpose of this unit is to generate the correct 32 bit immediate, signed and unsigned, starting from the decoded 16 or 26 bit immediate (Figure 3.3).

This unit consists of two sign extension logic (one one 16-32 bit, the other on 26-32 bit) and two unsigned extension logic and a multiplexer for choosing between them.

In figure 3.3 is shown the selection, the SEL_IMM signal comes from the control unit.

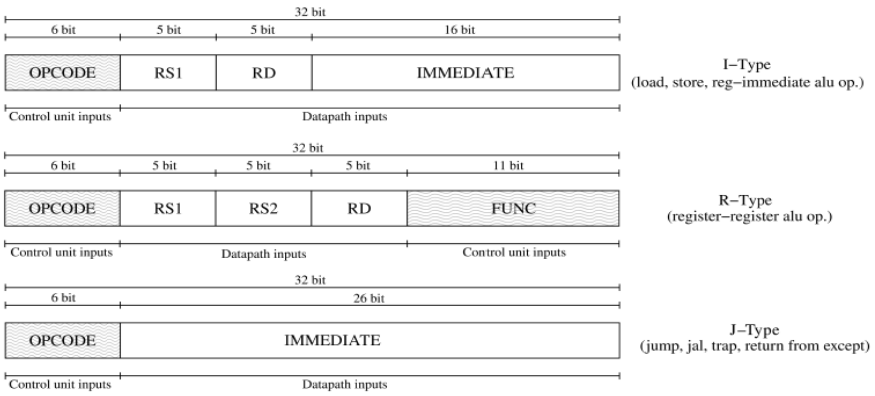


Figure 3.2: Types of instructions

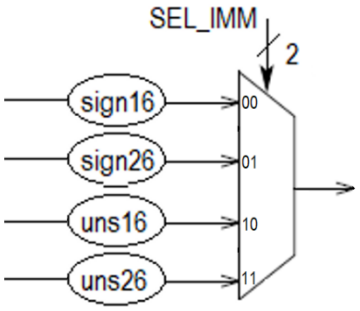


Figure 3.3: Selection between Immediates

3.2 Register File description

Register File is a component made of 32 general-purpose registers, each on 32 bits. Registers can be used to save temporary data during the execution of a program (note that the content of R0 is always zero, could not change).
It has two read ports and a single write port. The two outputs data go to the registers A and B, respectively.
When the Reset signal is asserted (active high), the Register File is cleared, and the content of each register is set to all 0's. Reset and Enable signals are synchronous to the rising edge of the clock.
The control signals from the Control Unit are RF1, RF2, WF1, EN1 and are used for writing/reading operations.

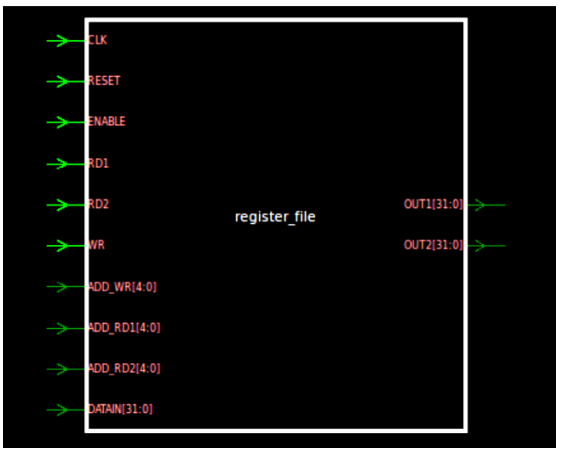


Figure 3.4 Register File

3.2.1 Read from Register File

Data read from Register File are used inside the Execution Unit, for example to perform ALU operations between the contents of two registers. The Reading process is asynchronous.
The control signals RF1 and RF2 are used with EN1: if RF1 ='1' and EN1='1' the content of one register is needed, is read at address_r1 and saved into register A.
If RF2='1' and EN='1', then the other register is needed, read at address_r2 and saved into register B. These two registers can be read both at the same time, when EN='1', RF='1' and RF2='1', for example in case of Register Type instructions.

3.2.2 Write on Register File

Write process is synchronous to the rising edge of the clock and starts when the control signals EN1='1' and WF1='1'. The data to write arrives from the last stage of Datapath and can be the output value of ALU, a value loaded from Data Memory or the value of NPC, depending on current instruction. The address comes from Rd2 register and it's synchronized with the proper address. For this reason Rd1 and Rd2 are inserted inside the Datapath, to delay the destination address coming from the IR decoder, that is inside the Fetch Unit.

3.3 NPC1, A,B,in2 and Rd1 registers.

In this stage there are also five registers on 32 bits: NPC1, A, B, IN2 and RD1. They are useful to propagate and synchronize different signals through the pipeline. NPC1 stores the value of NPC coming from Fetch Unit, A and B store the content of two registers of Register File, IN2 stores an immediate extended on 32 bits, RD1 stores the address (destination register) for writing inside Register File.

Chapter 4

Execution Unit

The Execution Unit is the third stage of the pipeline.

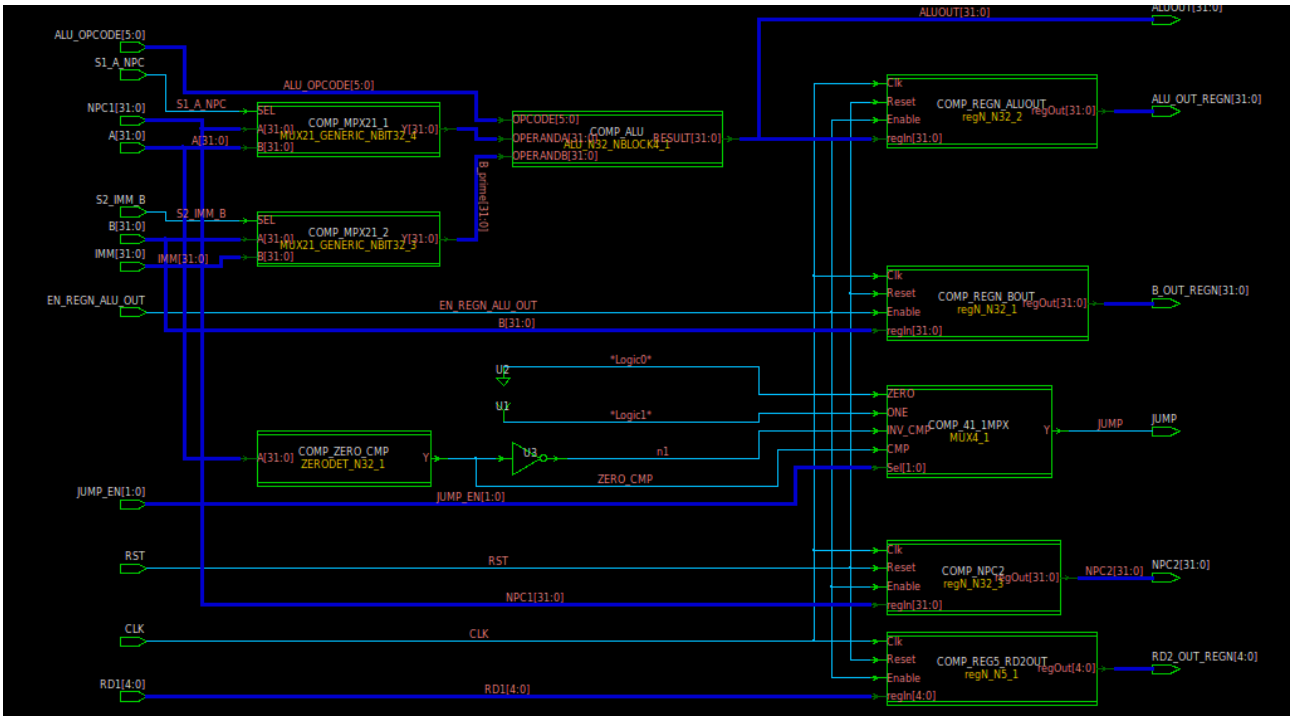


Figure 4.1 Execution Unit top level view

In this stage two important actions are done:

1) Arithmetic and Logic operations , by ALU

2) Jump and Branch management by Zero Detector and MUX4_1

The result of ALU operations are sent to next stage MEMU through ALU_OUT register, while the Jump/Branch decision is sent to first stage FU.

4.1 ALU

This component has been designed with optimized arithmetic blocks coming from both the Pentium4 and the NiagaraT2. The operations are controlled by a 6 bit signal called ALU_OPCODE that is shared between all the blocks inside the ALU (Figure 4.2).

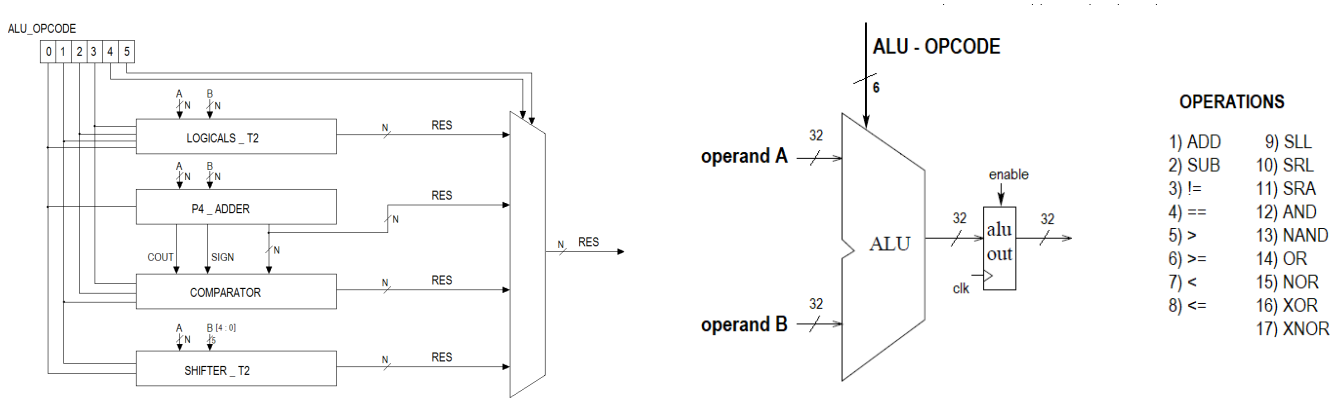


Figure 4.2 ALU schematic and functions

Depending on the content of ALU_OPCODE control signal, a different operation is executed every clock cycle. The two operands can be a combination of NPC, register A, register B and an immediate, depending on control signals S1 and S2. The output of ALU goes to the register ALU_OUT and back to the first stage FU (address of a jump is computed by ALU).
The main components are:

1) P4 Adder, which can compute sum and sub operations

2) Shifter T2, which can compute shift left logical, shift right logical, shift right arithmetic

3) Logicals, which can perform logical operations..

4) Comparator.

4.1.1 Pentium 4 Adder

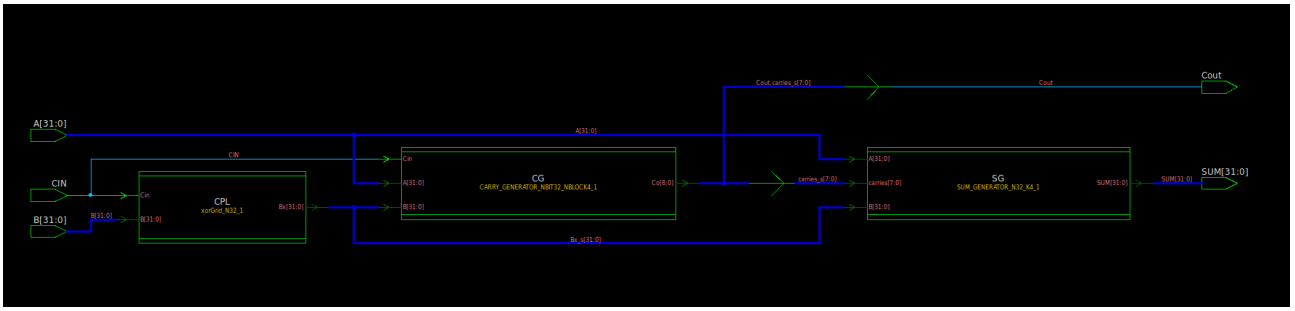


Figure 4.3 P4 Adder top level view

This component is designed to perform additions and subtractions. The top level view shown in figure 4.3 is made of 3 blocks:

1) B-COMPLEMENT: generates the 2's complement of the operand B, in case of subtraction
2) Carry Generator: generates carries every K bits
3) Sum Generator: computes sum between A and B, on blocks of K bits

Note that N is the number of bits of operands (N=32), K is the number of bits used in each block (K=4)

4.1.1.1 B-COMPLEMENT

To obtain the 2's Complement of operand B, we need to make a XOR between each single bit of B and the value of CarryIn `Bx(i) <= B(i) XOR Cin.` The result is used with operand A to compute SUM/SUB.

If the value of Cin = '0'
B(i) doesn't change.

If Cin='1' (subtraction)
B is complemented.

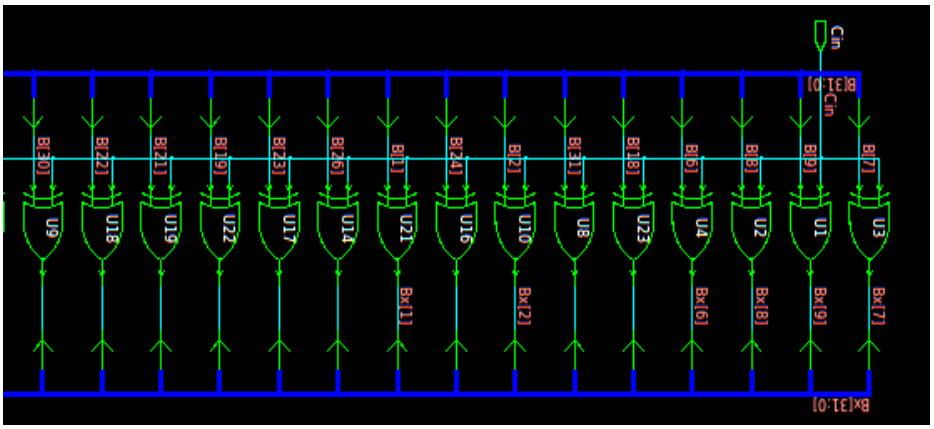


Figure 4.4 Xor Grid to obtain 2's Complement of B

4.1.1.2 Carry Generator

PG NETWORK is made of PG BLOCKs and G BLOCKs. They are the basic components to build the final structure shown in figure 4.5. This solution allows to optimize the propagation delay of carry, instead of a big chain of Full Adders, in which the delay is proportional to the length of chain.

The choice of using this kind of tree adder is due to the fact that most of the DLX instructions need to perform additions: with this structure the delay due to the carry propagation becomes logarithmic, in terms of time complexity. and this leads to a great boost of the ALU performances and, as a consequence, to the DLX itself.

A PG_BLOCK computes the terms `GIJ <= GIK or (PIK and GK1J)`

`PIJ <= PIK and PK1J`

G_BLOCK computes the term `GIJ <= GIK or (PIK and GK1J)`

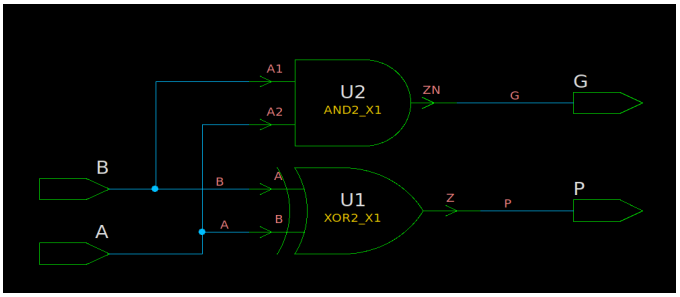


Figure 4.6 PG_BLOCK

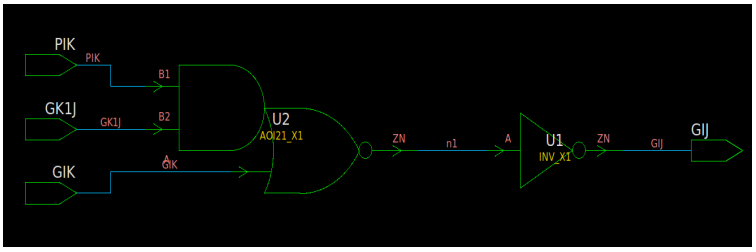


Figure 4.7 G BLOCK

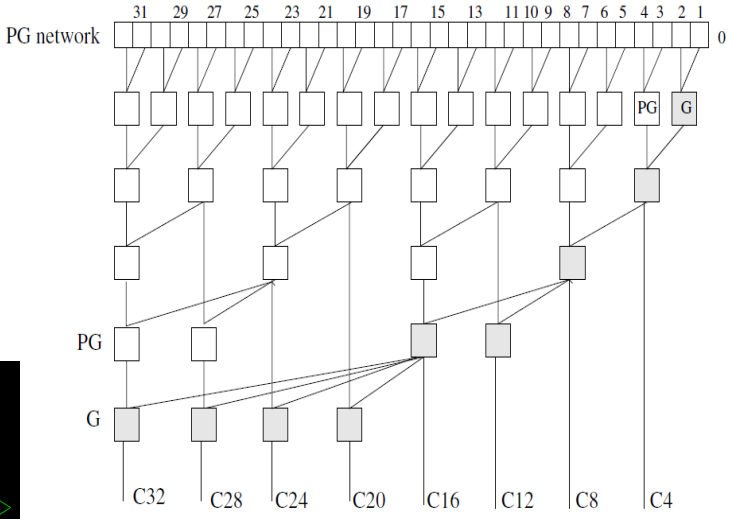


Figure 4.5 PG_NETWORK

PG NETWORK implements a sparse tree propagate network. The input signals are the two operands A and B. The carries generated at the output go to the Sum Generator: they are used as selection signals for the muxes inside the Sum Generator's blocks.

4.1.1.3 Sum Generator

It is implemented with N/K (32/4 = 8) blocks. Each block is made of two Ripple Carry Adders: first one computes the sum between A and B (on K bits), with carry in Cin='0', while the second the same sum, but with Cin='1'.
A MUX2_1 is used to choose the right sum result: the proper carry selection signal comes from the Carry Generator. The K-bits results are grouped, at the output, to obtain the final N-bits result.

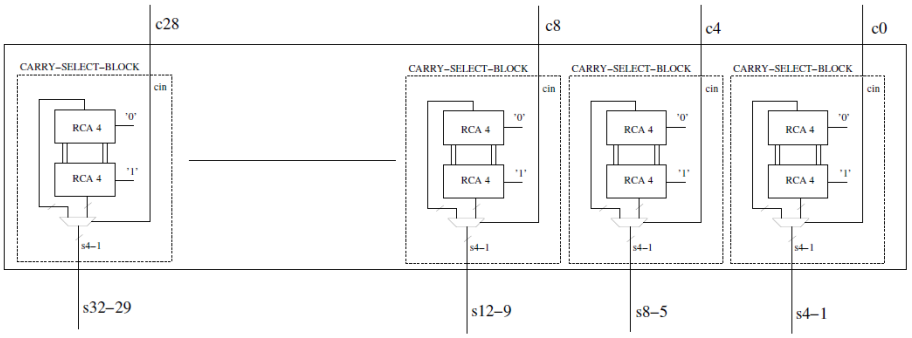


Figure 4.8 Sum Generator

A Ripple Carry Adder is a component which computes the sum between two operands on K bits.
Is made of K cascaded Full Adders .
Each FA computes Sum and CarryOut for each pair of bits A[i] and B[i], considering also Cin.
Sum is obtained by making a xor between A,B and Cin.

```
S = A xor B xor Cin
```

Cout is computed with the formula

```
c_out =((a and b) or (a and c_in) or (b and c_in))
```

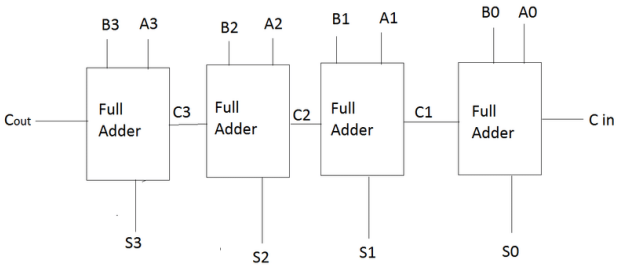


Figure 4.9 Ripple Carry Adder

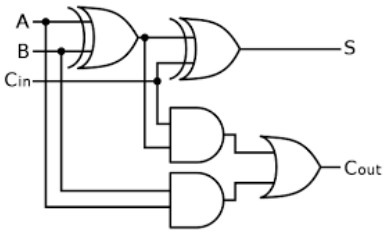


Figure 4.10 Full Adder Schematic

The additional component shown in figure 4.11 goes into the first G_BLOCK of PG_NETWORK.
In this way it is possible to compute SUBTRACTION, extending the functionality of P4 Adder designed during lab session.

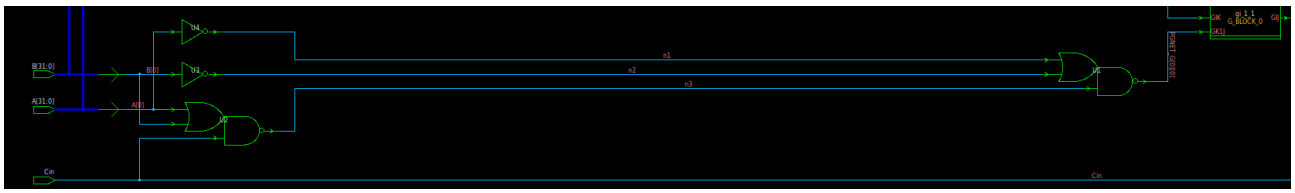


Figure 4.11 Additional component for subtraction

4.1.2 Shifter T2

The Shifter is a component of the Sparc T2 processor, adapted to a 32-bits architecture. It can perform Logic Left, Logic Right or Logic Arithmetic shifts.
The input signal A is the value to shift, the input signal B (on 5 bits) indicates the number of positions, the signal SEL indicates the type of shift.

The shifter is made of 3 levels:

1) Level 1 for mask generation: starting from A and depending on type of shift, generates four masks (mask00 no shift, mask08 eight positions shift, mask16 sixteen positions shift and mask24 twenty four positions shift).
2) Level 2: with two higher bits of B (B [4:3]) selects one mask from the previous level
3) Level 3: with the remaining bits of B (B [2:0]) makes a fine grained shift, starting from the mask selected at previous level.

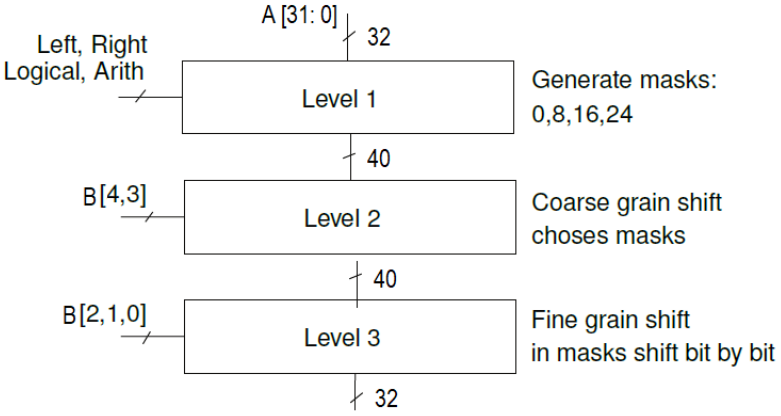


Figure 4.12 T2 Shifter

4.1.3 Logicals

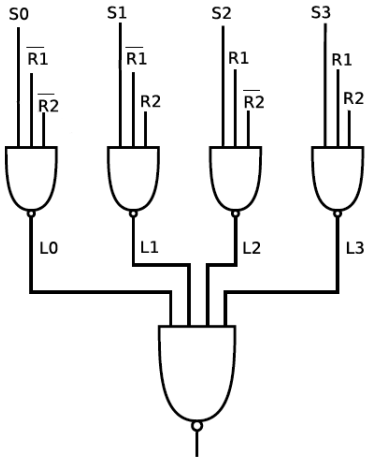
The component Logicals is the same as the logic unit inside Sparc T2, designed on 32 bits for DLX.
It is implemented with two NAND gate levels (each NAND has input on 32 bits) and is able to compute logical instructions: **AND NAND OR NOR XOR NXOR**

The first level is made of 4 NAND gates with 3 inputs: one for selection signal and two for operands.
The second is made of a single 4-inputs NAND: it receives the output signals from the upper level.
Instead of placing a single gate for each logic function (higher cost and bigger number of wires) we can obtain the same result with this design. Depending on the value of selection signals we can calculate a different operation:

- AND: select0=0, select1=0, select2=0, select3=1
- NAND: select0=1, select1=1, select2=1, select3=0
- OR: select0=0, select1=1, select2=1, select3=1
- NOR: select0=1, select1=0, select2=0, select3=0
- XOR: select0=0, select1=1, select2=1, select3=0
- XNOR: select0=1, select1=0, select2=0, select3=1

Note that for each first level NAND gate the operands can be a combination of $R1/\overline{R1}$ and $R2/\overline{R2}$

In figure 4.13 are shown the block scheme and the logic operations inside Logicals



$L0 = \overline{S0 \cdot R1 \cdot R2}$ $L1 = \overline{S1 \cdot R1 \cdot R2}$ $L2 = \overline{S2 \cdot R1 \cdot R2}$ $L3 = \overline{S3 \cdot R1 \cdot R2}$

$Logical\_Out = \overline{L0.L1.L2.L3} = \overline{L0.L1} + \overline{L2.L3} = \overline{L0} + \overline{L1} + \overline{L2} + \overline{L3}$

$Logical\_Out = S0.\overline{R1}.\overline{R2} + S1.\overline{R1}.R2 + S2.R1.\overline{R2} + S3.R1.R2$

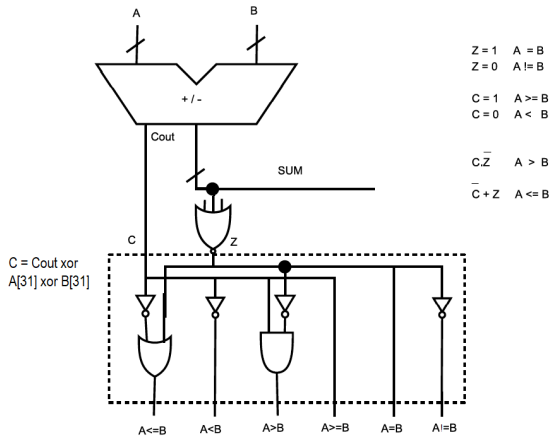
S0	S1	S2	S3
0	0	0	1
1	1	1	0
0	1	1	1
1	0	0	0
0	1	1	0
1	0	0	1

$R1.R2$	AND
$\overline{R1}.\overline{R2} + \overline{R1}.R2 + R1.\overline{R2} = \overline{R1.R2}$	NAND
$\overline{R1}.R2 + R1.\overline{R2} + R1.R2 = R1+R2$	OR
$\overline{R1}.\overline{R2} = \overline{R1+R2}$	NOR
$\overline{R1}.R2 + R1.\overline{R2} =$	XOR
$=$	XNOR

Figure 4.13 Block scheme and logic operations inside T2 Logicals

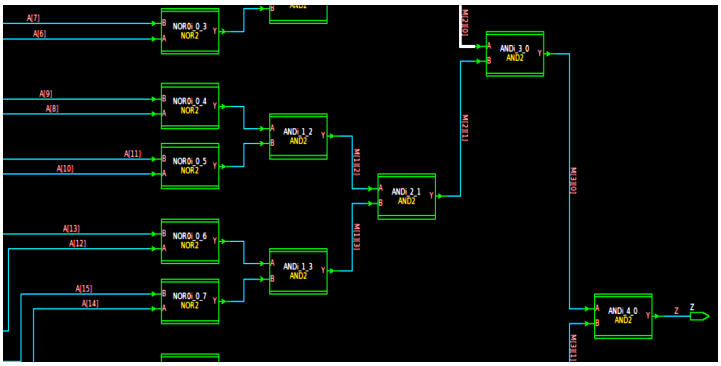
4.1.4 Comparator

Comparator is a component that receives the SUM (between operands A and B) from Adder and Carry Out and is able to detect if the two starting operands are equal, different, greater/greater equal, less/less equal. For each condition verified, the corresponding output bit is set to '1'.
The schematic is shown in Figure 4.14 (in this component, Adder is not present).
The Sum signal enters into a big NORN logic gate (implemented with a cascade of NOR and AND gates, visible in figure 4.15) and generates the output signal Z, which is used with C to verify if a certain relation between A and B is true.
C is obtained with the formula C = Cout xor (A[31] xor B[31]), this is used because otherwise the comparator does not work on signed integers.



$$
\begin{aligned}
A > B &\Rightarrow C \ and \ \overline{Z} \\
A \geq B &\Rightarrow C \\
A < B &\Rightarrow \overline{C} \\
A \leq B &\Rightarrow \overline{C} \ or \ Z \\
A = B &\Rightarrow Z \\
A \neq B &\Rightarrow \overline{Z}
\end{aligned}
$$

Figure 4.14 Comparator and Schematic of relationships



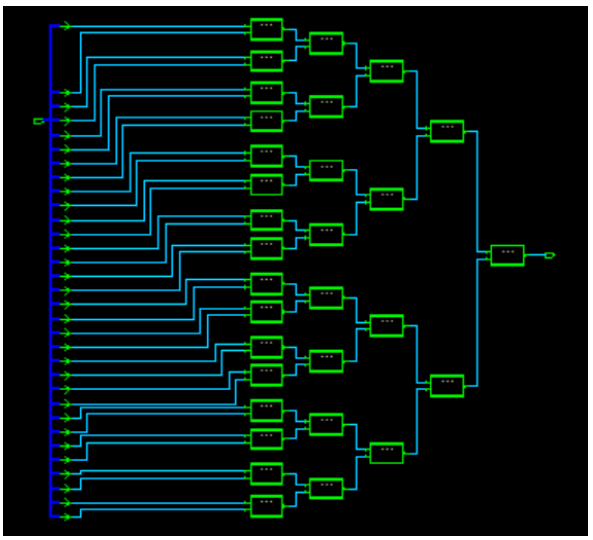


Figure 4.15 NORN schematic

4.2 Jump and Branch management

A jump/branch occurs when an instruction needs to change the normal flow of the program and move to another portion of code. The jump can be unconditional (when the instruction is present, jump always) or conditioned (jump only if a certain condition is verified).
A label is used inside code as a pointer: the next instruction to jump to, is the next one to the label.

The jump and branch management takes directly the output of the ALU instead of using the register output: in this way is possible to propagate the new address in the execution stage and avoid waiting for the memory stage (note that after the zero detector block there is no register).

The pipeline needs two NOP after a jump or branch instruction, because we need to wait two clock cycles to compute the address.

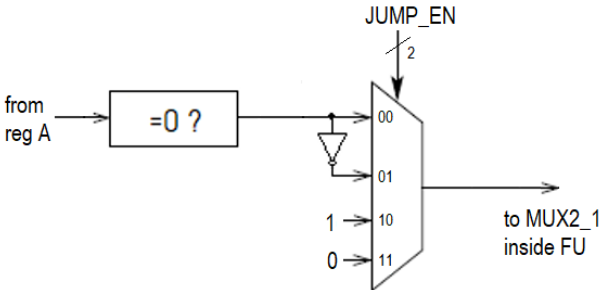


Figure 4.15 MUX 4_1

4.2.1 Mux4_1 selections

To manage jumps we need the control signal JUMP_EN, on two bits, provided by the control unit. This is the selection signal for the **MUX4_1**, shown in figure 4.15.

When the output is equal to '1', a jump is done. We need to remember that inside the fetch unit there is a MUX2_1 that selects between ALU_OUT and PC+4 values. When its selection signal (the output of MUX4_1) is equal to '1', the address of the next instruction to fetch is provided by the ALU (JUMP).

JUMP_EN is equal to "00" when a jump is not needed, so the value '0' is chosen and propagates to the output.
JUMP_EN is equal to "01" when there is a JUMP instruction, so the value '1' is chosen. This is done unconditionally, by J, JAL, JR and JALR instructions.

JUMP_EN is equal to "11" when there is a condition to verify before jumping. The condition is the following: if the content of register A (from the decode unit) is equal to zero, then jump. This is the case of BEQZ instructions, which jump only if the content of a certain register, loaded inside register A by Register File, is equal to zero.

Otherwise continue normally with the next instruction.

JUMP_EN is equal to "10" when there is another condition to verify before jumping: the content of register A must be different from zero. If true, then jump.
It's important to notice that this condition is the opposite of the previous one, for this reason an INVERTER gate is inserted, to complement the result.

4.2.2 Zero Detector

This component is used with the MUX4_1 to manage branches.
If the input value (on 32 bits) is equal to all 0's, the result is 1, 0 otherwise. The input arrives from register A of Decode Unit, while the output goes to the MUX4_1 previously described.
It's used to verify the branch condition of instruction `BEQZ Ri, label`. If the content of register `Ri` is zero, then result is 1, so a jump to `label` is performed.

To verify the branch condition of instruction `BNEZ Ri, label` it's enough to complement the output of Zero Comparator: if the input value is NOT zero, the result is 0, that becomes 1 going through an inverter (BNEZ in fact jumps only if the content of register `Ri` is different from zero).

To implement this component, we have chosen the T2 Zero Detector. The 32-bits input A is the signals arriving from register A of the Decode Unit.
Design consists of XNOR and AND gates cascaded:

- the first level is made of **XNOR** gates (between A[i] and '0')
- the remaining levels are made of **AND** gates, receiving, as inputs, output values from the previous levels.

The output value is a single bit Z, that is equal to '1' only if each single bit of A is 0.

In figure 4.16 and 4.17 are visible the top level view and hierarchical order of gates.

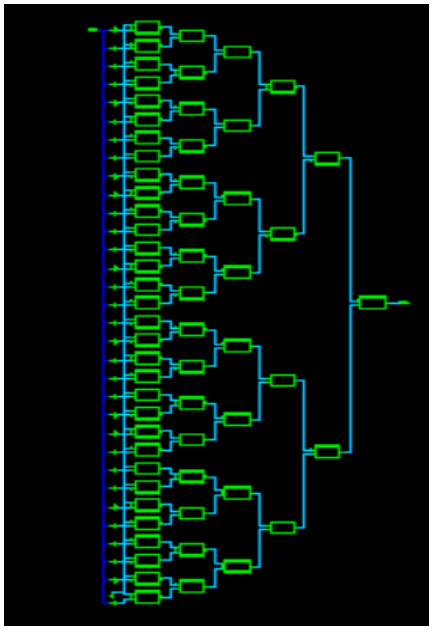


Figure 4.16 Zero Detector top level view

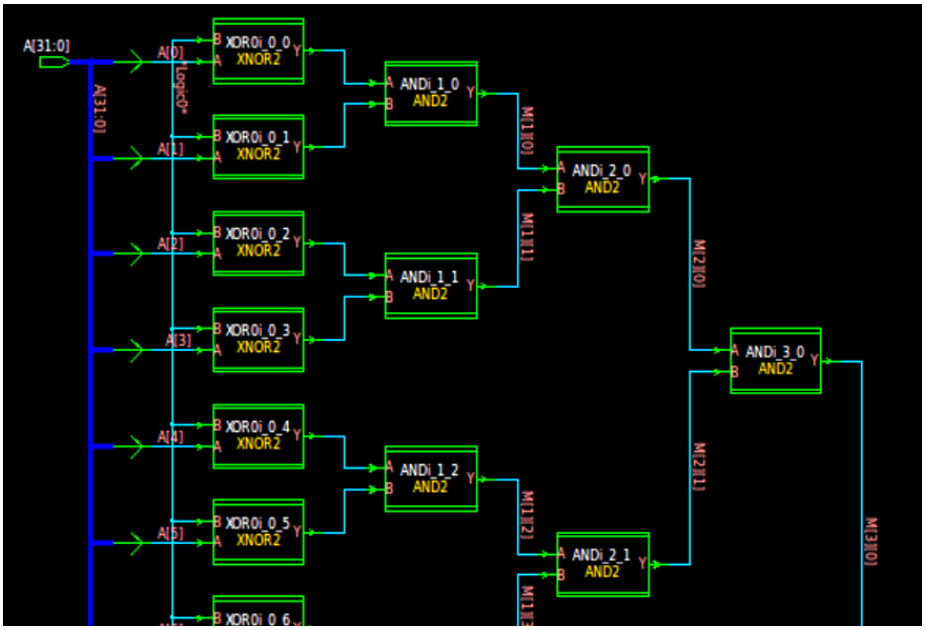


Figure 4.17 Hierarchical order of XNOR and AND gates

Chapter 5

Memory + Write Back Unit

Memory + Write Back Unit is the fourth stage of the pipeline. It's composed by:

1) Data Memory
2) two Multiplexer MUX2_1 (Figure 5.1).

This stage receives as input:
- the propagation of NPC, from NPC2 register
- the content of a general purpose-register, from Me (used here as data input for Data Memory)
- the output value of ALU, that can be used as an address for DATA MEMORY (in case of a load/store instruction) or bypassed and directly sent to the first MUX1.

MUX1 selects between Data Memory output value and ALU output, with the selection signal S3.
Mux2 selects between the output of MUX1 and the value of NPC, with the control signal S4.
The NPC's value is used for JAL/JALR instructions because they save the value of PC+4 inside the register R31 of the Register File. The output of MUX2, in fact, is used as data input for Register File, for writing (corresponding address is the content of Rd2 register).

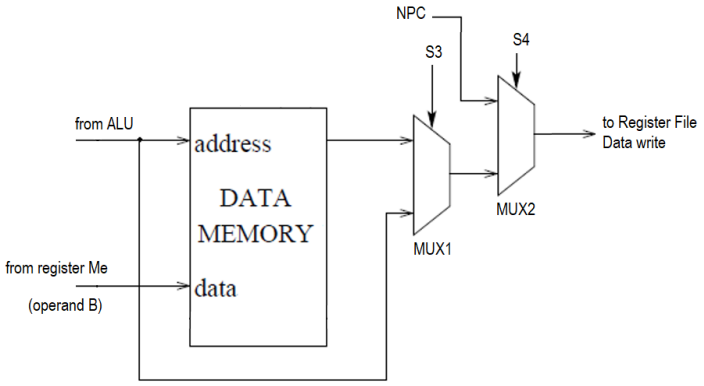


Figure 5.1: Fetch Unit

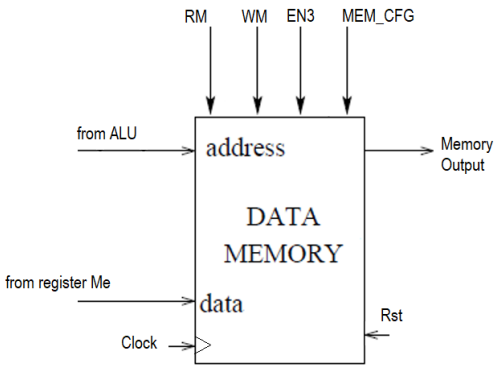


Figure 5.2: Data Memory and control signals

5.1 Data Memory

Data Memory is a RAM memory that contains data useful for load/store instructions.
Load instructions save the content of a register inside the Memory. Store instructions, instead, read data from Memory and save it to a general-purpose register.
Write and Read operations can differ depending on the type of load/store instruction, the memory is configurable to perform this different type of operations by the signal MEM_CFG, generated by the Control Unit.

5.1.1 Write and Read Processes Data Memory

The Address input port is unique for write and read operations (only one action at a time). The data input for writing arrives from register Me, that propagates the content of a general-purpose register, previously saved inside B register. Write operation starts when WM='1' and EN3='1' and is synchronous to the rising edge of the clock (in case of a store instruction).
Read operation is asynchronous and starts when RM='1' and EN3='1'. The address defines the content to be read (in case of a load instruction).

Chapter 6

Control Unit

The Control Unit is a very important component for the design. It generates the proper control words, depending on current instruction, in the proper clock cycle.
Inside the Control Unit there are the following components:

1) Look Up Table: contains all the control words
2) Finite State Machine: detects if a STALL is needed
3) MUX2_1 which selects between the current generated Control Word and the control signals to put a stall in the pipeline. (Figure 6.2)

The main input signal is the IR content, in particular the two fields OPCODE and FUNC which are used to specify the instruction to execute. They enter the LOOK-UP TABLE, act as addresses, in order to obtain the proper control word.
In figure 6.1 is visible the schematic of an Hardwired Control Unit, in which all the control signals exit from LUT and are correctly delayed (with registers), to stay synchronized with the stage of the pipelined Datapath. The names are the same as in the Datapath, with the same order.

FSM for RAW hazards:

There's a simple FSM able to detect if the current instruction (in the IF stage) is going to produce a RAW hazard during the ID stage.
As shown in the picture 6.1 the FSM reads the RD_ID (the destination register of the very previous instruction) and the RD_IE (the destination register of the instruction fetched 2 clock cycles before), if the current instruction is going to read from one of those registers (RS1, and RS2 are used for this check) then a stall signal is raised.
The control unit uses the stall signal just to select between the control word of the current instruction or the control word of a NOP instruction, in this way if stall signal is high then a NOP will be injected in the pipeline for 1 or 2 clock cycles.
The stall signal is then propagated to the fetch unit (FLUSH signal) in order to freeze the stage until RAW is solved!

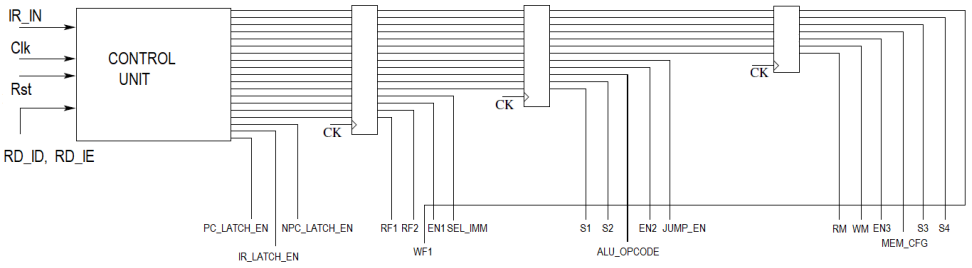


Figure 6.1 Hardwired Control Unit

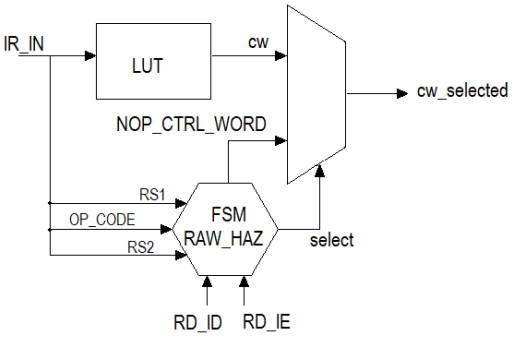


Figure 6.2 Control Unit

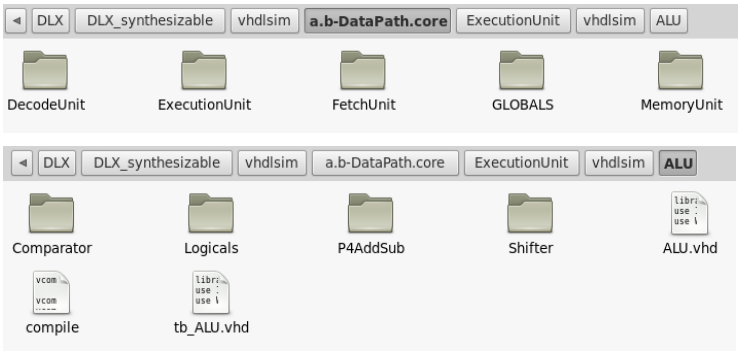
Chapter 7

File organization and scripts

The DLX has been designed using a bottom up approach so all 4 pipeline stages, their components and subcomponents, as well as for the control unit, are organized in separate directories and subdirectories.

With this file organization it becomes easier to to test and fix every part of the design as it gets more complicated.

Every layer contains a **compile** file that lists all the vhdl modules, coming from layers below, used by the layer's top level entity. In the figure there's an example of the organization for the DLX's ALU in the Execution stage.




```
vcom ../../../GLOBALS/constants.vhd

vcom ../../../GLOBALS/lfsr.vhd
vcom ../../../GLOBALS/iv.vhd
vcom ../../../GLOBALS/nd2.vhd
vcom ../../../GLOBALS/mux21.vhd
vcom ../../../GLOBALS/mux21_generic.vhd
vcom ./P4AddSub/FA.vhd
vcom ./P4AddSub/RCA.vhd
vcom ./P4AddSub/SUM_BLOCK.vhd
vcom ./P4AddSub/SUM_GENERATOR.vhd
vcom ./P4AddSub/g_block.vhd
vcom ./P4AddSub/pg_block.vhd
vcom ./P4AddSub/pg_net.vhd
vcom ./P4AddSub/xorGrid.vhd
vcom ./P4AddSub/carry_gen.vhd
vcom ./P4AddSub/P4_Adder.vhd

vcom ./Shifter/vhdlsim/ShifterT2.vhd

vcom ../../../GLOBALS/nd3.vhd
vcom ../../../GLOBALS/nd4.vhd
vcom ./Logicals/Logical.vhd

vcom ../../../GLOBALS/NOR2.vhd
vcom ../../../GLOBALS/AND2.vhd
vcom ../../../GLOBALS/NORN.vhd
vcom ../../../GLOBALS/MUX6_1Generic.vhd
vcom ./Comparator/vhdlsim/comparator.vhd
vcom ./Comparator/vhdlsim/topLevelCMP.vhd

vcom ../../../GLOBALS/MUX4_1Generic.vhd
vcom ./ALU.vhd


vcom tb_ALU.vhd
```

Figure A.6: Example file organization and compile file.

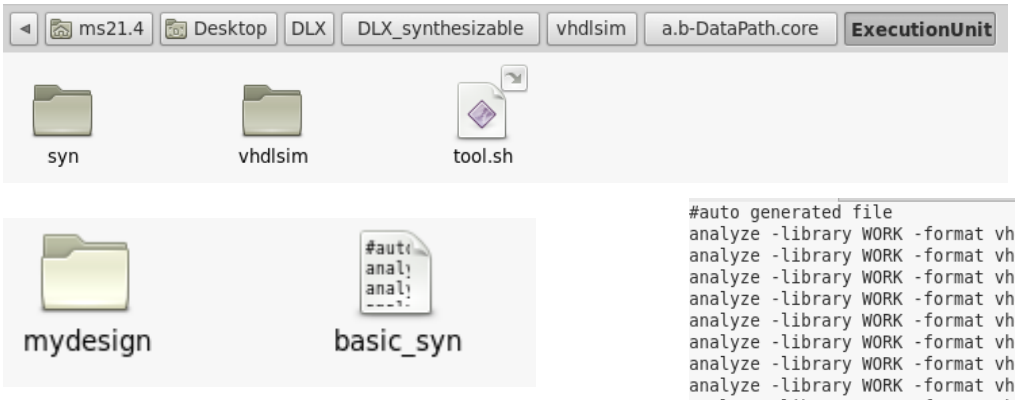
7.1 **tool.sh** script

This script is useful when, after you have designed and tested a component (like the ALU, for example) there's the need to perform a very basic synthesis, just to check if the design can be correctly synthesized or used as a starting point for a more advanced synthesis analysis.

The file organization discussed before is very important here because this script needs to be used within this type of organization.

The starting point for the script is the compile file in the vhdlsim directory, all the components listed in are copied in an appropriate directory (syn/mydesign) and then a basic synthesis script (basic_syn) is generated.

example: ./tool.sh $pathOfDesign $entityName




```
#auto generated file
analyze -library WORK -format vhdl {constants.vhd}
analyze -library WORK -format vhdl {iv.vhd}
analyze -library WORK -format vhdl {nd2.vhd}
analyze -library WORK -format vhdl {mux21.vhd}
analyze -library WORK -format vhdl {mux21_generic.vhd}
analyze -library WORK -format vhdl {FA.vhd}
analyze -library WORK -format vhdl {RCA.vhd}
analyze -library WORK -format vhdl {SUM_BLOCK.vhd}
analyze -library WORK -format vhdl {SUM_GENERATOR.vhd}
analyze -library WORK -format vhdl {g_block.vhd}
analyze -library WORK -format vhdl {pg_block.vhd}
analyze -library WORK -format vhdl {pg_net.vhd}
analyze -library WORK -format vhdl {xorGrid.vhd}
analyze -library WORK -format vhdl {carry_gen.vhd}
analyze -library WORK -format vhdl {P4_Adder.vhd}
analyze -library WORK -format vhdl {ShifterT2.vhd}
analyze -library WORK -format vhdl {nd3.vhd}
analyze -library WORK -format vhdl {nd4.vhd}
analyze -library WORK -format vhdl {Logical.vhd}
analyze -library WORK -format vhdl {NOR2.vhd}
analyze -library WORK -format vhdl {AND2.vhd}
analyze -library WORK -format vhdl {NORN.vhd}
analyze -library WORK -format vhdl {MUX6_1Generic.vhd}
analyze -library WORK -format vhdl {comparator.vhd}
analyze -library WORK -format vhdl {topLevelCMP.vhd}
analyze -library WORK -format vhdl {MUX4_1Generic.vhd}
analyze -library WORK -format vhdl {Mux4_1.vhd}
analyze -library WORK -format vhdl {ALU.vhd}
analyze -library WORK -format vhdl {XNOR2.vhd}
analyze -library WORK -format vhdl {zerodetect.vhd}
analyze -library WORK -format vhdl {ffd.vhd}
analyze -library WORK -format vhdl {RegisterN.vhd}
analyze -library WORK -format vhdl {TopLevelEXU.vhd}
elaborate EXUNIT -library WORK
compile -exact_map
report_power > power_report.txt
report_area > area_report.txt
report_timing > timing_report.txt
write -hierarchy -format verilog -output syn_netlist.v
```

Figure A.7: when tool.sh is executed in ExecutionUnit

7.2 Synthesis script: **synthesis.tcl**

This synthesis script has been used for the optimization of the entire DLX, it performs a synthesis under timing constraints both on the datapath + control unit.

In order to reduce the power consumption of the registers, we used a clock gating technique to all latches in the design.

The starting point is the synthesis script generated by the tool.sh script, in this way a "pre-synthesis" is performed before the real one.

Chapter 8

Physical Design

Last step after the synthesis process is the physical design using Innovus by Cadence CAD.
Starting from the post synthesis netlist, it is possible to process the physical standard cell layout of the DLX.
These are all the steps to be done:

1) Configuring
2) Floorplanning
3) Power planning and routing
4) Cell placing
5) Signal routing
6) Post routing optimization
7) Analysis for timing and integrity

Figures 8.1, 8.2: post routing design analysis, Connectivity and Geometry verifications

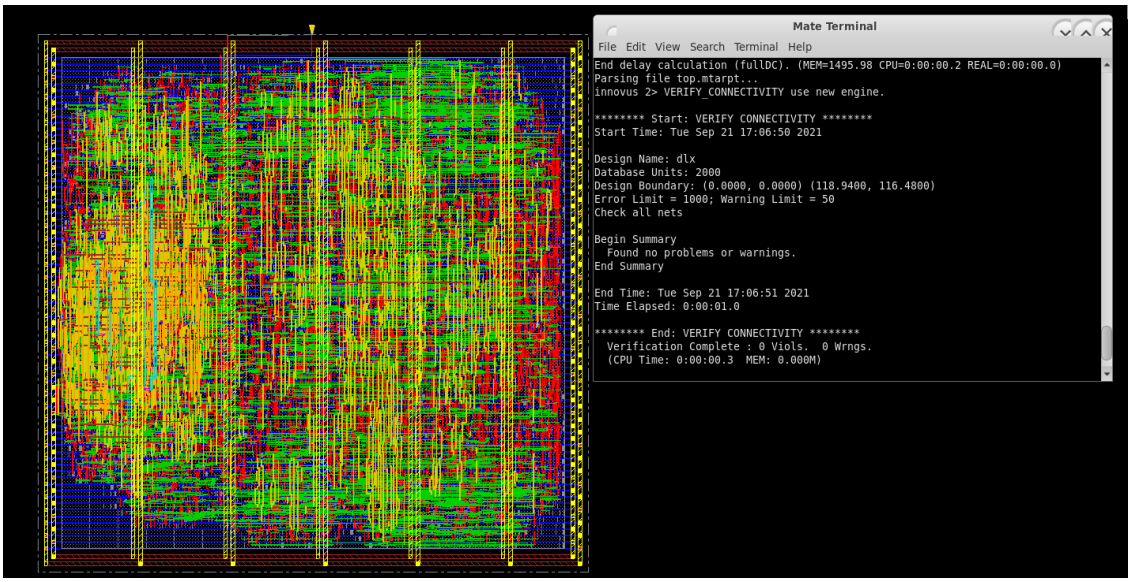


Figure 8.1: Connectivity Verification result

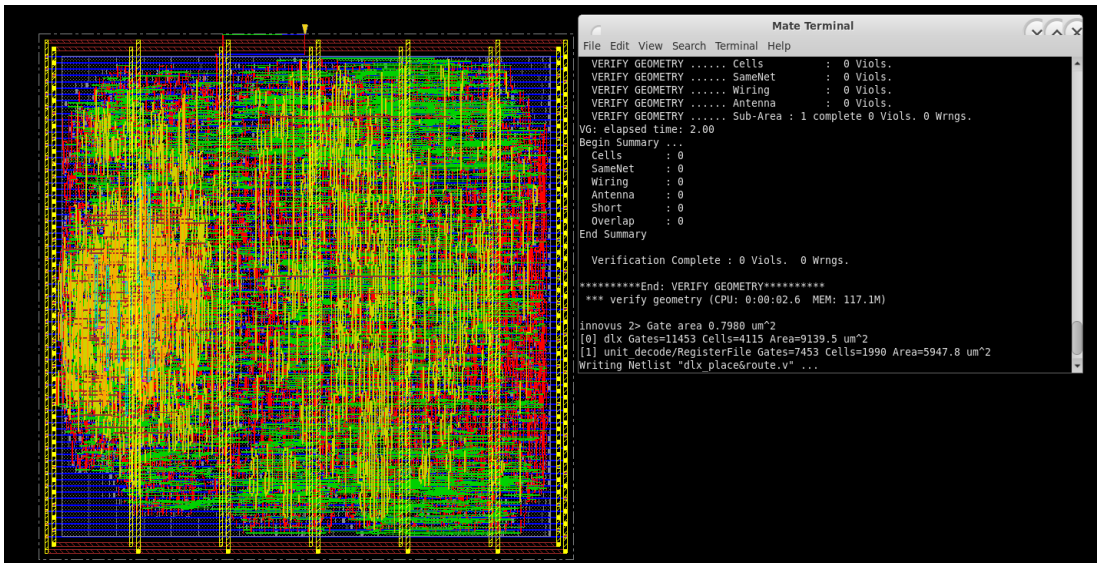


Figure 8.2: Geometry verification result

Conclusion

This project is the very starting point for a real RISC processor, these are some of the further improvements that can be done in future:

1) Complete Hazard detection unit
2) Windowed Register File for procedures and MMU
3) Floating Point Unit
4) Multi cycle operations like DIV and MUL
5) Branch prediction units like BHT

APPENDIX A

Basic and frequently used components. They can be found inside the directory GLOBALS.

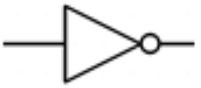
Nand2: simple NAND logic gate, computes `Y <= not (A and B)`

And2: Simple AND logic gate, computes `Y <= A and B`

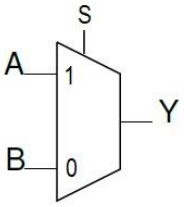
Nor2: Simple NOR logic gate, computes `Y <= not (A or B)`

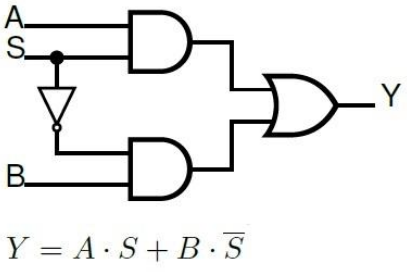
Xnor2: Simple XOR logic gate, computes `Y <= not (A xor B)`

Inv: Simple INVERTER logic gate, computes `Y <= not(A)`



Multiplexer MUX2_1:
is a Multiplexer that selects between two inputs, each on one single bit. Its implemented with the following formula `Y = (A and SEL) or (B and (not(SEL)))`





$$Y = A \cdot S + B \cdot \overline{S}$$

Figure A.8 Mux2_1 schematic

Multiplexer MUX2_1_generic:
is a Multiplexer that selects between two inputs, each on N bits. It is made with N multiplexers MUX2_1 and has the same behavior.

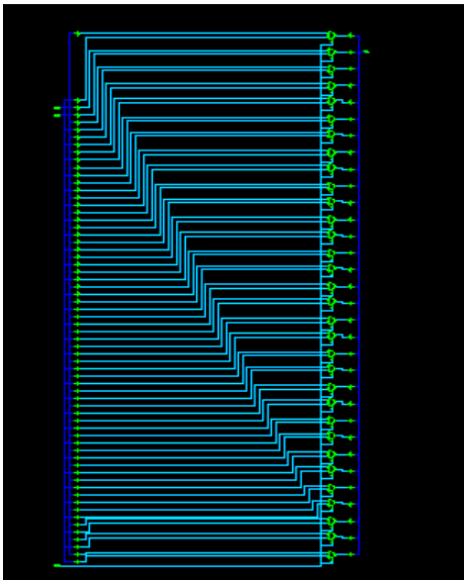


Figure A.9: MUX2_1 Generic on N=32 bits

Ffd:
Is a flip-flop of D type, with latch enable signal. It is synchronous to the rising edge of the clock. When the Reset signal is equal to ‘1’, the output Q is set to the value ‘0’. Otherwise, if the Enable signal is active (high), the input signal D is propagated to the output Q, when there is a positive edge of the clock signal.

RegisterN:
Is the most used component inside all the design (for all pipeline registers named previously).
It is composed of N Flip Flop D, so has the same behavior of the Ffd (synchronous to rising edge clock, latch enabled).

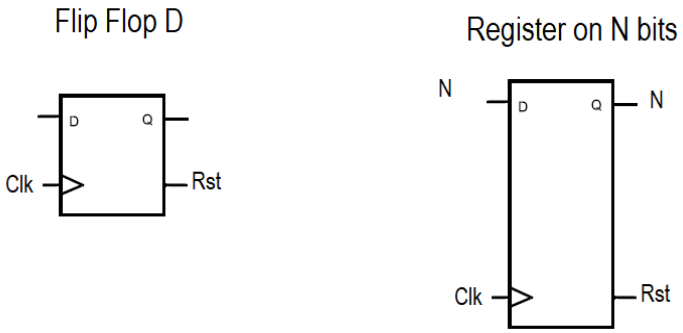


Figure A.10: Flip Flop D and RegisterN