Politecnico di Torino

Microelectronic Systems Final Project

DLX pro project

Student:
Marco Meloni
269145

Professor.
Dr. Mariagrazia Graziano

Contents

Chapter 1

Introduction

In this project I implemented a processor based on the DLX architecture, the processor design followed three phases:

- VHDL design and simulation
- Synthesis with frequency optimization
- Physical design with routing optimization

This report will go over each one of them.

1.1 General architecture of the processor

The processor is based on the DLX architecture, and thus is a pipelined processor with a 5-stage pipeline with load and store instructions.

To make the pipeline work the datapath has blocks of registers inbetween each stage (the PC register is a synchronous register as well), while all the components inside each stage are combinational.

1.2 Top level design

The CPU is composed of the Datapath and the Control Unit that interact with two memories, the Instruction Memory (IRAM) and the Data RAM (DRAM).

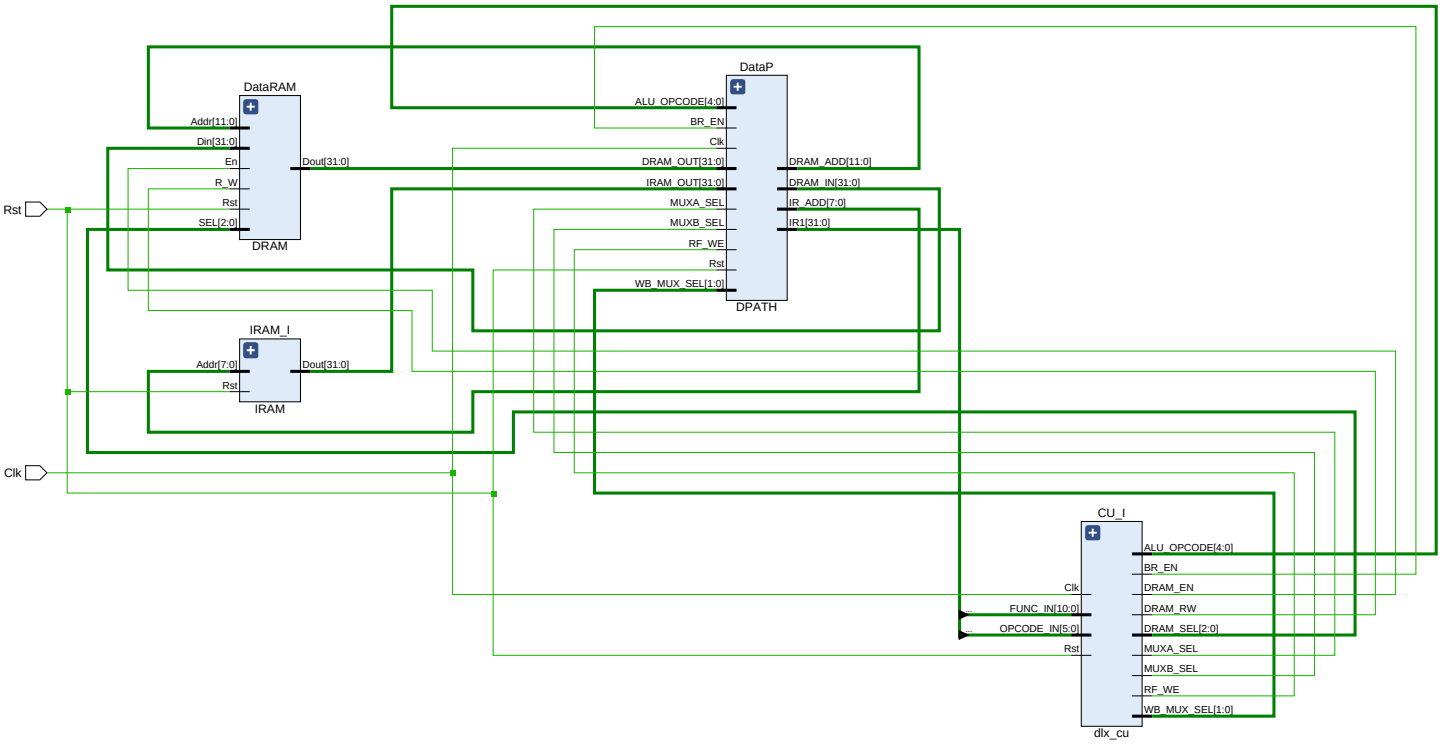


Figure 1.1: DLX structure

1.3 Instruction set

In the DLX we have three types of instructions:

- R-type instructions (Figure 1.2)
- I-type instructions (Figure 1.3)
- J-type instructions (Figure 1.4)

All instructions are on 32 bits and I-type and J-type instructions have a unique OPCODE for every instruction, while the OPCODE of R-type instructions is the same for all of them and they are differentiated by their FUNC field.

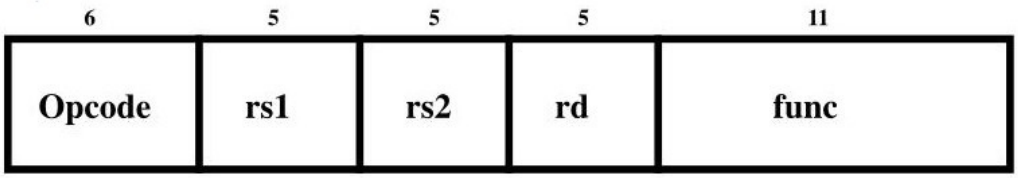


Figure 1.2: R-type instruction. Rs1 and Rs2 are the addresses of the source registers, Rd is the address of the destination register. The func field is used to determine which kind of operation will be done, so it's used to choose which control signals send to the ALU

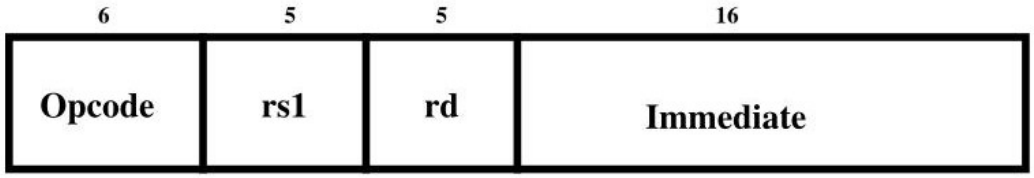


Figure 1.3: R-type instruction. Rs1 is the address of the source registers, Rd is the address of the destination register. The immediate field will be extended to 32 bits as signed or unsigned depending on the instruction

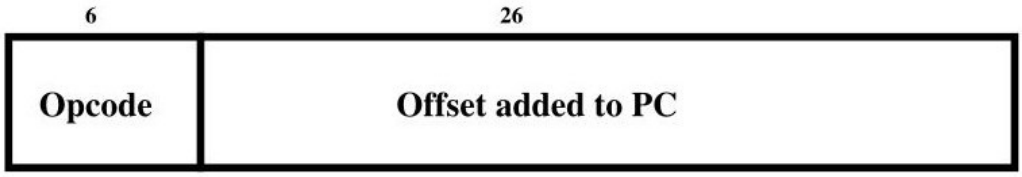


Figure 1.4: The 26 bit offset will be extended to a 32 bit signed number

This particular implementation supports this set of instructions:

- nop
- j
- jal
- jr
- jalr
- beqz
- bnez
- add
- addi
- addu
- addui
- sub
- subi
- subu
- subui
- and
- andi
- or
- ori
- xor
- xori
- sll
- slli
- srl
- srli
- sra
- srai
- seq
- seqi
- sne
- snei
- slt
- slti
- sltu
- sltui
- sgt
- sgti
- sgtu
- sgtui
- sle
- slei
- sleu
- sleui
- sge
- sgei
- sgeu
- sgeui
- lb
- lbu
- lh
- lhu
- lw
- sb
- sh
- sw

For a total of 55 instructions, 28 more than the 27 required for the basic version.

Chapter 2

CPU design

2.1 Control Unit

The Control Unit is a Hardwired control unit. It has a microcode memory inside of it containing all the control words for each possible instruction in out ISA. We access the microcode memory by using the OPCODE of the instruction as the address.

The control word is then inserted in a pipeline used to deliver the right signals for that clock cycle for each stage.

The control signals for the ALU have a pipeline of their own, and the signals that get inserted in it are decided depending on the OPCODE of I-type and J-type instructions or the FUNC field in the case of R-type instructions.

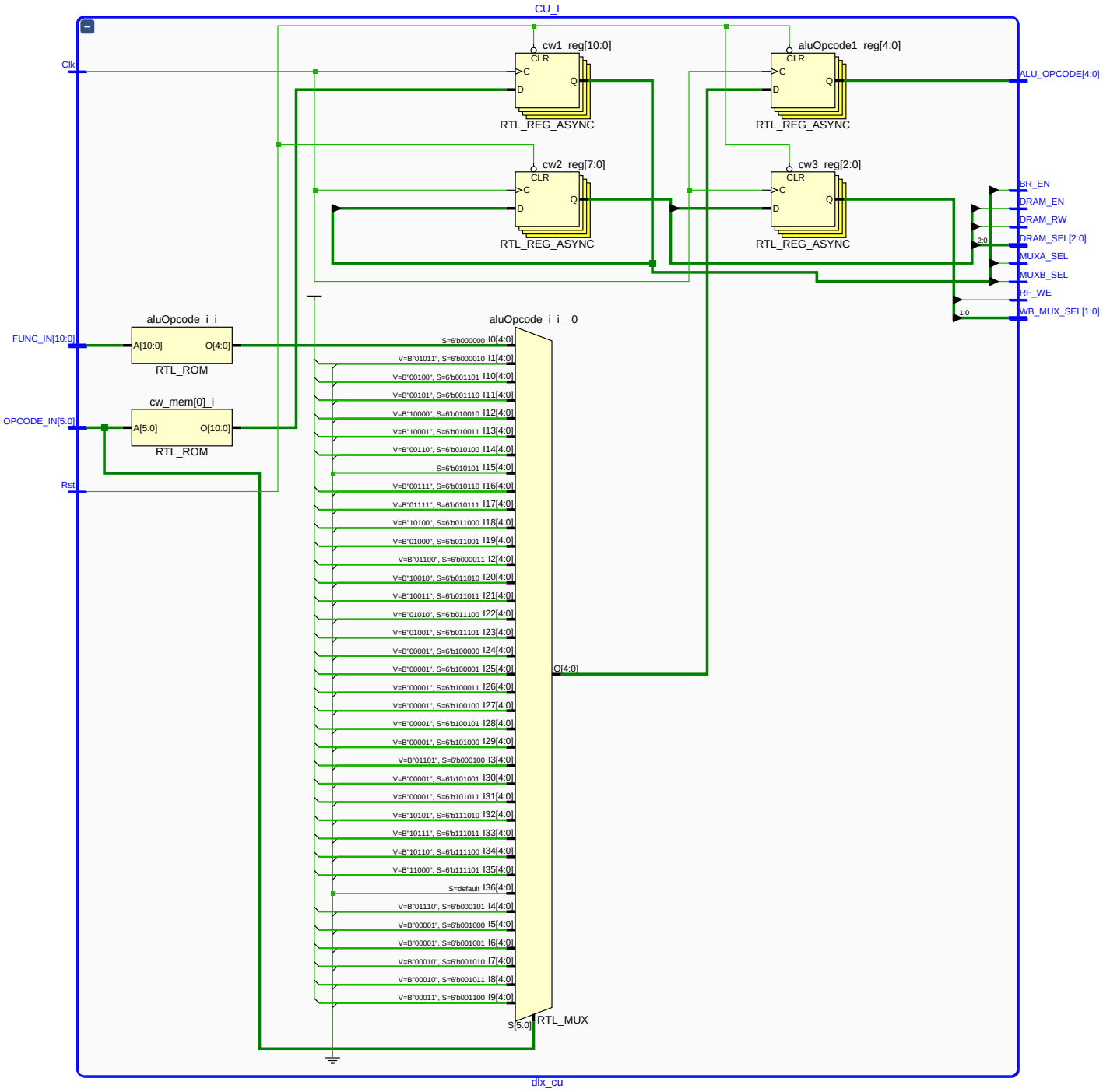


Figure 2.1: Schematic of the Control Unit

2.2 Instruction Memory

The instruction Memory (IRAM) in this design has a total of 256 entries by 32 bits each. It is a simple memory with asynchronous read.

At reset (active low) it fills up with the instructions written on a mem-file we generate by compiling an assembly program.

The instructions loaded on the IRAM are then accessed by using the value of the PC register as the address.

2.3 Data Ram

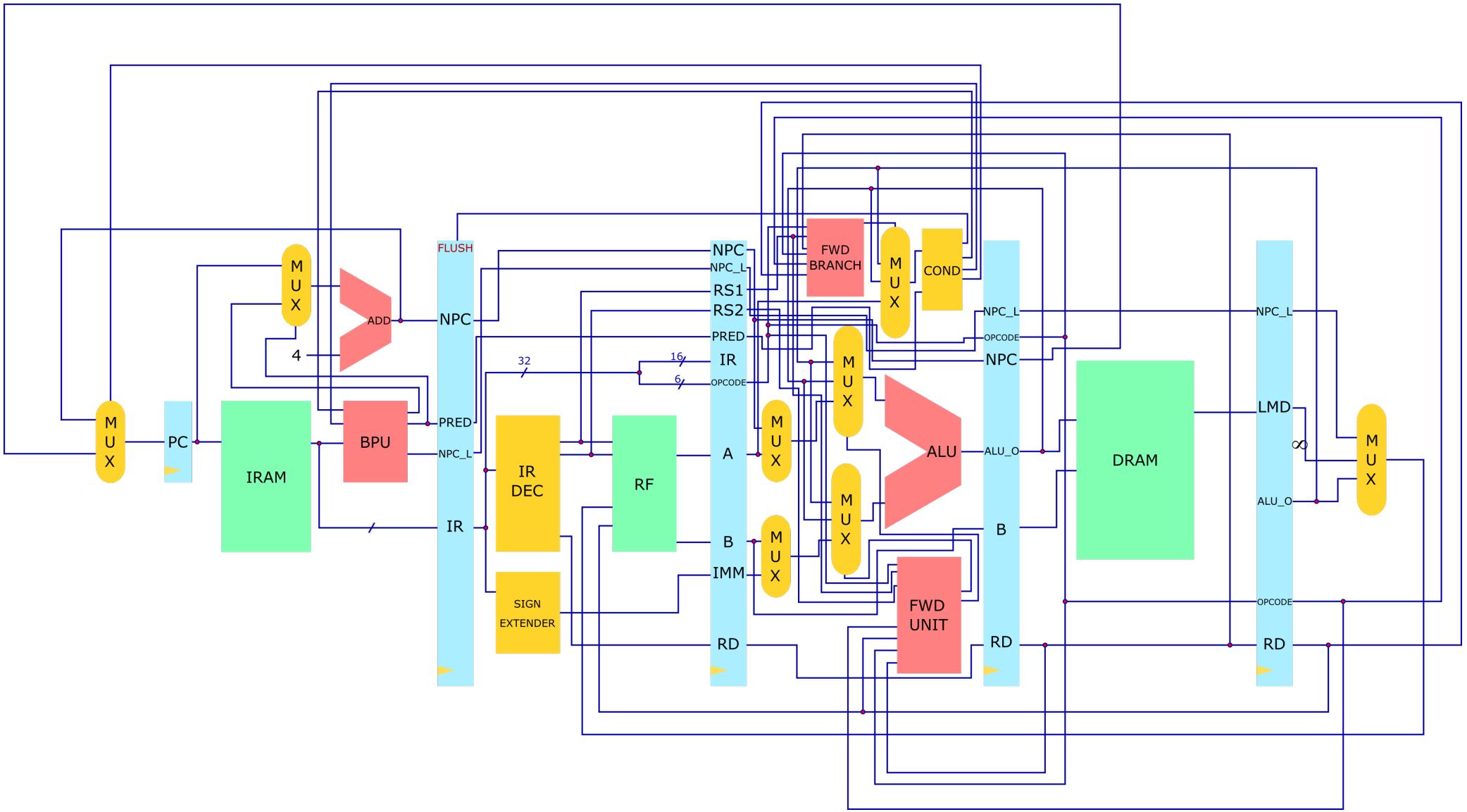
The Data Ram (DRAM) is implemented as a 4kbyte memory with single byte access.

In the DRAM block there's also some logic used to implement both the storing and loading of data in big-endian mode and the support of access to bytes and half-words.

2.4 Datapath

The Datapath is divided into 5 stages. They are:

- Istruction Fetch stage (IF), where the CPU loads the instruction pointed by the Program Counter (PC) from the instruction memory.
- Instruction Decode stage (ID), where the instruction gets decoded, forwarded to the Control Unit to get the appropriate control signals and the values from the required registers are loaded from the Register File.
- Execution stage (EXE), where the data is processed by the ALU and we also evaluate the branch instructions.
- Memory access stage (MEM), where the processor accesses the DRAM, for load or store instructions.
- Writeback stage (WB), where the result from the instruction gets written in the Register File.


MUX
PC
IRAM
BPU
MUX
4
ADD
FLUSH
NPC
PRED
NPC_L
IR
32
IR DEC
SIGN EXTENDER
RF
16
6
NPC
NPC_L
RS1
RS2
PRED
IR
OPCODE
A
B
IMM
RD
FWD BRANCH
MUX
COND
MUX
MUX
MUX
MUX
ALU
FWD UNIT
NPC_L
OPCODE
NPC
ALU_O
B
RD
DRAM
NPC_L
LMD
ALU_O
OPCODE
RD
MUX

2.4.1 Instruction Fetch stage

Inside this stage are the Program Counter (PC), which is just a simple synchronous register, an adder to compute the Next PC value (NPC) and the Branch Prediction Unit (BPU). The adder for the NPC is a simple adder that just increases by 4 the value of its input, until it reaches a max value corresponding to the last addressable value of the IRAM.

2.4.1.1 BPU

The Branch Prediction Unit uses a BHT with a 2-bit FSM system for dynamic branch prediction. I decided to have a BHT with 32 entries, by supposing that around 20% of all instructions (on average) are jump instructions and that around 80% of those are conditional jumps. This means that around 16% of all instructions would be conditional jumps, and thus I decided that 32 entries was a reasonable size relatively to the maximum number of instructions in my architecture (which is 256, the number of words storable in the IRAM).

In the case of non-conditional jumps (J and JAL) they are obviously always taken and the BHT is not used in that case. The BPU does **NOT** support JR and JALR instructions, those still need to go through the ID and reach the EXE stage, so there's always a penalty for using them.

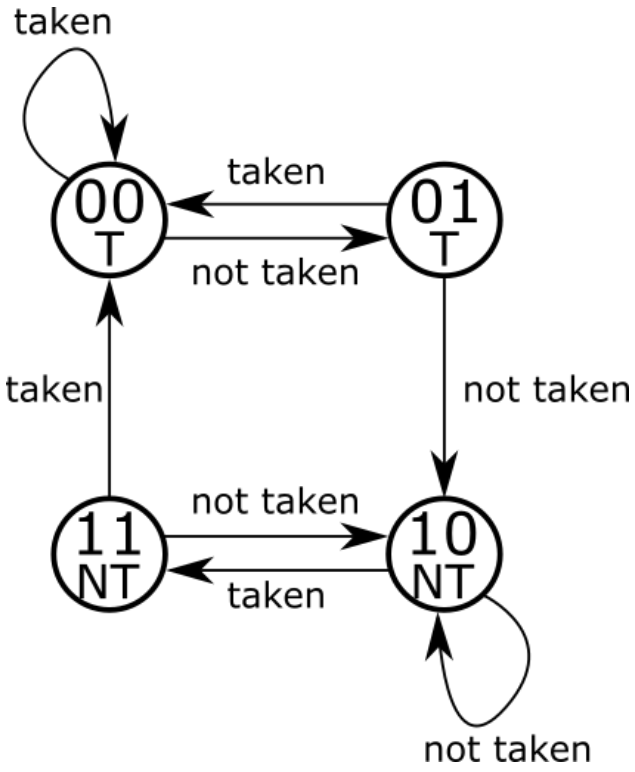
The BPU, depending on the prediction, controls a multiplexer put in front of the input of the NPC adder; if the branch is taken, the input is switched to the one output from the BPU, if it is not taken it lets the current PC pass.

The BPU also outputs the PC of the instruction it's reading and inserts it in the pipeline, so that when the branch gets actually evaluated in the EXE stage it will be possible to rollback if the prediction turned out to be incorrect. It's also necessary to update the relative entry in the BHT.
The BPU knows if the prediction was right or wrong thanks to two input signals coming from the branch condition evaluation in the EXE stage.

The logic governing the use of the BHT works like this: the BPU takes the bits 2 to 7 of the PC as address for the table (the 2 LSBs are ignored since they will always be 0). When a branch instruction is received, it reads

the relative value inside the BHT and makes a prediction based on that.
At reset all the entries of the BHT are set to **"00"**, which will be interpreted as a "Taken" prediction. I decided to make "Taken" the standard prediction based on the results from several branch prediction benchmarks using the SPEC95 benchmark suite[1]. These benchmarks confirm the already popular notion that "always taken" static branch prediction in significantly preferable to "always not taken", and so I decided it was reasonable to choose to take the branch as the standard initial behaviour.

The state machine for prediction works as illustrated here:



T means that if the BHT entry for the branch instruction that's being evaluated is in that state, the BPU will predict taken, while **NT** means it will predict not taken. The transitions depend on the result of the evaluation done in the EXE stage.

[1] http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/

2.4.2 Instruction Decode stage

Inside this stage the CPU decodes the instruction and gets the required data (if necessary) from the Register File. There are three components in this stage:

- **Instruction Decoder**: this component reads the instruction word and outputs the addresses for Rs1, Rs2 and Rd (source register 1, source register 2 and destination register respectively). Rs1 and Rs2 are an input to the RF while Rd is inserted in the pipeline.
- **Immediate Extender**: this component extends to 32 bits the immediate value required by the instructions that use them. Depending on the kind of instruction it will do an unsigned or signed extension, the input immediate can be either 16 or 26 bits long.
- **Register File**: it's a simple memory with asynchronous read and write containing 32 entries by 32 bits each. These are the 32 registers of the DLX processor.
  The write operation is only allowed when the Read Enable signal is raised, the write address is the previously pipelined Rd arrived to the WB stage. The register R0 is a read-only register always containing 0.

2.4.3 Execution stage

Inside this stage is where the computation of data actually happens. This is mostly the job of the Arithmetic Logic Unit (ALU). There's also the evaluation of the conditional jumps. A Forwarding Unit is present to allow the forwarding for arithmetic operations and branch evaluation, getting rid of most Read After Write (RAW) data hazards.

To allow forwarding and to switch to PC or immediate value for the operators there are several multiplexers in this stage, some controlled directly by the CU and some by the Forwarding Unit.

2.4.3.1 ALU

The ALU has 4 main components inside of it:

- **Adder**: here I used the Pentium 4 adder written for the labs, wrapped inside an adder component that inverts the sign of the second operand if a subtraction is required. It does operations on 32 bits.
- **Comparator**: simple behavioural comparator for 32 bits inputs.
- **Shifter**: generic behavioural shifter able to do shifts in both directions, bot logical and arithmetic.
- **Logic Unit**: component that handles the required logic operations, in this case AND, OR and XOR for both R-type and I-type instructions.

All these components' outputs are then input in a multiplexer that sets the required one as the output for the ALU depending on the ALUCODE received from the CU.

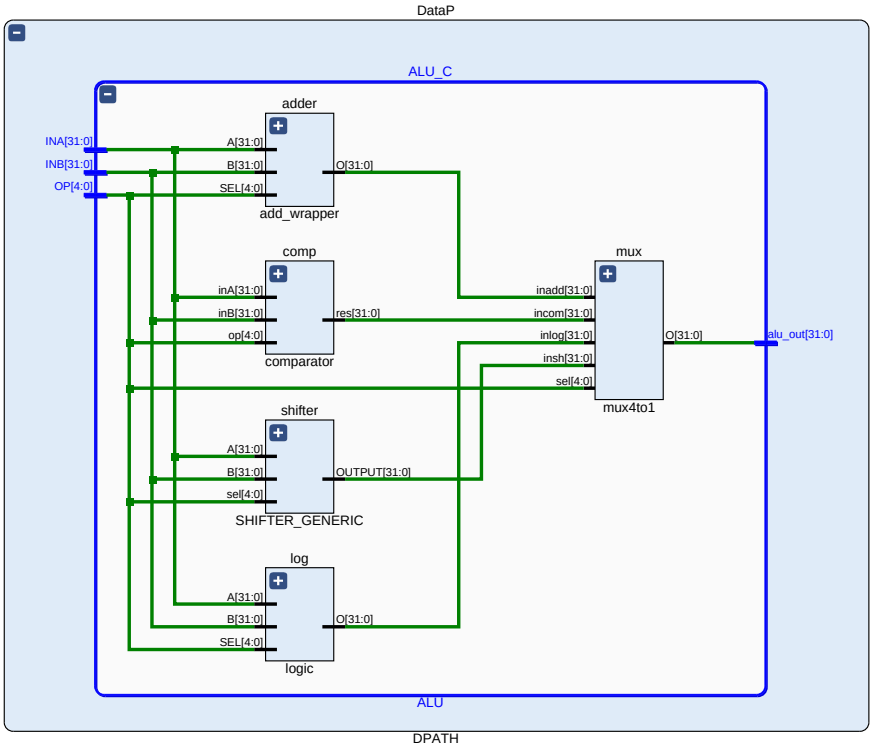


Figure 2.2: Schematic of the Arithmetic Logic Unit

2.4.3.2 Forwarding Unit

There are actually two separate Forwarding Units in this processor, one for arithmetic/logical operations and one for branching, but their behaviour and structure is similar, so here I'll just describe the former for brevity.

The FU receives as input the current OPCODE, the OPCODEs of the instructions in the MEM and WB stage, the Rs1 and Rs2 of the current operation and the Rd of both the instructions in the MEM and WB stage. By using this data it checks whether one of the required registers for the current operation was the destination register for one of the previous 2 instructions. It also checks if the previous and the current instructions are compatible operations. This is done by checking if they are either a R-type operation or one of the compatible I-type operations listed in a small CAM inside the unit.

If there is a match for both requirements then the ALU output from the previous instructions (which is being propagated through the pipeline) is set as the input for the ALU instead of the value read from the RF. This is done by putting a multiplexer in front of the inputs of the ALU. In case there is a match for the same register for both the instruction in the MEM stage and in the WB stage, the one in MEM stage is the one that will be forwarded since it's the most recent one.

The Forwarding Unit does **NOT** support store and load operations, which are still susceptible to RAW hazards.

2.4.3.3 Branch Evaluator

This block checks the condition for a branch to determine if the branch should have been taken or not and compares it with the prediction made by the BPU in the IF stage. If the prediction matches the evaluation, the block just sends a signal to the BPU to signal that the evaluation was in fact right, so that the BHT can be updated accordingly; if the prediction was wrong it sends a different signal to the BPU relative to that event and also flushes the pipeline, inserting two NOP instructions in place of the ones that were being wrongly loaded.

The block also flushes the pipeline when a JR or JALR instruction is detected.

The inputs of this block go through a multiplexer driven by the forwarding unit for branches, so that we avoid the RAW hazard of evaluating an old value for the register; if the register checked for the condition was the destination register of one of the 2 previous instructions, its new future value will be forwarded to the Branch Evaluator.

2.4.4 Memory access stage

This stage exists solely to allow writing and reading to/from the DRAM in case of load and store operations, the only component in this stage is the DRAM block itself, interacting with the memory through its internal logic previously described.

2.4.5 Writeback

In this stage the only component present is a multiplexer to drive the input of the write port of the RF. Depending on the selection signal coming from the CU it will let through the NPC value (for JAL and JALR instructions), the LMD (for load instructions) or the ALU output (for arithmetic/logical operations). All these values are the ones coming out of the MEM/WB registers. The address of the register where the data will be written also comes out of the Rd register at this stage.

Chapter 3

Synthesis

The goal of the synthesis was to optimize frequency and, if possible, power as a second objective. To achieve that I tried several different approaches and saved and analysed the 4 most significant ones. For the place and route part I chose only one of them.

The synthesis had to be done on just the Datapath and Control Unit, not the memories, so for that purpose I created an entity containing the DP and CU that connects them together and to the external memories. This was the top entity used for synthesis.

Since the synthesis software doesn't always produce the same result for the same run, when optimizing for each small improvement in the clock period (going by 0.01ns steps) I did several runs if I couldn't get a positive slack, and decided it was not a feasible frequency after 5 failed runs in a row. The following results are the best ones I managed to achieve using the aforementioned method.

3.1 600MHZ, no ungroup, no clock gating

For the first instance I wanted to see how much I could push the frequency without ungrouping the design. I managed to get to a clock period of 1.67ns before incurring in slack violations. This translates to an operating frequency of around 600MHz.

To synthesize my design I used the command *compile* with the option *-map_effort high*. The results for timing, power and area can be seen in the reports:

```
clock CLK (rise edge)                                    1.67       1.67
clock network delay (ideal)                              0.00       1.67
DataP/PC_reg/O_reg[0]/CK (DFF_X1)                        0.00       1.67 r
library setup time                                      -0.04       1.63
data required time                                                  1.63
--------------------------------------------------------------------------
data required time                                                  1.63
data arrival time                                                  -1.63
--------------------------------------------------------------------------
slack (MET)                                                         0.00
```

Figure 3.1: Worst path timing result

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
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       2.5765e+03          64.3970         4.9546e+04         2.6904e+03  (  69.42%)
sequential       189.0432           6.3770         4.7708e+04           243.1281  (   6.27%)
combinational    314.7234         458.0812         1.6917e+05           941.9764  (  24.31%)
--------------------------------------------------------------------------------------------------
Total          3.0802e+03 uW       528.8552 uW     2.6642e+05 nW     3.8755e+03 uW
```

Figure 3.2: Power report, total power of 3.88mW

```
Number of ports:                          123
Number of nets:                           151
Number of cells:                            2
Number of combinational cells:              0
Number of sequential cells:                 0
Number of macros:                           0
Number of buf/inv:                          0
Number of references:                       2

Combinational area:       7298.508007
Noncombinational area:    6443.318186
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          13741.826192
Total area:                 undefined
```

Figure 3.3: Total area report

3.2 600MHZ, no ungroup, clock gating

After the previous synthesis, I tried to add clock gating as a way to reduce the switching power consumption, hoping it would bring the total power consumption down. To do that I added to the command used previously the option *-gate_clock*. Unfortunately, I wasn't able to make the clock gated design respect the slack requirements, even after several synthesis runs. In the best run it barely violated the slack, but I still decided to count it as a failed try. Even if I didn't, the other reports show that, while the area was slightly reduced, the total power consumption would have actually slightly increased, meaning that probably gate clocking was not a viable strategy for this design.

```
clock CLK (rise edge)                                    1.67       1.67
clock network delay (ideal)                              0.00       1.67
DataP/PC_reg/O_reg[29]/CK (DFF_X1)                       0.00       1.67 r
library setup time                                      -0.04       1.63
data required time                                                  1.63
--------------------------------------------------------------------------
data required time                                                  1.63
data arrival time                                                  -1.63
--------------------------------------------------------------------------
slack (VIOLATED: increase signficant digits)                        0.00
```

Figure 3.4: Worst path timing result

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
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      8.1021          40.4267           55.7675            48.5846  (   1.24%)
register       2.5743e+03          64.1613       4.9508e+04         2.6879e+03  (  68.62%)
sequential       189.0385           6.3981       4.7708e+04           243.1445  (   6.21%)
combinational    313.9861         455.5023       1.6801e+05           937.5034  (  23.93%)
--------------------------------------------------------------------------------------------------
Total          3.0854e+03 uW      566.4884 uW    2.6529e+05 nW     3.9172e+03 uW
```

Figure 3.5: Power report, total power of 3.92mW

```
Number of ports:                          123
Number of nets:                           151
Number of cells:                            2
Number of combinational cells:              0
Number of sequential cells:                 0
Number of macros:                           0
Number of buf/inv:                          0
Number of references:                       2

Combinational area:       7238.126006
Noncombinational area:    6446.776186
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          13684.902192
Total area:                 undefined
```

Figure 3.6: Total area report

3.3 730MHZ, ungroup, no clock gating, advanced timing script

After the previous results, I decided to try to ungroup the design and synthesize the design this way. This choice would completely change the structure of the original design, making it more difficult to analyze and read, but I decided it would be an acceptable trade-off if the frequency increase was significant enough. To do this I used the *compile_ultra* command, which has auto-ungroup by default, along with the *-timing_high_effort_script* option, a script that makes use of extra optimization strategies to improve the delay of the circuit, at the cost of the time needed to perform the synthesis. This made the average run-time go from an average of 3 minutes per run (for the 600MHz attempts) to around 6-7 minutes, more than double but still absolutely acceptable.

These changes made me able to push the clock period down to 1.37ns, achieving a frequency of around 730MHz, an increase of over 21%. This was a huge improvement and so the strategy proved to be successful.

There was an increase of around 13% in power consumption, but since the main goal was to increase the frequency this was considered acceptable. The area is also reduced, as expected from the ungrouping.

```
clock CLK (rise edge)                                   1.37       1.37
clock network delay (ideal)                             0.00       1.37
DataP/PC_reg/O_reg[17]/CK (DFF_X1)                      0.00       1.37 r
library setup time                                     -0.04       1.33
data required time                                                 1.33
--------------------------------------------------------------------------
data required time                                                 1.33
data arrival time                                                 -1.33
--------------------------------------------------------------------------
slack (MET)                                                        0.00
```

Figure 3.7: Worst path timing result

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	(%)	Attrs
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	3.0695e+03	72.9579	4.8587e+04	3.1911e+03	(72.70%)	
sequential	261.7708	7.0290	4.5224e+04	314.0242	(7.15%)	
combinational	274.5080	482.5377	1.2695e+05	883.9908	(20.14%)	
Total	3.6058e+03 uW	562.5247 uW	2.2076e+05 nW	4.3891e+03 uW		

Figure 3.8: Power report, total power of 4.39mW

```
Number of ports:                          123
Number of nets:                          3968
Number of cells:                         3547
Number of combinational cells:           2933
Number of sequential cells:               612
Number of macros:                           0
Number of buf/inv:                        496
Number of references:                      59

Combinational area:       5899.082070
Noncombinational area:    6161.624176
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          12060.706245
Total area:                 undefined
```

Figure 3.9: Total area report

3.4 730MHZ, ungroup, clock gating, advanced timing script

At last, I tried to clock gate the design from the previous step, so I added the *-gate_clock* option to the command previously used and ran it. This time I managed successfully to respect the slack constraint, but in the end, like before, the total power consumption actually went slightly up, making the clock gating useless. The area went slightly down, but I decided to consider this run a fail, giving the power consumption higher priority.

```
clock CLK (rise edge)                                    1.37       1.37
clock network delay (ideal)                              0.00       1.37
DataP/PC_reg/O_reg[7]/CK (DFF_X1)                        0.00       1.37 r
library setup time                                      -0.05       1.32
data required time                                                  1.32
---------------------------------------------------------------------------
data required time                                                  1.32
data arrival time                                                  -1.32
---------------------------------------------------------------------------
slack (MET)                                                         0.00
```

Figure 3.10: Worst path timing result

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
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      9.8759          49.3036             55.7645            59.2352  (   1.32%)
register       3.0643e+03          71.5953        4.8530e+04         3.1844e+03  (  71.20%)
sequential       261.0213           8.8024        4.5331e+04           315.1541  (   7.05%)
combinational    278.5742         512.7195        1.2261e+05           913.9090  (  20.43%)
--------------------------------------------------------------------------------------------------
Total          3.6138e+03 uW       642.4208 uW     2.1653e+05 nW     4.4727e+03 uW
```

Figure 3.11: Power report, total power of 4.47mW

```
Number of ports:                          123
Number of nets:                          3746
Number of cells:                         3364
Number of combinational cells:           2749
Number of sequential cells:               612
Number of macros:                           0
Number of buf/inv:                        415
Number of references:                      57

Combinational area:       5722.724071
Noncombinational area:    6166.412175
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          11889.136246
Total area:                 undefined
```

Figure 3.12: Total area report

Chapter 4

Physical Design

The last step of the project was the physical design and place and route, done through the use of the Innovus software. For this step, I decided to use the synthesized design with a frequency of 730MHz and no clock gating, as it was the better one overall.

4.1 Place and route

A first iteration of the design was discarded because of a geometry violation. A short happened between two wires coming from 2 different pins interfacing with the outside, the place where this happened can be seen marked by a white cross in Figure 4.2

```
SHORT: Regular Via of Net IRAM_ADDRESS[7] & Regular Wire of Net DRAM_DATA_IN[0]  ( metal1 )
Bounds : ( 151.455, 149.835 ) ( 151.595, 149.905 )


Begin Summary ...
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 1
  Overlap     : 0
End Summary

  Total Violations : 1 Viols.
```

Figure 4.1: Geometry report

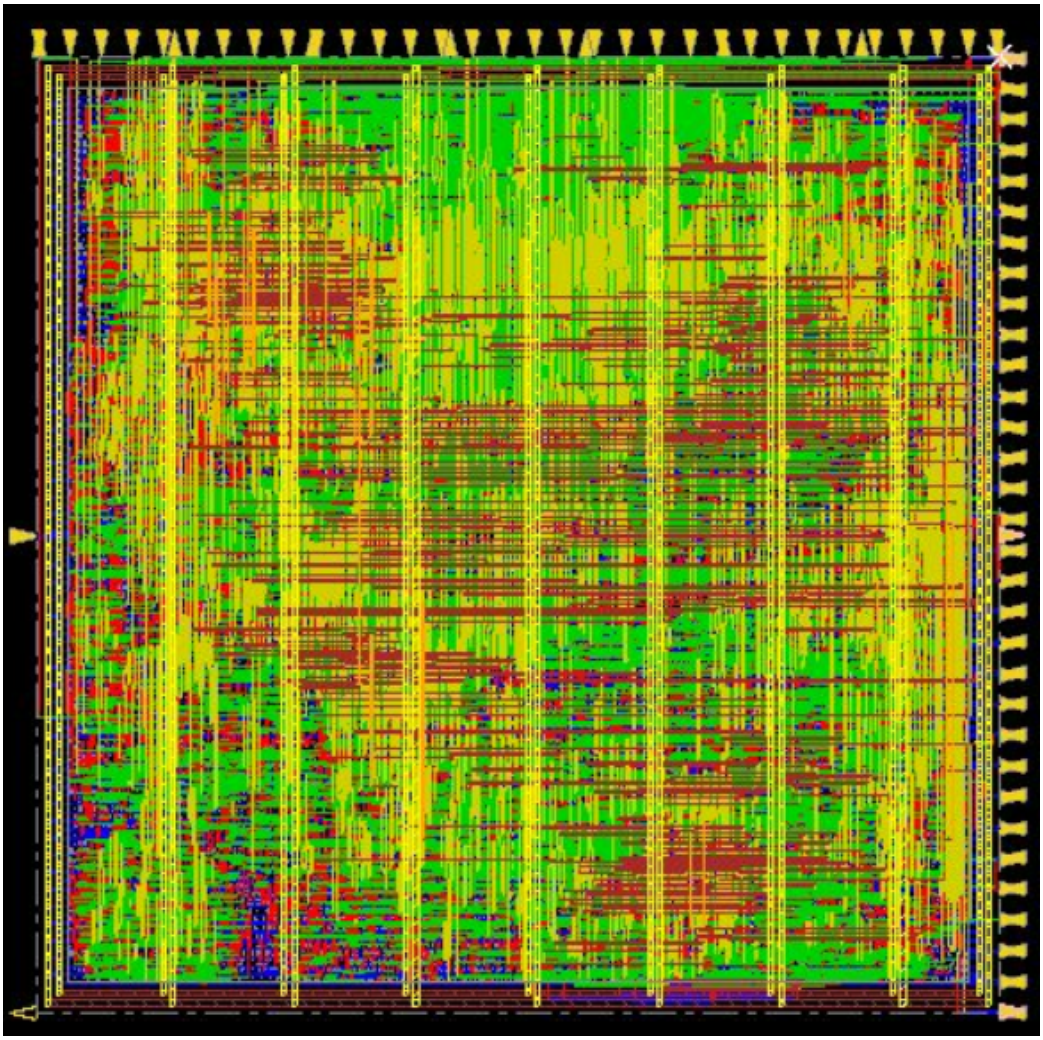


Figure 4.2: Violation on the top right corner

From the place and the kind of short I realized that the problem was most likely wire routing congestion. The program had to put both wires on the same metal layer (METAL1) and couldn't route them in a way that would avoid the shorting.

So I decided to re-do the design and spread out the pins in a way that would make that problem less likely, by not putting large buses on adjacent borders of the die.

After this more careful pin distribution (as in Figure 4.3), as expected the problem disappeared and no more violations were found.

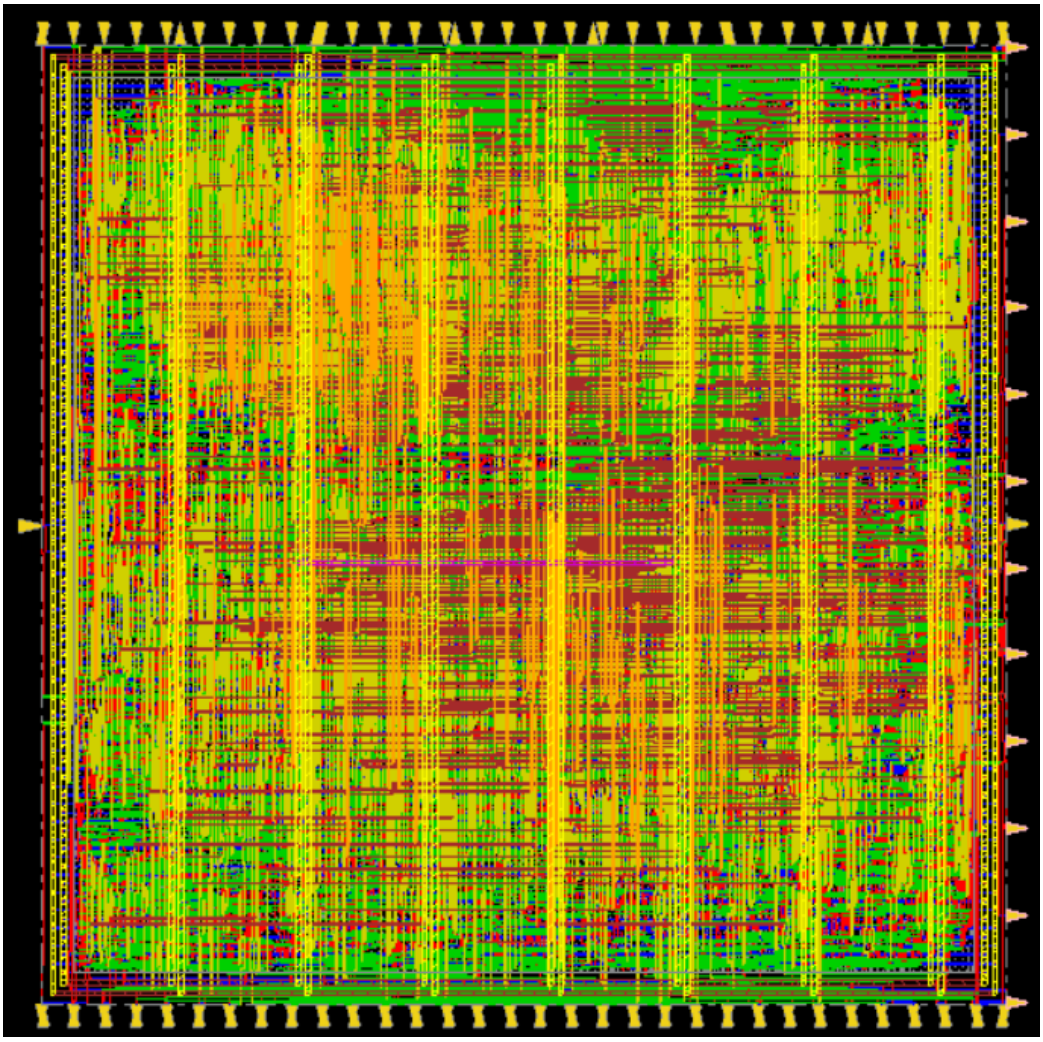


Figure 4.3: Pin distribution after re-planning

4.2 Timing reports

Both post-TCS and post-route optimizations were run during the design, and at the end reports for the resulting timing were produced. Different results for setup and hold timings were produced and read.

4.2.1 Setup times

The setup times respected the time requirements for every possible path, always maintaining a positive slack.

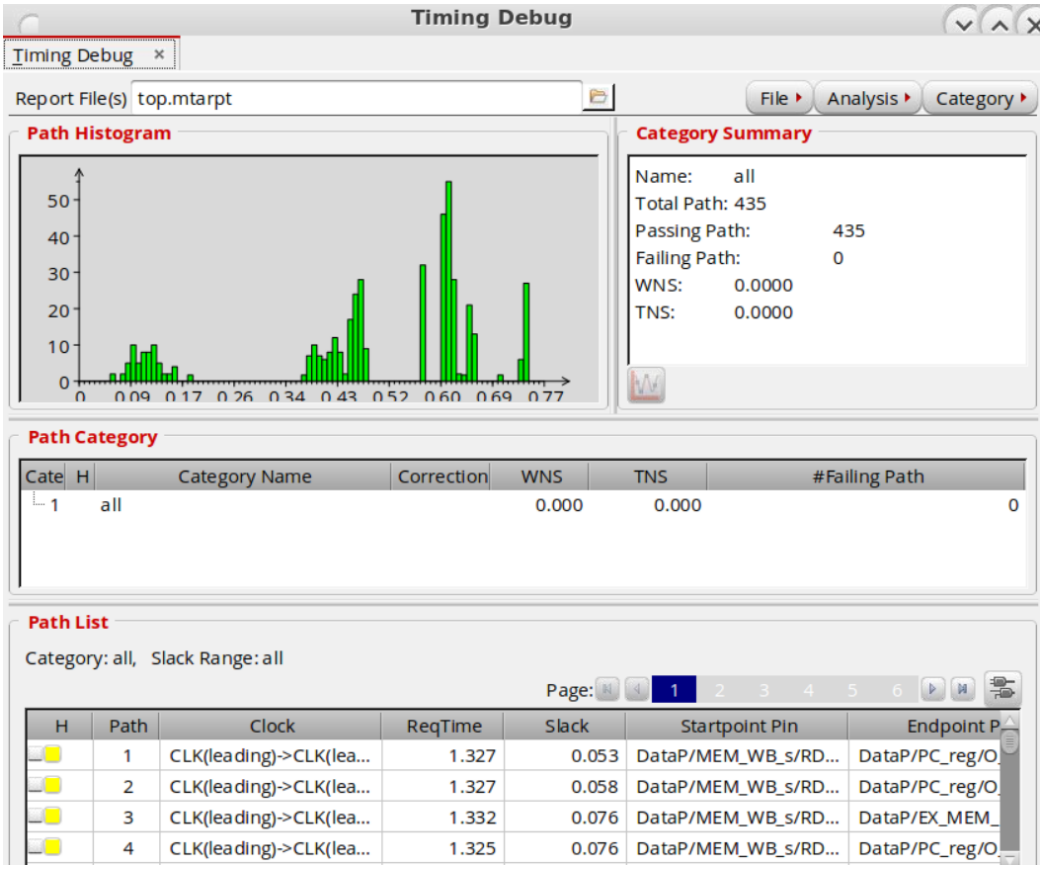


Figure 4.4: Setup times summary

4.2.2 Hold times

The hold times did **not** always respect the time requirements, with a total of 6 failing paths. The negative slack in each of those paths is very small, but still present nonetheless.

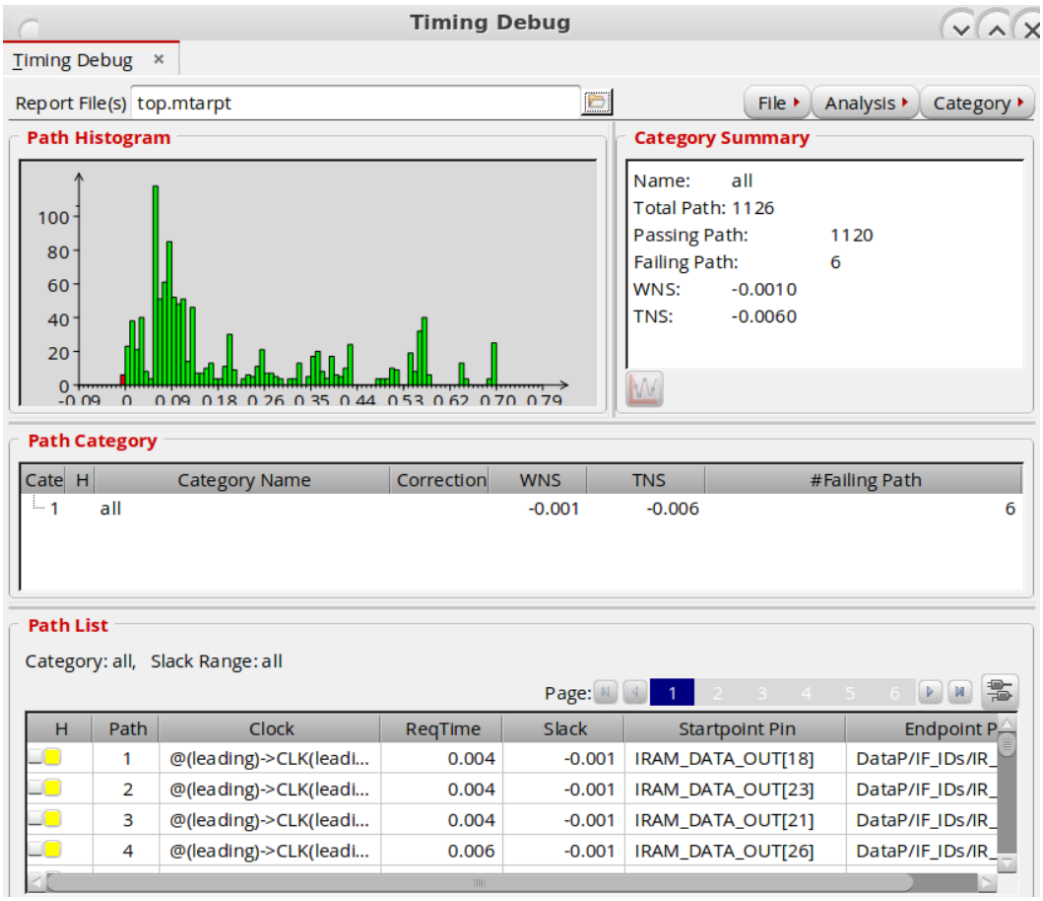


Figure 4.5: Hold times summary

Chapter 5

Conclusion and possible improvements

As it stands now, the processor is fairly limited. It is able to do reasonably complex algorithms, like the ones present in the asm files provided (for example Bubble sort, Binary search or the calculation of mcm and GCD with the use of the euclidean formula) but it is only able to do integer calculations and does not have a multiplier nor a divisor. Possible improvements for the CPU could be:

- Support for floating point operations and relative registers
- Implementation of a pipelined multiplier
- Implementation of a divisor
- Improvement of the Forwarding Unit, to support forwarding for store and load operations
- Improvement of the Branch Prediction Unit, to get rid of the penalty coming from using JR and JALR instructions
- Possible modification to the branch evaluation and flushing mechanism of the pipeline. It's possible to go from a 2-cycle penalty to 1 cycle, but that would probably impact the maximum clock frequency, so this may not necessarily be an improvement depending on the frequency of branch instructions and branch penalties in the code that's being run