POLITECNICO DI TORINO

Master Degree in Computer Engineering: Embedded Systems
MICROELECTRONICS SYSTEMS

Design and Development of a DLX Microprocessor

Professor: Mariagrazia Graziano

Autore: Alessandro Salvato 237771 (gr.45)

Data: September 17, 2018

Contents

Chapter 1

Introduction

This report aims at being a short documentation on the Microelectronic Systems course project at the first year of the master degree both in Computer and Electronics Engineering at Politecnico di Torino.
The target is the **design** and the **implementation** phase of a pipelined processor by VHDL, as is described in [1], followed by a simple **synthesis** and a **physical layout** definition. The main flow has passed through the following steps:

1. Design and implementation at RTL level
2. Simulation by assembly codes
3. Synthesis and gathering results about timing, area and power consuption
4. Realization of the physical layout

For 1 and 2, I used the *Xilinx ISE Design Suite 14.7*, although during the laboratory sessions the tool was *ModelSim*. I made this choice because I think it's better both from the management of files point of view and for the more intuitive interface of the simulation tool. The synthesis step has been performed by *Synopsys' DesignVision*, whose license has been granted to the Politecnico di Torino. In order to use it, I haved to use a virtual machine, running on *Linux*, provided by the Department of Electronics and Telecommunications servers. Same conditions for the last step, by *Cadence Innovus Implementation System*. Moreover, I liked exploiting *GitHub* functionalities to make easier the passage of data from my *Windows* laptop and the virtual machine; you can download it from `https://github.com/sandrosalvato94/MyDLX`

1.1 Basic vs Pro version

Specification are described in [4], you can find it among files in Documents directory. Summarizing, basics features for the Basic version are:

- **Pipeline**
- **Simple instruction set** (see below)
- **Simple datapath**
- **Basic synthesis**
- **Basic physical design**
- This **report**

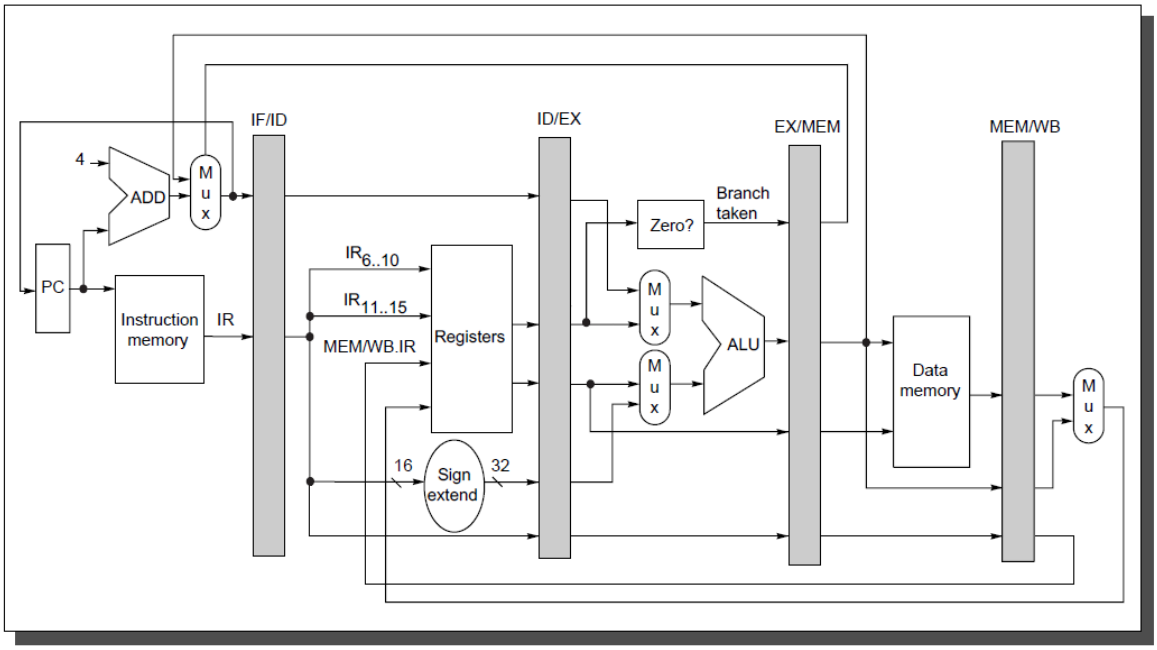

Figure 1.1: Basic DLX Datapath. Figure 3.4 from [1]

Among pro hints, these are reported:

- **Extended instruction set** (see below)
- **Extended datapath**
- **Windowing**
- **Control Hazard**
- **Optimization of the power consuption**
- **Caching**
- **Advanced synthesis**
- **Advanced physical design**
- **Whatever I wanted...**

I targeted the pro version, exploring and implementing the following features:

- **Extended instruction set**
  - **More** instructions
  - **Modified** instructions
  - **Totally new** instructions
- **Control hazard** management by a **Branch Target Buffer**
- **Data hazard** management by a **forwarding logic** (*just r-type and i-type instructions*)
- **Extended datapath** by new components

1.2 Instruction Set

Basics information on the DLX instruction structures can be found in [4].

Instruction	Type	Annotation
add addi and andi beqz bnez j jal lw nop or ori sge sgei sle slei sll slli sne snei srl srli sub subi sw xor xori	Basic DLX	n.27
addui subui lhi jr jalr srai seqi slti sgti lb lh lbu lhu sb sh sltui sgtui sleui sgeui sra addu subu seq slt sgt sltu sgtu sleu sgeu	Pro DLX	n.29
mult multu	Modified instructions	The binary format is changed. mult(u) r1, r2, r3 has been replaced by **mult(u) r2, r3** <– where the product is stored in two special registers, one for the 32 lowest bits and the other for the 32 highest.
mflo mfhi	New instructions	**mflo r1** <– loads the 32 lowest bits of a product in r1 **mfhi r1** <– loads the 32 highest bits in r1

Table 1.1: Instruction set

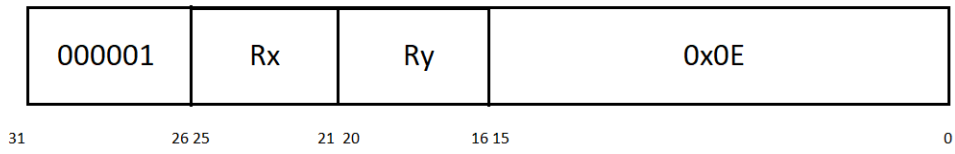

Figure 1.2: Mult format

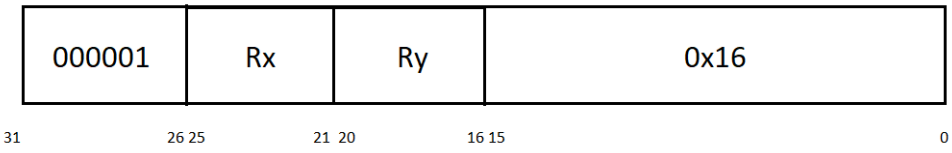

Figure 1.3: Multu format

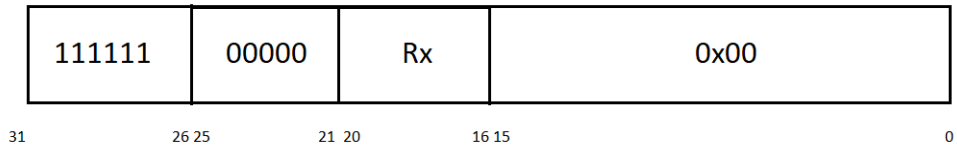

Figure 1.4: Mflo format

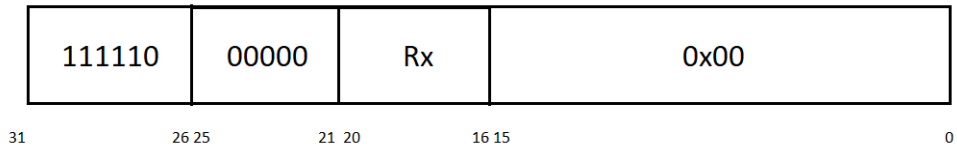

Figure 1.5: Mfhi format

More information on Basic and Pro DLX instruction in [4].

Chapter 2

Register Transfer Level

2.1 Core

This is the highest level of the design. It's interfaced with the external world and the data and the instruction memory, which are not reported in this document.

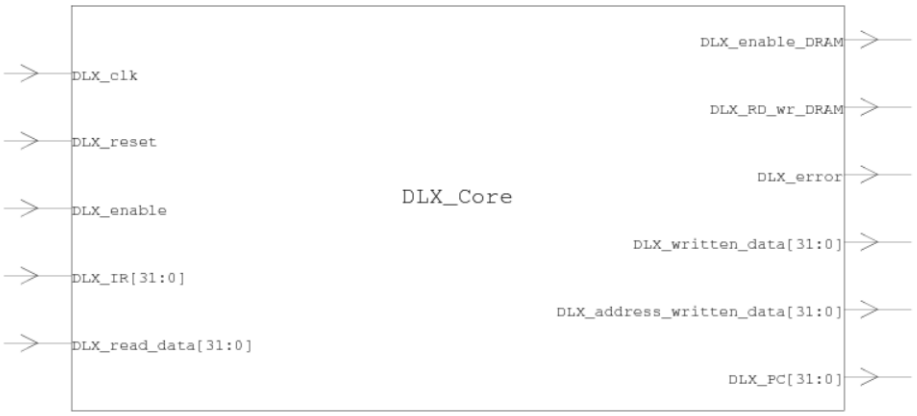

Figure 2.1: Core pinout

Just two informations about these signals. *Clk* and *enable* own their basic functionalities, *IR* is the data read from the instruction RAM at each clock cycle by fetching. *Read_data* comes from the DRAM; *enable_DRAM* allows read and write operation on it, where the choice is made by *RD_wr_RAM*. *Error* is just a single pin out used for debugging, so not consider it. *Written_data* and *address_written_data* are signals to the DRAM; in the end the *PC* to the IRAM.
The core is composed by 4 macroblocks:

- Datapath
- Control unit
- Branch target buffer
- Branch misprediction manager

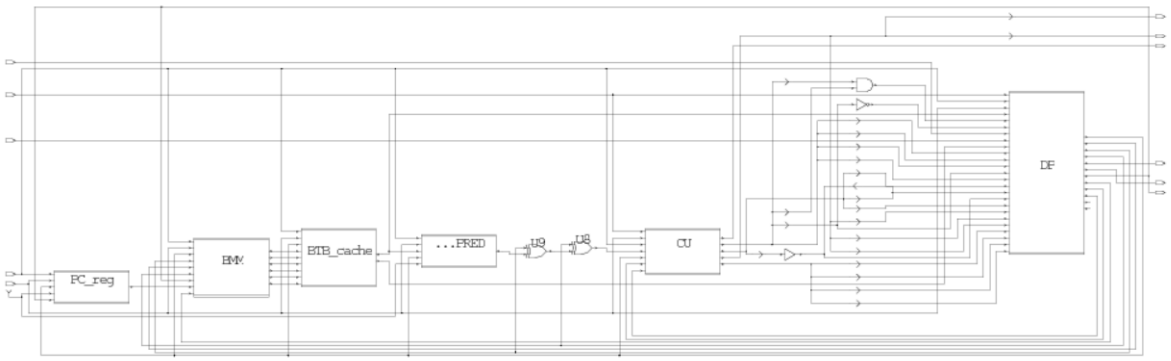

Figure 2.2: Core schematic - I

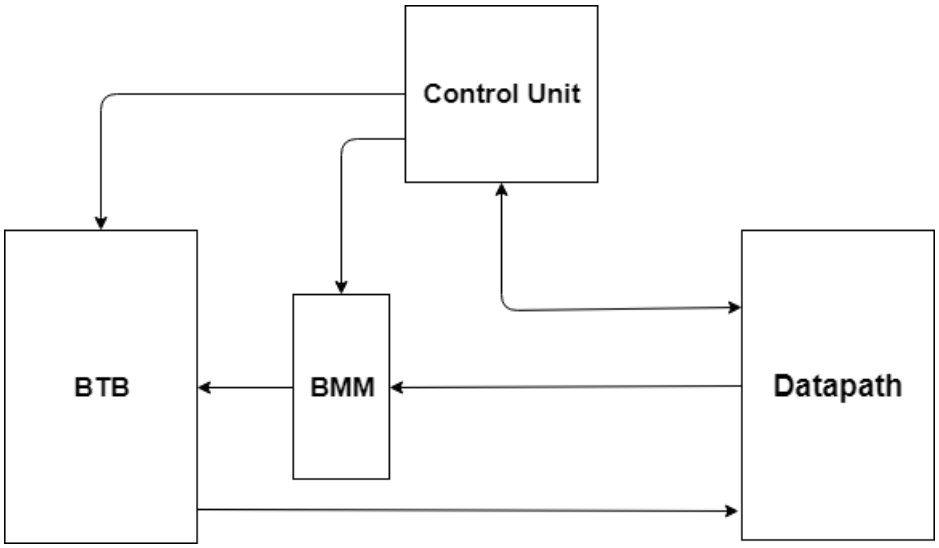

Figure 2.3: Core schematic - II

The datapath is the part of microprocessor computing instructions. In this design is the biggest component, where are defined the five pipeline stages. The branch target buffer is a sort of cache memory, used to manage control hazards; in fact branch addresses are stored in it. However almost all communications between datapath and BTB are managed by the Branch Misprediction Manager, which works when a wrong prediction occurs. Finally there is the hardwired Control Unit, that is the main brain of the DLX; as the datapath it's pipelined too.
Datapath sends to BMM, so to BTB, information about the current instruction and the previous one such as: the program counter and whether one instruction is a branch or not. On the other side the BTB responds sending the branch target and the boolean value of the prediction (taken or untaken). The control unit oversees all macroblocks. It receives the instruction register of the instruction getting in the decode stage and produces all control signals.

2.2 BTB

I guess talking about BTB before datapath should let you understand better the overall behaviour of this microprocessor. Basically a Branch Target Buffer behaves like a cache. It's composed of

a set of rows, where each rows defines three fields :

- Entry
- Target
- Status of prediction

The entry is basically the value of the PC related to the branch instruction. The target stands for the address to jump if that branch is taken. The status is held by a saturation counter, where the most significant bit defines the kind of prediction. In this design, this component has 32 entry 32 bits long, as well as the PC, the same for the target. Saturation counters count on 3 bits.

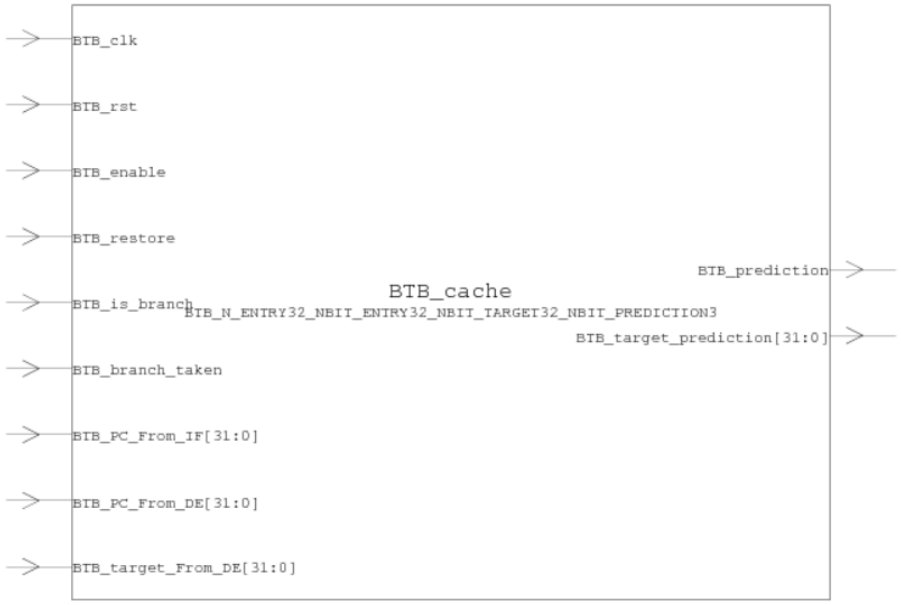

Figure 2.4: BTB pinout

Now it's important to focus on the meaning of each signals and understand the timing

Signal	**From**	**To**	**Note**
BTB_enable	Control Unit	-	Active high. It gets low because of stalls
BTB_is_branch	Datapath->BMM	-	It is generated during the decode stage. It's high when the instruction is a branch, low otherwise
BTB_restore	Datapath->BMM	-	When a bad prediction is given, it's needed to restore the previous status of prediction
BTB_branch_taken	Datapath->BMM	-	It's generated by decode stage. Regardless of kind of instruction and prediction, it provides a correct information on any kind of change of context. This signal is used by BTB to update the SAT

Table 2.1: BTB signals I

Signal	From	To	Note
BTB_PC_from_IF	Datapath->BMM	-	It's the value of the PC of the instruction at fetching. It's sent to BTB immediately, so in case of hit, the predicted PC is granted back at the next clock cycle, without wasting time
BTB_PC_from_DE	Datapath->BMM	-	It's the value of the PC of the branch instruction at decoding. It's used just in case of miss to store the new entry. It usually has the same value of BTB_PC_from_IF at the next cc
BTB_target_from_DE	Datapath->BMM	-	It's the value of the target to be jumped of the branch at decoding. It's used just in case of miss to store the new target.
BTB_prediction	-	Datapath	True: prediction taken. False, otherwise
BTB_target_prediction	-	Datapath	New PC to be loaded in the fetch PC register

Table 2.2: BTB signals II

Look at the following code, which exploits the branch target buffer and analyse what happens inside. After the instruction is reported the address where it's stored in the IRAM.

Listing 2.1: Branch.asm

```
nop                             ;0
nop                             ;4
addi r1, r0, 100        ;8
xor r2, r2, r2          ;12

ciclo:
lw r3, 0(r2)            ;16
addi r3, r3, 10         ;20
sw 100(r2), r3          ;24
subi r1, r1, 1          ;28
addi r2, r2, 4          ;32
bnez r1, ciclo          ;36


addi r4, r0, 65535 ;40
ori r5, r4, 100000      ;44
add r6, r4, r5          ;48

end:
j end                           ;52
```

Miss

The first main event occurs at 125 ns, when the **bnez** instruction is fetched. You can see PC_from_IF is 36, the previous PC instruction in decode, PC_from_de is 32; all entries and targets are empty. The saturation counter in the pictures is related to the first entry, others else are not reported, since they aren't used in this example. At 135 ns the fetched instruction is

the `addi`, but here the DLX is already wasting time, because we know that `bnez` is taken. However, PC_from_de and Target_from_de report all significant data of the branch instruction; moreover, the decode stage provides BTB also the information that `bnez` is a branch instruction and that it'has been a real taken. These data are stored by BTB: PC_from_de among entries, Target_from_de among targets and finally the saturation counter is set to 100.
If the set of rows is full the replacement policy adopted is the **first in first out**, implemented by means of a rotate register.

Hit

At 205 ns `bnez` is fetched again. Immediately BTB matches and sends out the prediction and target. This leads an advantage because at 215 ns the PC is 16, rather than 40. No clock cycles are lost. Concurrently the saturation counter increases by 1, strengthening the taken status.

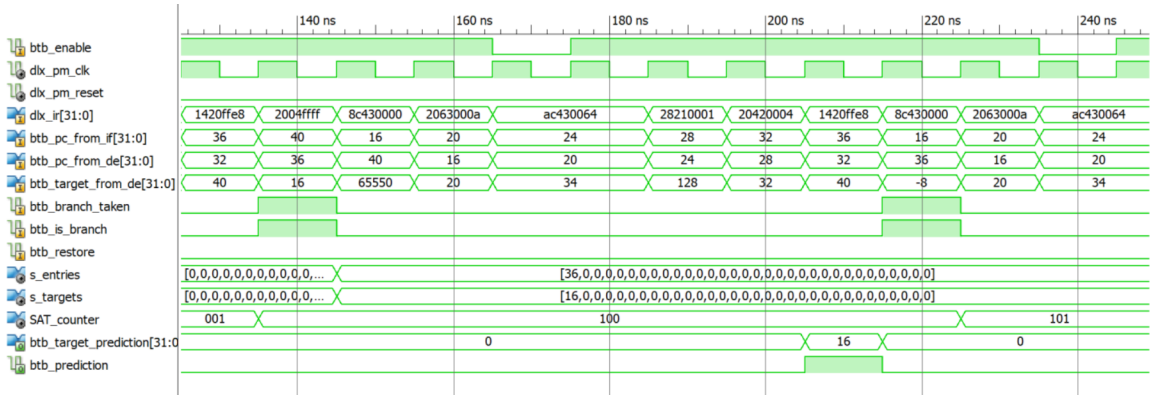

Figure 2.5: BTB waveform I

Restoring

Execution goes on until the last iteration. At 7065 ns `bnez` is fetched, BTB predicts taken. At 7075 ns the PC is updated to the target value, but decode stage recognizes the error and the datapath grants the restore signal. Remember that between BTB and datapath there is the Branch Misprediction Manager, which runs in transparent mode when everything is OK, but it gets working when a restore of the BTB is required. BMM recovers the entry and the target of the instruction with a wrong prediction, then for 2 clock cycles it keeps them at the BTB input. The same procedure works with Is_branch and Branch_taken signal; the latter is negated by BMM. Restoring protocol is used to fix the saturation counter content. Let's consider ϕ_1 as the current value of the saturation counter.

1. BTB sends out prediction, here taken
2. Taken prediction makes $\phi_2 = \phi_1 + 1$: actually no branch is taken
3. Restoring triggers. First cycle $\phi_3 = \phi_2 - 1$, it's a backtrack
4. Second restoring cycle $\phi_4 = \phi_3 - 1$, saving the correct and updated status

Finally $\phi_4 = \phi_1 - 1$. The reason why I choose 3 bits follows. Let's consider a weak taken $\phi_1 = 10_2$, but branch untaken, so at the end $\phi_4 = 01_2$ a weak untaken, here everything works well. Now let's take into account a strong taken $\phi_1 = 11_2$, but branch untaken, at the end we get $\phi_4 = 01_2$, ϕ_2 saturates to 11_2 and at the end of the procedure we'd have a weak untaken: that's not make any sense. Adding a third bit overrides this issue and from a strong taken the status moves to a weak one: $\phi_1 = 111_2 \rightarrow \phi_2 = 111_2 \rightarrow \phi_3 = 110_2 \rightarrow \phi_4 = 101_2$

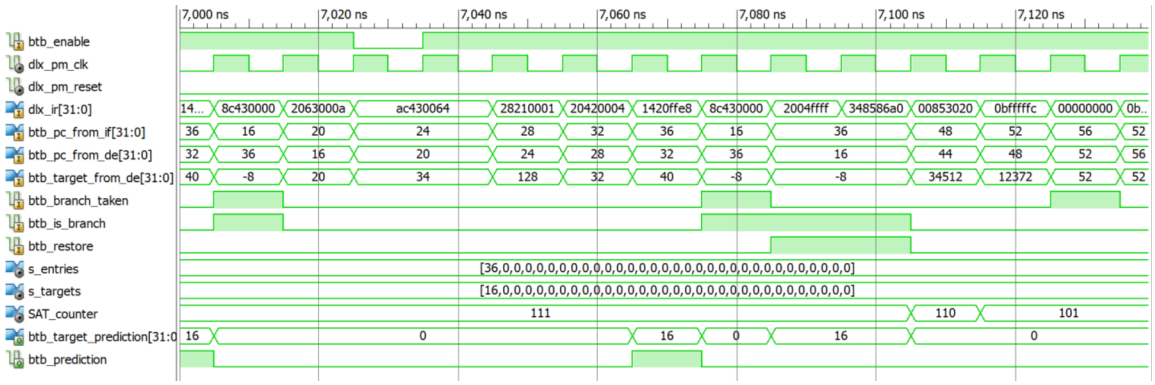

Figure 2.6: BTB waveform II

Main components making up the Branch Target Buffer are

- 32 Comparators (just check eq.)
- 1 Priority Encoder
- 64 Registers 32 bits
- 1 Rotate register
- 32 Saturation counters

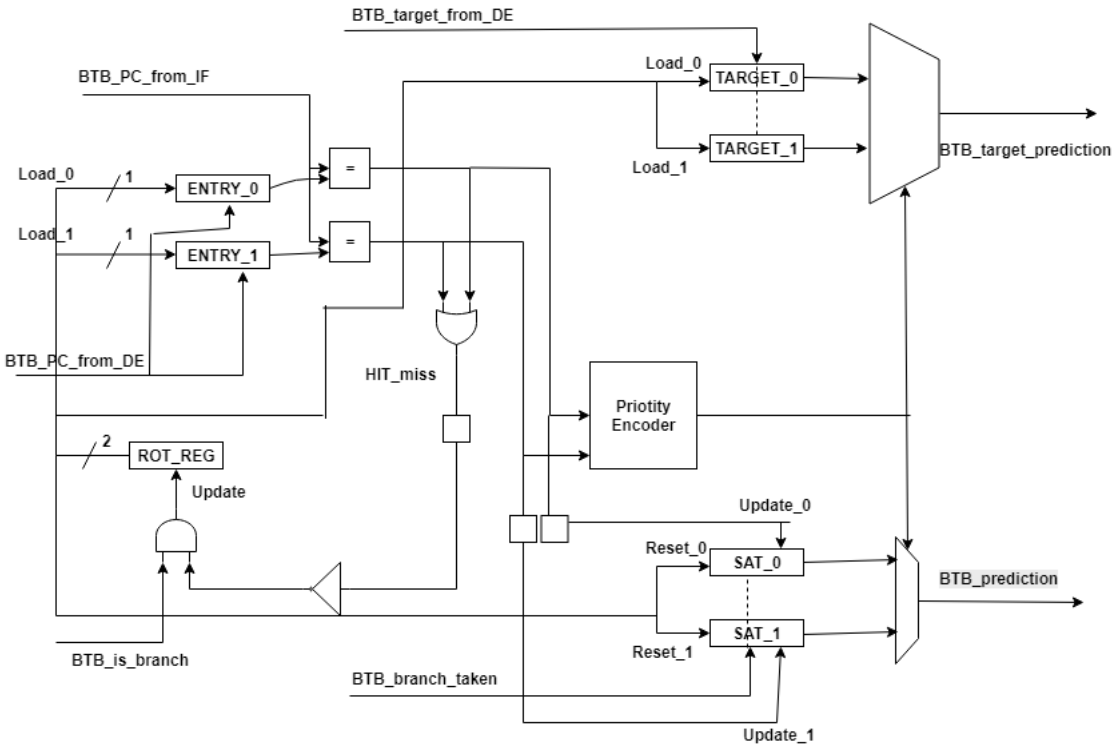

Figure 2.7: BTB simplyfied microarchitecture

Comparator with enable

This component matches the entries with the value of the PC. Further usual inputs I added the enable in order to avoid errors due to combinational logic, when the BTB is off.

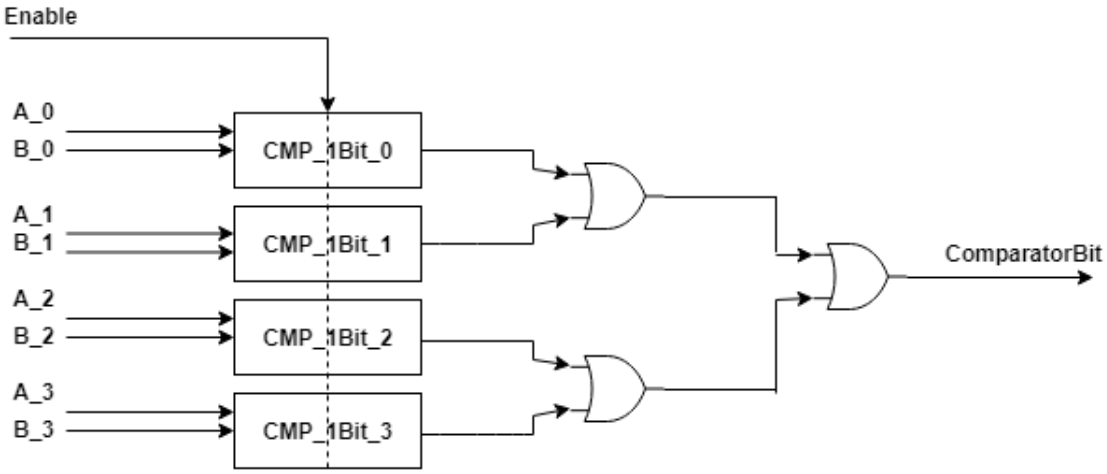

Figure 2.8: Comparator 4 bits schematic

As you can see, this component has 2 areas. The former is a set of 1-bit comparator, the latter is a pyramid of OR gates, in order to compute the final result: true equal, false unequal. Deeper, the 1-bit comparator has the following structure:

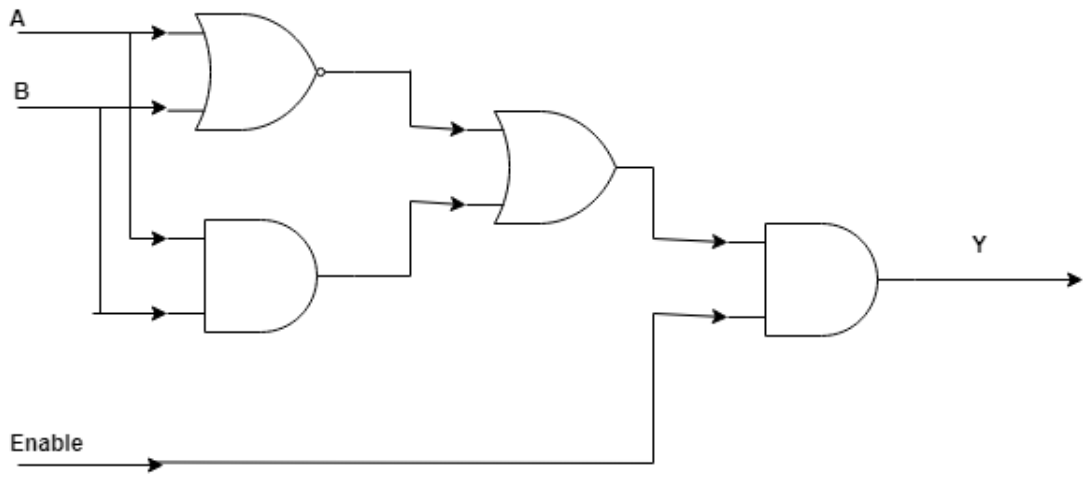

Figure 2.9: Comparator 1 bit schematic

A	B	Enable	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 2.3: Comparator 1 bit truth table

Rotate register

The BTB exploits a rotate register in order to implement the replacement policy ([2]) First In First Out. This technique requires a pointer which is updated in case of a miss; the role of the pointer is covered by the rotate register output. When the BTB is resetted all the content of the rotate register is set to 0, except for the most significant bit to 1; moreover is never possible to load new data inside. It means that just reset, rotate (by 1) and keeping modes are allowed. That single bit to 1 is the pointer I'm referring to.

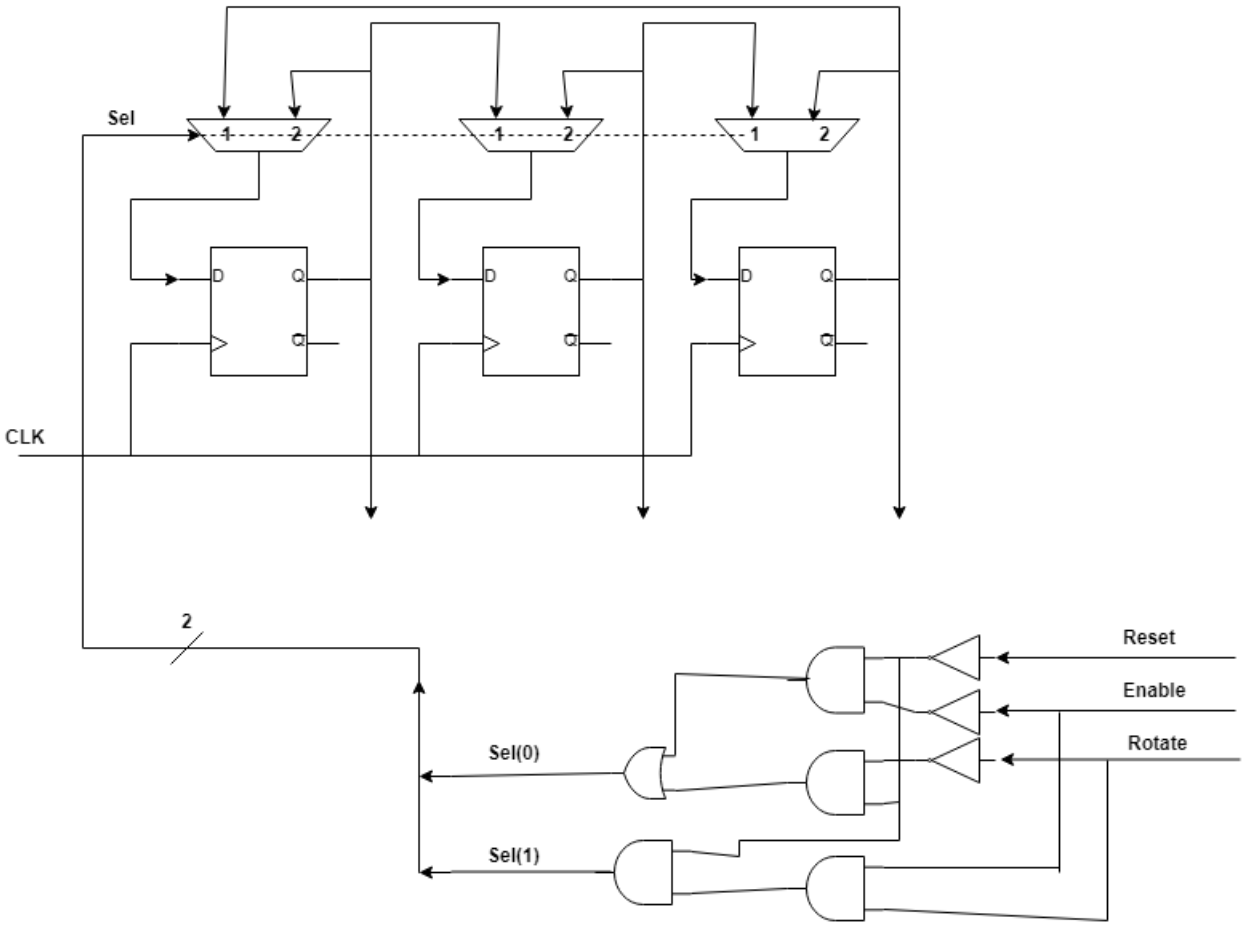

Figure 2.10: Rotate register schematic

Reset	Enable	Rotate	Sel
0	0	0	10 Keep
0	0	1	10 Keep
0	1	0	10 Keep
0	1	1	01 Rotate
1	-	-	00 Reset

Table 2.4: Multiplexer selector truth table

Saturation counter

A saturation counter is a finite state machine which behavies like a usual counter, implementing up and down operations, but the difference is that it cannot get either overflow or underflow.

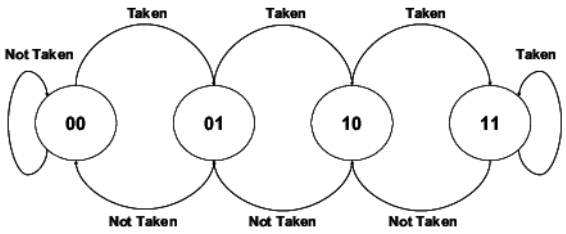

Figure 2.11: 2-bit SAT counter FSM

The usual architecture of a FSM is composed of a control unit and a datapath. The role of the datapath is covered by a UpDownCounter, which is not a saturation one. The UDCounter can get overflow and underflow. On the other side the control unit checks the status of the datapath in order to stall the counting in case an Up is required when has been already reached the maximum, or a Down has to be performed when has been already hit the minimum value.

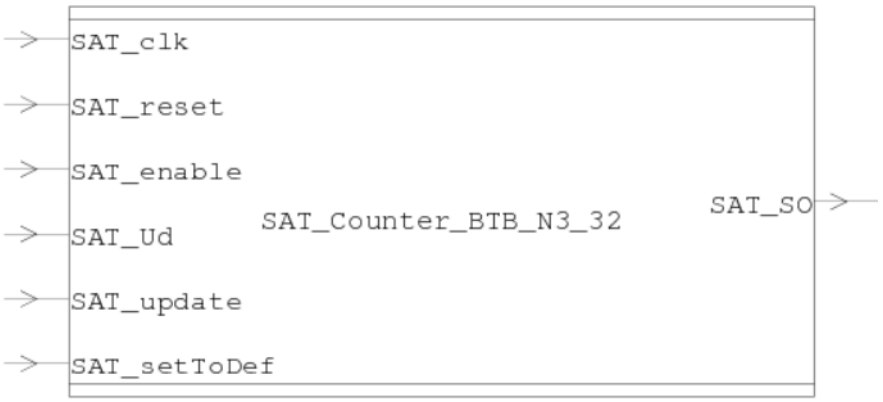

Figure 2.12: SAT counter pinout

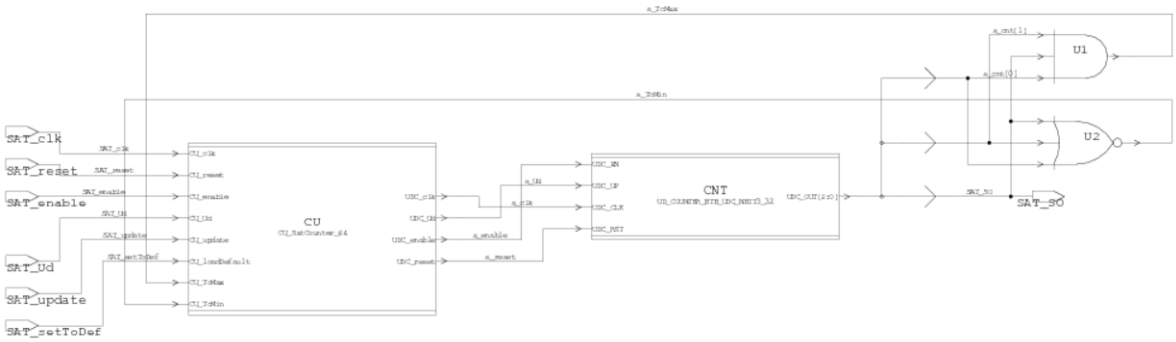

Figure 2.13: SAT counter microarchitecture

From the VHDL point of view, the control unit owns a behavioral description, the UDCounter a structural one, built with TFF. Since I designed a reset value equals to a weak taken status (MSB = '1', others '0'), two versions of FF are used, where the difference is just the value to load into when reset signal is triggered.
The most significant bit is used as predictor.

2.3 BTB misprediction manager

The BTB misprediction manager is the interface between datapath and BTB. This component is totally original, I didn't get inspired from anything, I designed just functionalities I needed. It has two operative modes:

- **Transparent** : Signals go from datapath to branch target buffer without incurring any penalties
- **Restoring** : Data are managed in order to recover the correct status of the saturation counter of the entry having been predicted in a wrong way

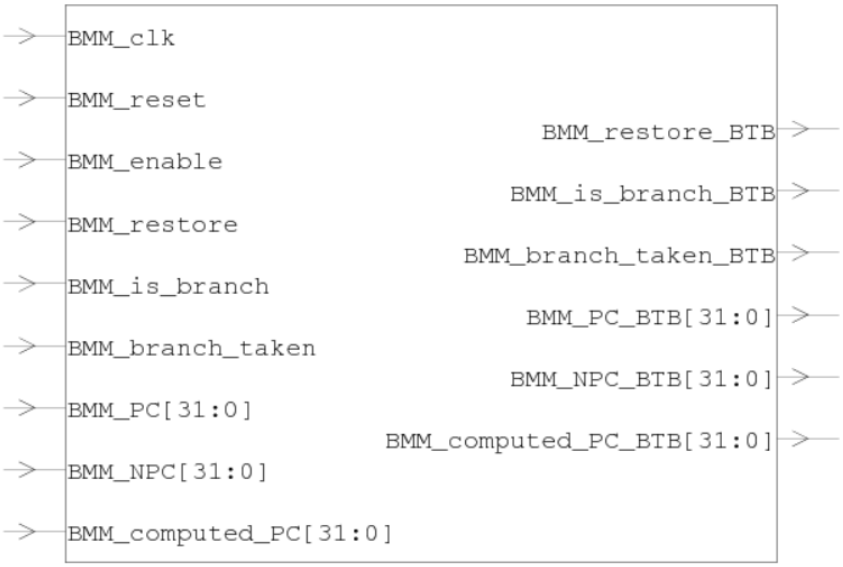

Figure 2.14: BMM pinout

Signal	From	To	Note
BMM_enable	Control Unit	-	Active high. It gets low because of stalls
BMM_restore	Datapath	-	When a bad prediction is given, it's needed to restore the previous status of prediction. When 1 triggers **restoring mode**, 0 keeps **transparent**
BMM_is_branch	Datapath	-	It is generated during the decode stage. It's high when the instruction is a branch, low otherwise
BMM_branch_taken	Datapath	-	It's generated by decode stage. Regardless of kind of instruction and prediction, it provides a correct information on any kind of change of context
BMM_PC	Datapath	-	It's the value of the PC of the instruction at fetching.
BMM_NPC	Datapath	-	It's the value of the PC of the branch instruction at decoding.
BMM_computed_PC	Datapath	-	It's the value of the target to be jumped of the branch at decoding.
BMM_restore_BTB	-	BTB	Read BMM_restore
BMM_is_branch_BTB	-	BTB	Read BMM_is_branch
BMM_branch_taken_BTB	-	BTB	Read BMM_branch_taken
BMM_PC_BTB	-	BTB	Read BMM_PC
BMM_NPC_BTB	-	BTB	Read BMM_NPC
BMM_computed_PC_BTB	-	BTB	Read BMM_computed_PC

Table 2.5: BMM signals

Transparent

Signals go through BMM when BMM_restore is 0, without wasting times.

Restoring

It triggers when BMM_restore goes to 1. It actives a saturation counter computing just down steps. That 2-bit counter starts from a value of 10. The output is connected to a XOR gate, whose result is 1 when the 2 input bits are not equal. This logical gate produces the enable, then negated, for registers, that will store all data coming from datapath; moreover the output multiplexers will switch to the input port wired to those registers outputs. As you can understand, the restoring mode lasts 2 clock cycle, the BMM output is driven to freezed values. Concurrently the register enable is kept to 0, by XNORing, for all the duration of this procedure.
BMM_restore is driven to 1, by datapath, just for 1 clock cycle (not 2), this means that it's possible the scenario where restoring is running and BMM_restore is 0.
The BMM microarchitecture is pretty regular, a small excpetion for BMM_branch_taken connections. Its register gets as input the negated value. In fact, when restoring procedure triggers

a wrong prediction occured, so BTB has to receive the opposite branch (real) condition.

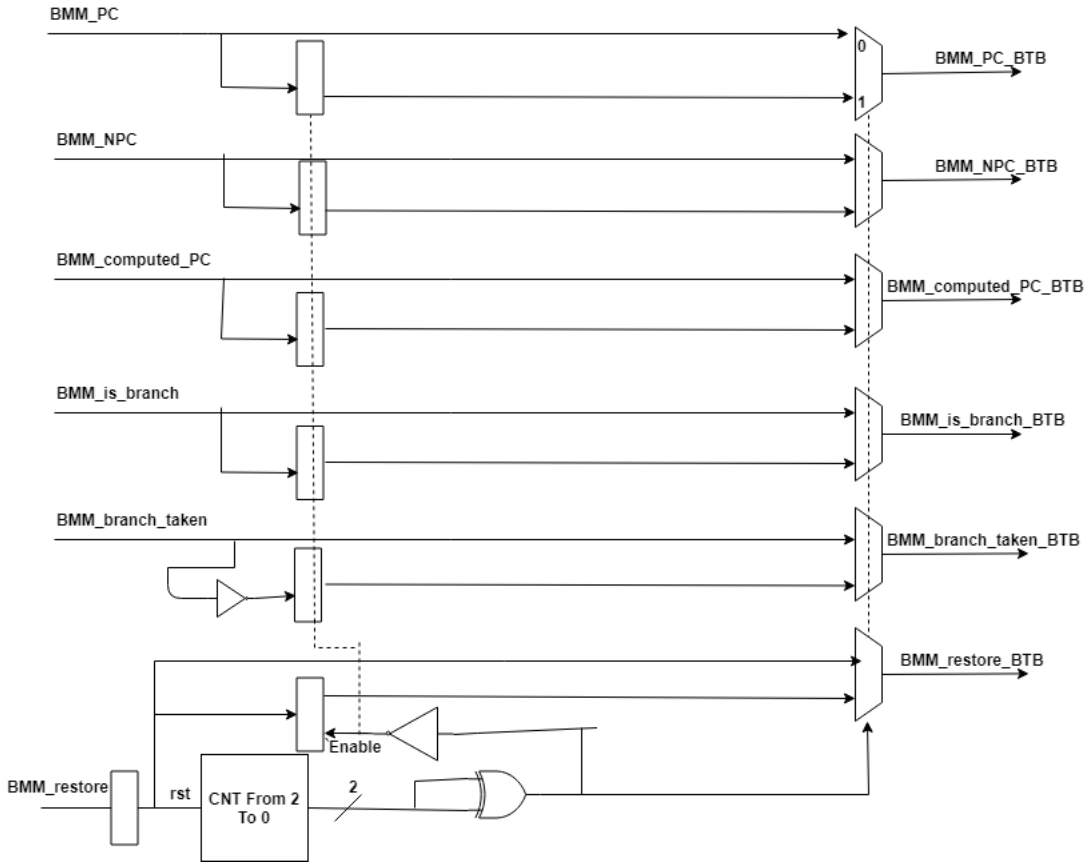

Figure 2.15: BMM microarchitecture

2.4 Datapath

The datapath is the biggest macroblock of the DLX, here instructions are read, executed and committed. Pipeline is composed of 5 stages:

1. **Fetch**
2. **Decode**
3. **Execute**
4. **Memory**
5. **Write Back**

In addition to these macroblocks, there is the forwording control unit, which aims at driving multiplexers between two stages, identifying what kind of instruction is run at each stage, inserting stalls.
As you can see from the picture, several signals are involved in, some from IRAM and DRAM, many from main control unit, some toward DRAM and other else to BMM.

Figure 2.16: Datapath pinout

Signal	From	To	Note
DP_btb_prediction	BTB	-	Value of a branch prediction. 1 taken, 0 untaken
DP_Rd1 DP_RD2 DP_wr	Control Unit	-	Signals for register file control. 1 on, 0 off
DP_save_PC	Control Unit	-	Signal for saving PC in a RF register. 1 on, 0 off. jal and jalr instructions exploit it
DP_use_immediate	Control Unit	-	Enable decode stage considering immediate value, rather than RF address. 1 enable, 0 otherwise. jal and jalr instructions exploit it
DP_reverse_operands	Control Unit	-	Inverts input operands at the execute stage. 1 enable, 0 otherwise

Table 2.6: Datapath signals I

Signal	From	To	Note
DP_EX_enable	Control Unit	-	Enable execute stage modules. 1 enable, 0 otherwise
DP_Store_reduce	Control Unit	-	Enable reducing of a data, to be stored in DRAM, to an half word or a byte. 1 enable, 0 otherwise
DP_Store_BYTE_half	Control Unit	-	1 reduces to a byte, 0 reduces to an half word
DP_WB_sel	Control Unit	-	Drives data to be stored in RF. 1 data from DRAM, 0 data from datapath
DP_Load_reduce	Control Unit	-	Enable reducing of a data, to be loaded from DRAM, to an half word or a byte. 1 enable, 0 otherwise
DP_Load_BYTE_half	Control Unit	-	1 reduces to a byte, 0 reduces to an half word
DP_Load_SGN_ung_reduce	Control Unit	-	1 reduces to a signed, 0 reduces to an unsigned
DP_btb_target_prediction	BTB	-	Predicted PC
DP_IR	IRAM	-	Fetched instruction
DP_JMP_branch	Control Unit	-	Control signals for Jump and Branch manager logic in the decode stage
DP_sign_extender	Control Unit	-	Control signals for sign extender logic in the decode stage
DP_shift_amount_sel	Control Unit	-	Drive muxes selecting the amount for shifting operation
DP_ALU_opcode	Control Unit	-	Let ALU perform right operation
DP_UUW_sel	Control Unit	-	Drive the output of the execute stage among the ALU, MFLO reg and MFHI reg
DP_Load_data_from_DRAM	DRAM	-	Data read by a load instruction
DP_insert_bubble	-	Control Unit, BMM, BTB	0 stalls pipeline, 1 otherwise
DP_IF_ID_instr_is_branch	-	BMM	1 fetched instruction is a branch, 0 otherwise
DP_restore_BTB	-	BMM	Triggers restiring BTB status procedure
DP_branch_taken	-	BMM	Jump or branch occuered
DP_PC	-	IRAM, BMM	Current value of the program counter
DP_NPC	-	BMM	Current value of the next program counter
DP_IR_opcode	-	Control Unit	$IR_{31..26}$
DP_IR_func	-	Control Unit	$IR_{10..0}$
DP_computed_new_PC DP_target	-	BMM	Value to be jumped in case of branch taken
DP_data_to_DRAM	-	DRAM	Data to be written in DRAM
DP_address_to_DRAM	-	DRAM	Address of the data to be written in DRAM

Table 2.7: Datapath signals II

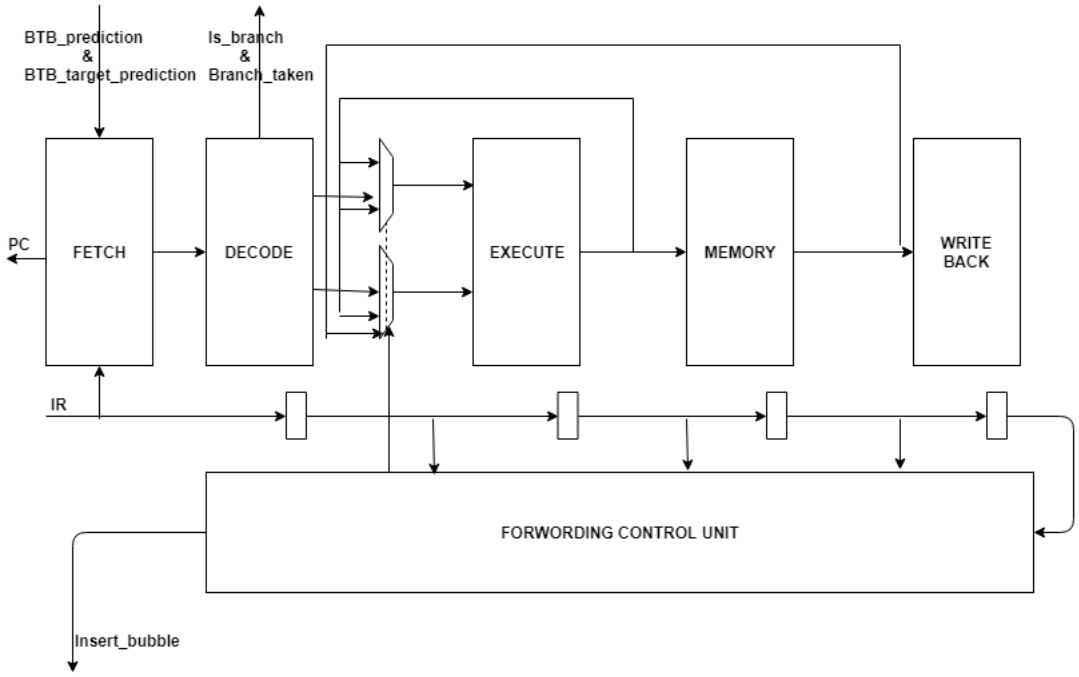

Figure 2.17: Simple datapath schematic

2.4.1 Fetch

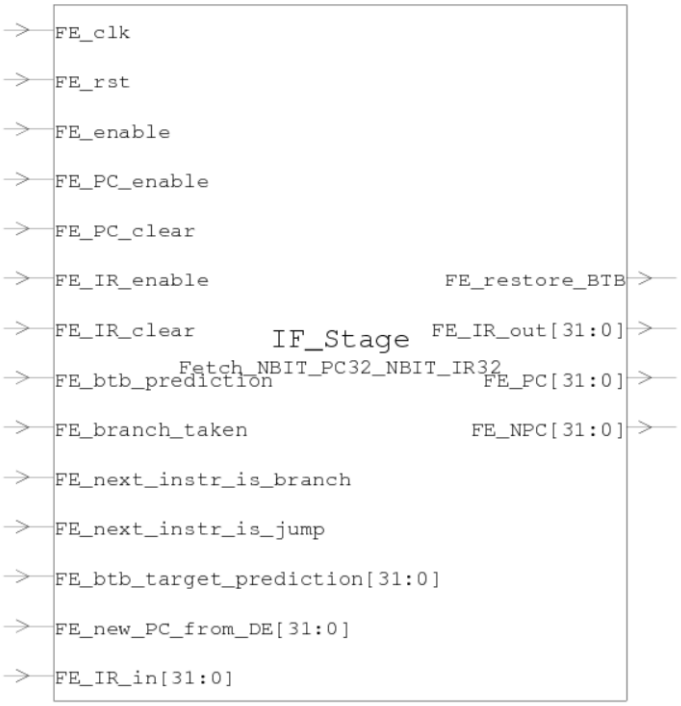

Figure 2.18: Fetch stage pinout

It's up to the fetch stage the read of a new instruction from the IRAM, by sending out the PC, then it's updated adding 4. This stage manages 4 scenarios:

- **Normal**: PC updated by adding 4
- **Branch prediction** : PC is updated with the value of DP_BTB_target_prediction, whether DP_BTB_prediction is 1
- **Jumping** : PC is updated due to a jump instruction
- **Restore** : After a wrong prediction, fetch recovers the right value of the PC, i.e. the NPC before updating

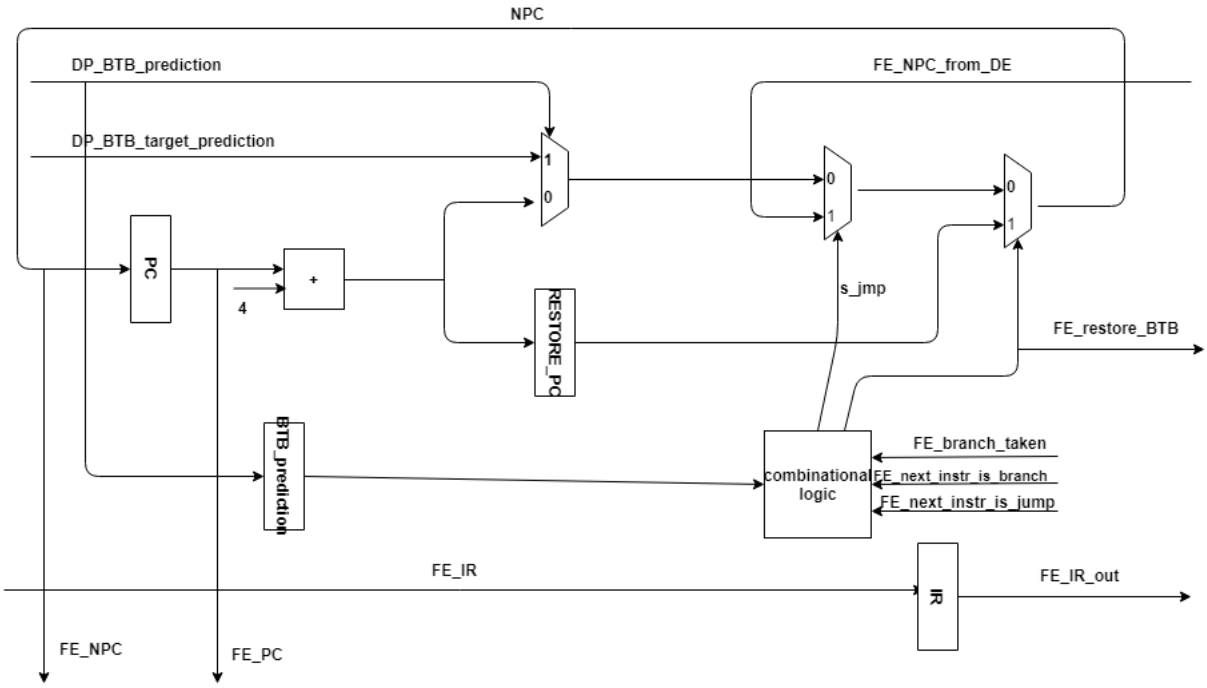

Figure 2.19: Fetch stage microarchitecture

Signal	Normal	Branch prediction	Jumping	Restore
DP_BTB_prediction0 *if*	0	a	-	-
DP_BTB_prediction1 *de*	-	-	-	a
FE_branch_taken	0	a	1	$\overline{a}$
FE_next_instr_is_branch	0	1	0	1
FE_next_instr_is_jmp	0	0	1	0

Table 2.8: Fetch running mode

From picture and table note BTB_prediction signal. A register is used to store it, because jumping and restore are checked at the next clock cycle, it means when the instruction is under decoding. As you can ses, all combinational logic inputs come from decode stage. In the table above DP_BTB_prediction0 is referred to the fetch stage, DP_BTB_prediction1 to the decode one. I choose to design that circuitry, using signals coming from decode, here, because I thought

that PC management feature was more pertinent in the fetch stage, where PC is computed. Actually no complex components make it up. It's just a combination of registers, one adder, some multiplexers and logic gates.

2.4.2 Decode

The main tasks of the decode stage are:

- To provide operands to the execute stage. Operands are
  - Read from the register file
  - Read from the IR as immediate
- To store data coming from write back stage
- To check branches and jumpes, providing to fetch new PC to be updated

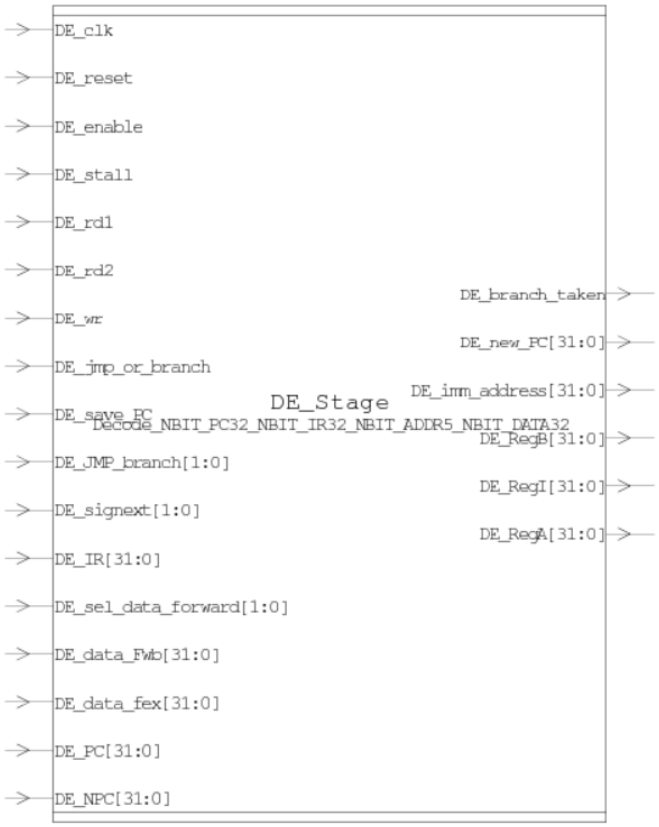

Figure 2.20: Decode stage pinout

Just a look on the architecture. There is a pipeline of 3 stages. They are used to syncronize the write operation on the register file; in fact those informations (write enable and address) come from the write back stage, three clock cycle later. A generic enable signal is pipelined as well, since stalls may be propagated along the pipeline.
On the left 2 multiplexers manage the write address: the mostleft one switches depending on the type of the instruction at the decode stage. In fact the binary format of r-type and [i,l,s]-type differs from the group of bits stressing the write register. R-type $IR_{20..16}$, [i,l,s]-type $IR_{15..11}$. Selector is basically the register file read enable of the second operand. The other one takes into

account 31 as one of the inputs; in the instruction set there are 2, `jal` and `jalr`, saving PC into. In fact its selector is basically a signal coming directly from the control unit.

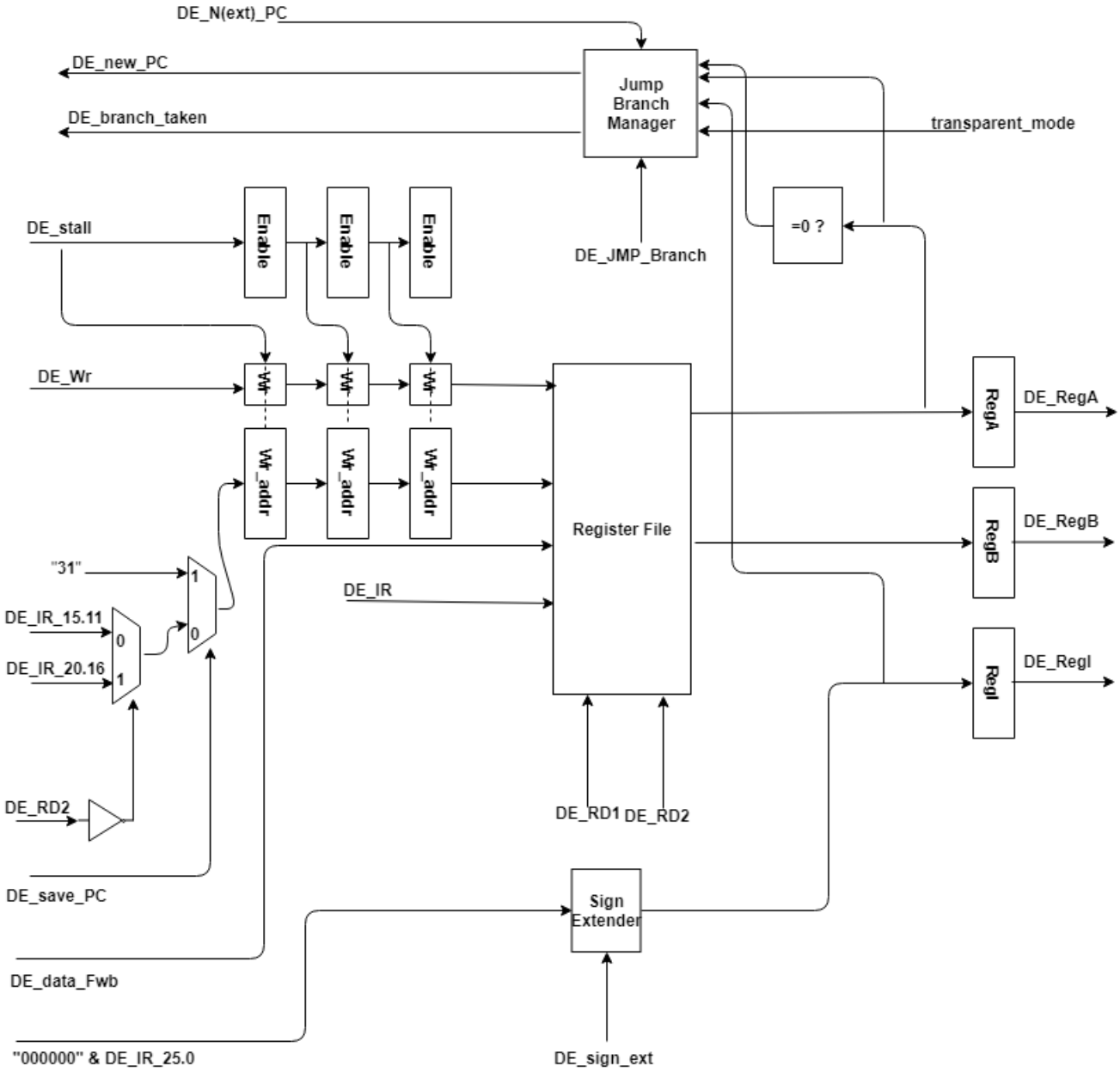

Figure 2.21: Decode stage microarchitecture

Usually zero testing and branch target calculation are performed in EX, but the stall from branch hazards can be reduced by moving them into the ID phase of the pipeline. That's what it has been designed [1].
Summarizing DE stage is composed of

- Register File
- Sign extender

- Comparator (just eq to 0)
- Jump branch manager
- General registers
- Multiplexers

2.4.3 Execute

The EX stage is the part of the pipeline where arithmetical operations are performed. It reads operands coming from DE and writes the result, ready to be stored in memory or in the register file. This DLX-pro includes a combinational shifter and **behavioral** multiplier on 64 bits, plus two 32-bit special registers to store the product result.

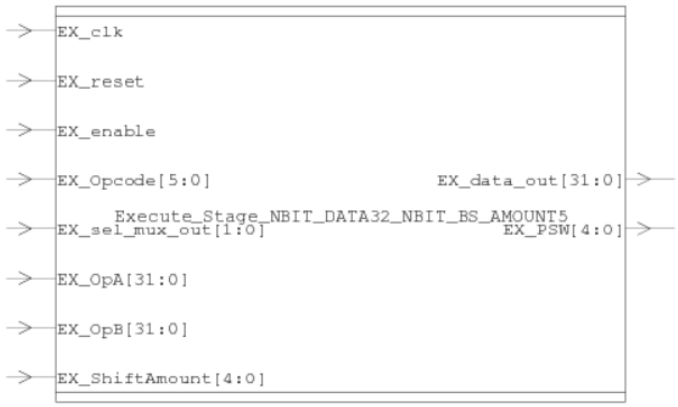

Figure 2.22: Execute stage pinout

On the left there are 4 enable interfaces. They are just rows of AND gates letting signals pass, toward functional units. Due to strong combinational nature of the execute stage, they avoid propagation when no arithmetical operation has to be performed. Then, there are 2 couple of them, one for ALU operands, the other for multiplier, both with the respective enable pin.
ALU executes addition, subtraction and shifting operations, moreover, bit-wise logic and comparison. Being an ALU, it generates a set of status bit, saved in the Process Status Word (PSW); in this project they are used just for debugging and are left floating. The output of EX is on 32 bits, multiplexed among ALU output, MLO and MHI registers.

Shifter

I seeked inspiration from the T2 version of the shifter register, as it's described in [3]. You can find a deeper explenation in ln-ms-cap4 file in the Documents directory.
Its microarchitecture is composed of three stages:

1. A first level of multiplexers, where all shifting by multiple of 2 are performed.
2. A second level of multiplexers makes the choice between masks. Selector takes into account the three most significant bits of BS_amount
3. A third level of multiplexers where inputs are computed by shifting by 0, then 1, 2, 3, ...7 the mask coming from the previous step. In the end, the output of this combinational shifter is choosen among those signals, where selector is formed by the 3 least significant bits of BS_amount

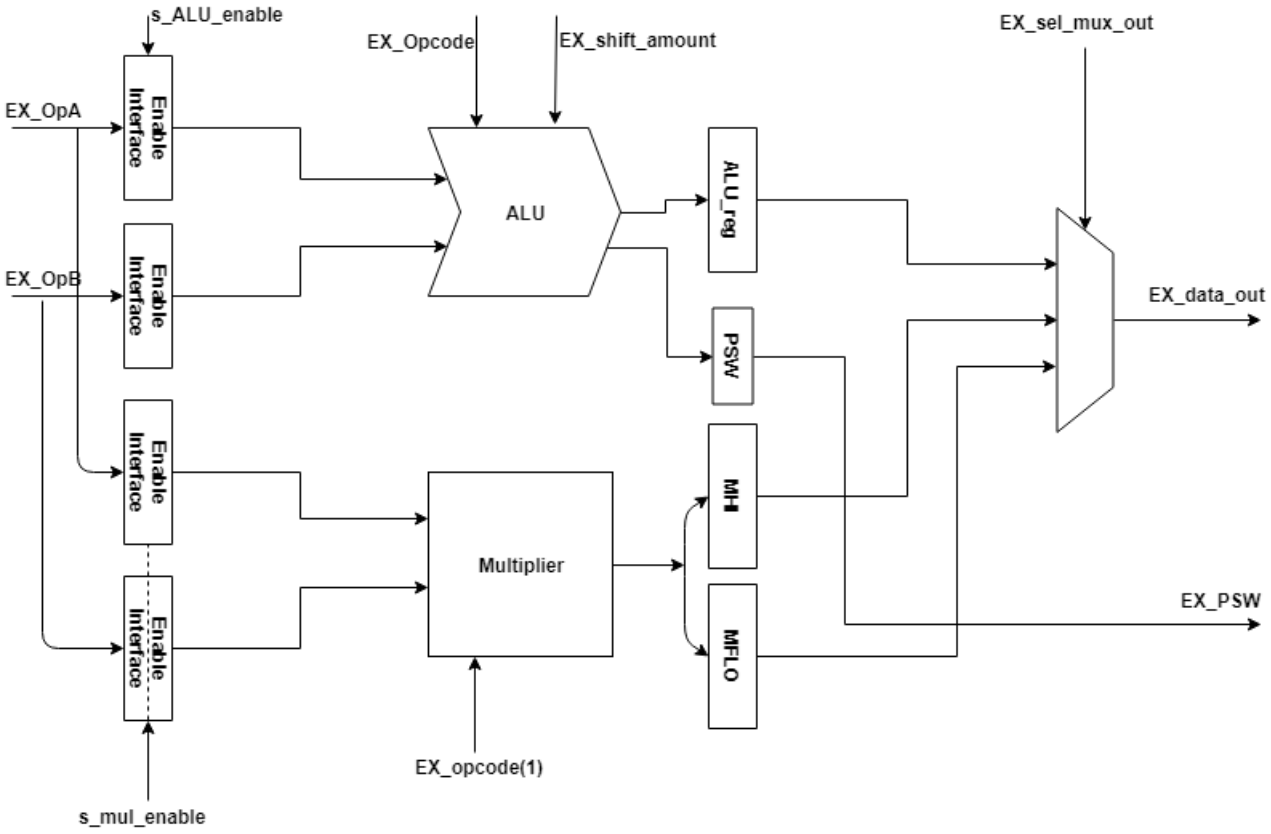

Figure 2.23: Execute stage microarchitecture

As you can see from ALU micrarchitecture picture, the shifter is on the same combinational path of P4 adder. I designed it thinking about what happens in modern microprocessor. There are instructions performing in the same clock cycle both shifting and adding operations. Starting from this fact, I inserted two new instruction to emulate this behavior. Unfortunately, I had to remove them because they didn't pass the testing phase and I hadn't enought time to redesign and fix.

P4 adder and Logic Unit

The P4 adder and the logic unit are totally designed taking into account features and specification described in [3]. You can find a deeper explenation and pictures in ln-ms-cap4 file in the Documents directory.

Comparison logic

Comparison logic checks if two numbers are equal, not equal, A lower than B, A greater than B, A lower than B or equal and A greater than B or equal. Finally, a result on 32 bits is produced. This component has 2 areas:

- A comparator generating 5 1-bit signals (eq, lt, le, gt, ge)
- A combinational network to adapt the result of comparations to 32 bits. By the VHDL point of view, this part has been realized in a behavioral way by the usage of a process.

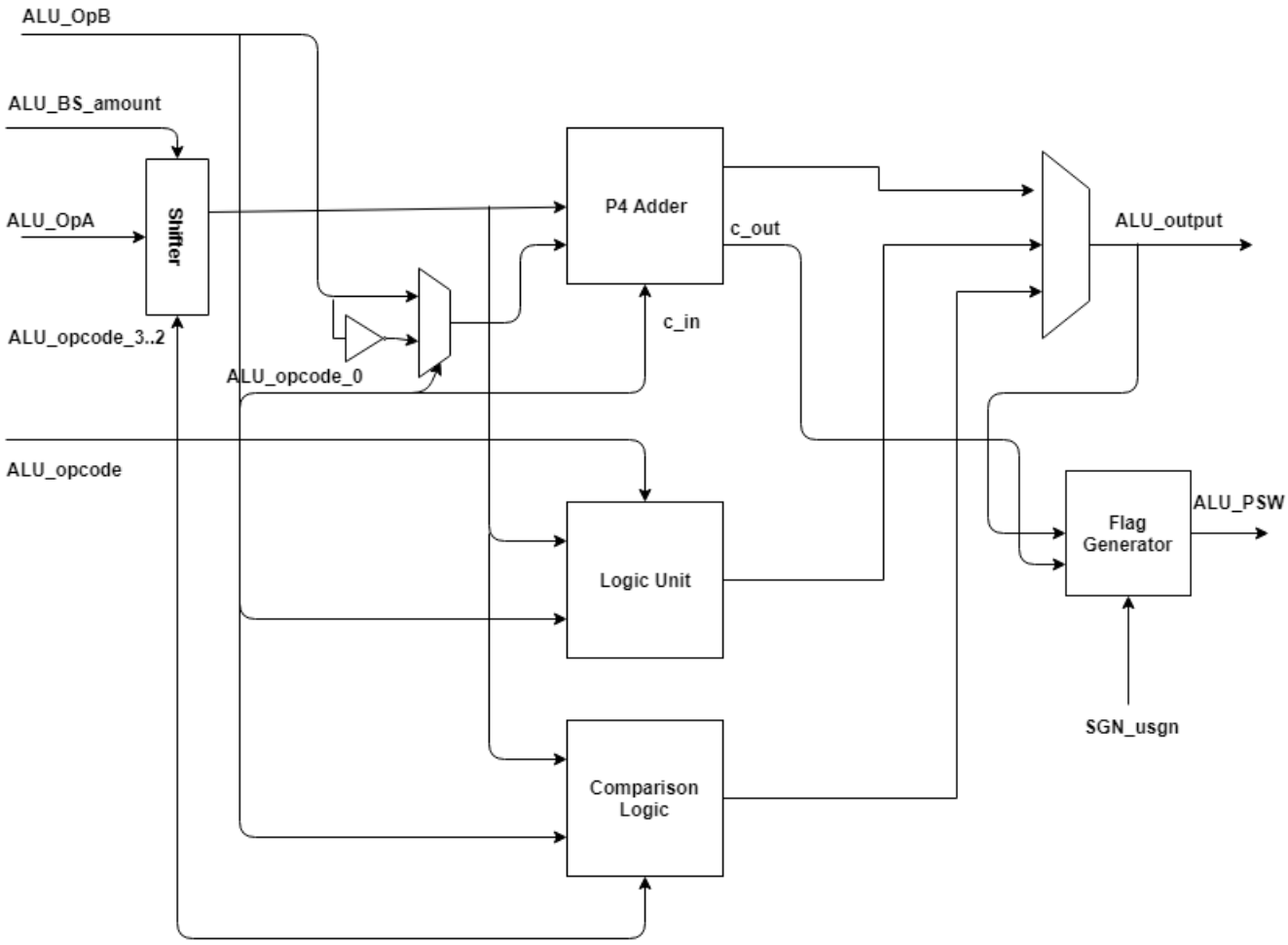

Figure 2.24: ALU microarchitecture

The 32-bit comparator is made up of a chain of simple one-bit priority comparators. The most siginifant one, referring to the two most significant bits, enables the next one if the result of the comparison is not equal, concurrently checks A_N lt B_N and A_N gt B_N. The second comparator raises and verifies $A_{N-1} = B_{N-1}$, $A_{N-1} < B_{N-1}$, $A_{N-1} > B_{N-1}$; if $A_{N-1} \neq B_{N-1}$ the third entity of the chain starts working, and so on to the least significant one. I adopted this priority based strategy in order to avoid useless propagation of signals when it's possible to determine, at the beginning for instance, the comparison result.

Flag generator

All microprocessor use a set of control signal in order to summarize the status of the machine at each clock cylce. They assume the same meaning of the systems state variables. I selected a subset of status flags [`https://it.wikipedia.org/wiki/Registro_di_stato`]:

- **ZF**: Zero flag indicates whether the result of a mathematical or logical operation is zero. Computed by OR-ing all bits

- **PF**: Parity flag is 1 whether the number of '1' in mathematical or logical result is even. Computed by XNOR-ing all bits

- **SF**: Sign flag coincides with the MSB of the result after an arithmetic operation when is the result of a signed operation, 0 otherwise
- **CF**: Carry flag indicates whether the result of an operation produces a non-limiting response in the bits used for the calculation
- **OF**: Oveflow flag indicates if the result of an operation is overflowed, according to the two's complement representation. It is similar to the carry flag, but is used in operations where the sign of operands is taken into account

Multiplier, MHI and MLO

For this area I got ideas from the miniMips architecture, which is based on a 32-bits one, like the DLX. This microprocessor owns a multiplier on 64 bits and the product can be accessed by special instructions managing separately the highest half from the lowest one. In fact, I changed the binary format of `mult` and `multu` in order to adapt them to the miniMIPS one, and I introduced two special instructions: `mfhi` and `mflo`.
Multiplier has been described behaviorally, leaving up to the synthetizer freedom to improve the design. Moreover, Booth's multiplier has a very low performing, that could be improved by implementing multicycling; however it requires additional hardware and structural hazard management making the design more complex.

Opcode analysis

I spent some times in the choice of bits making up the ALU_opcode. In fact, I didn't assign randomic value, but each bit holds a specific meaning. This strategy lowers the number of interconnection and avoids the usage of decoders to drive control signals toward each functional unit. ALU_opcode(5) and ALU_opcode(4) are the most important bits. They are associated to functional units.

ALU_opcode(5)	ALU_opcode(4)	Functional unit
0	0	Shifter and Adder
0	1	Logic unit
1	0	Comparison logic
1	1	Multiplier

Table 2.9: ALU_opcode(5) and ALU_opcode(4) meaning

Depending on the functional unit under work, the structure of the opcode changes.

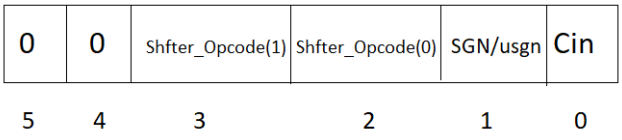

Figure 2.25: Shifter and adder ALU opcode

When a sum/subtraction is going to be executed, usually bits 3 and 2 are kept to 0. However, it's possible to combine shifting and sum/subtraction, performing the former at the beginning and using the output to complete the latter.

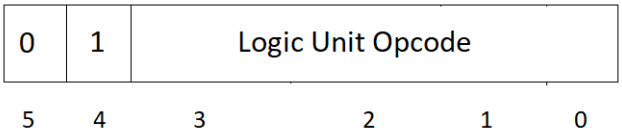

Figure 2.26: Logic unit ALU opcode

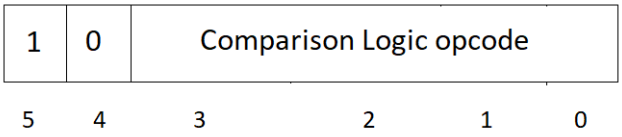

Figure 2.27: Comparison logic ALU opcode

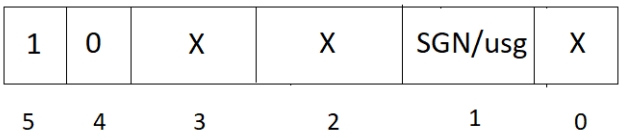

Figure 2.28: Multipliare EX opcode

You can find a full list of opcodes in EX_control_signals in Documents directory.

2.4.4 Memory

The MEM is the fourth pipeline step. It provides signals to DRAM in order to perform opeartion of loading and saving.

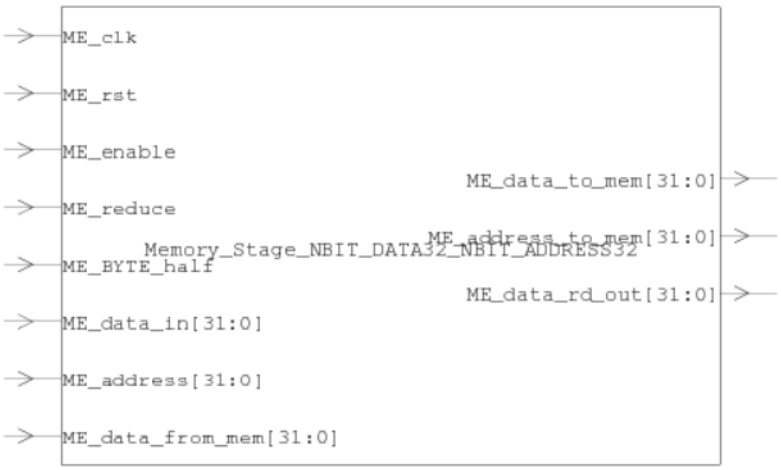

Figure 2.29: Memory stage pinout

With respect to the previous stages, this one and the write back are really simple. Here signals are driven from and to the Data RAM. In case of particular instructions, such as: `lb`, `lh`, `lbu` and

`lhu`, data reducer triggers, handling the 32-bit data coming from the DRAM and reducing them to a signed/unsigned byte/halfbyte.

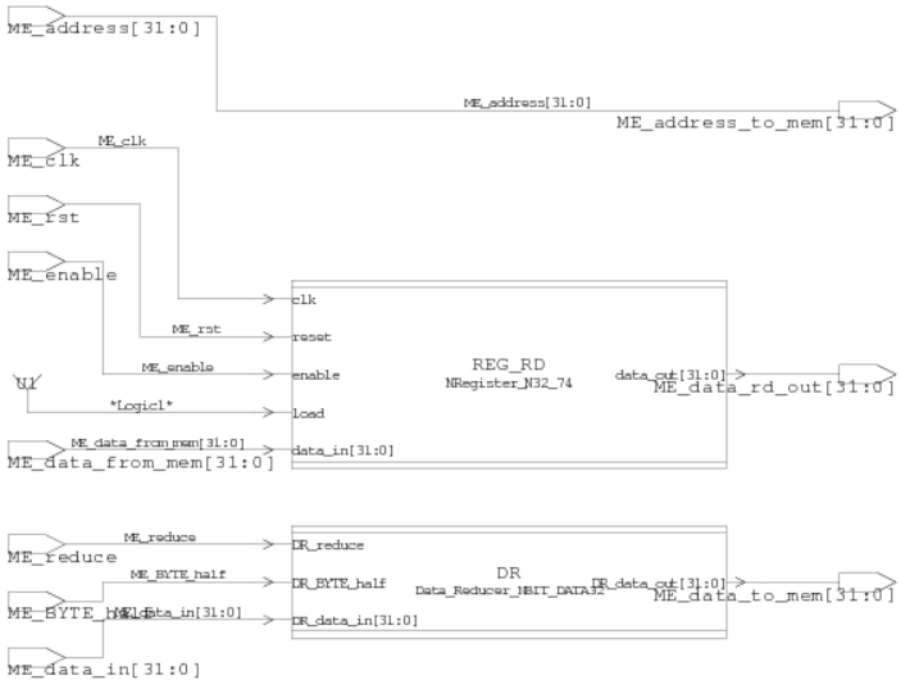

Figure 2.30: Memory stage microarchitecture

2.4.5 Write back

The last block of the pipeline aims at storing in the register file the data, which is the result of either an internal operation or a store one.

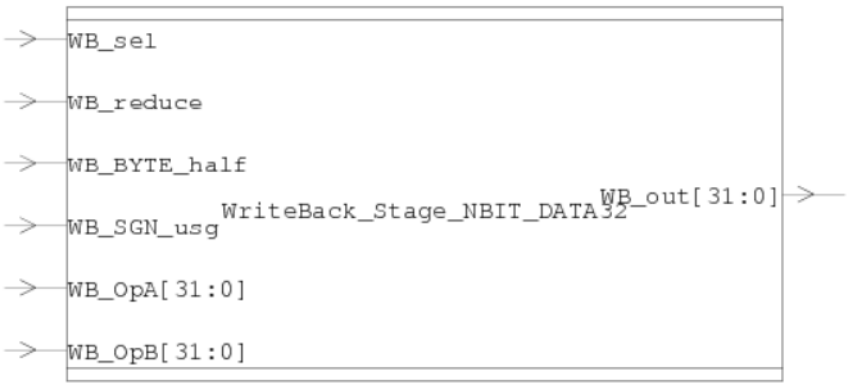

Figure 2.31: Write back stage pinout

In case of particular instructions, such as: `sb` and `sh` data reducer triggers, handling the 32-bit data going to the DRAM and reducing them to a signed/unsigned byte/halfbyte.

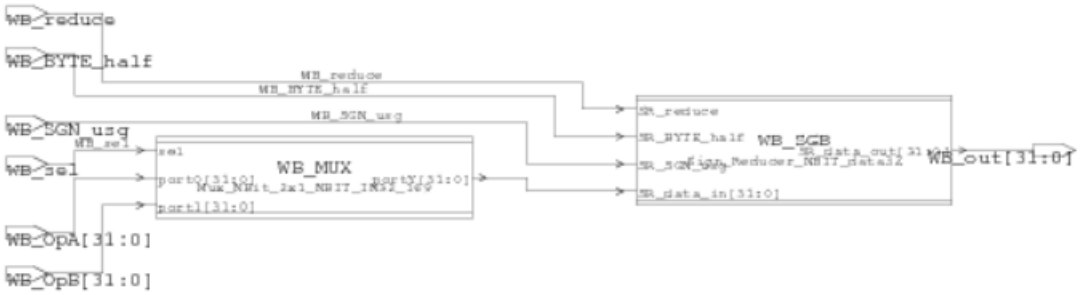

Figure 2.32: Write back stage microarchitecture

2.4.6 Forworing control unit

This DLX implements forworing logic, so a circuitry managing multiplexers selectors between 2 consecutive stages is required.

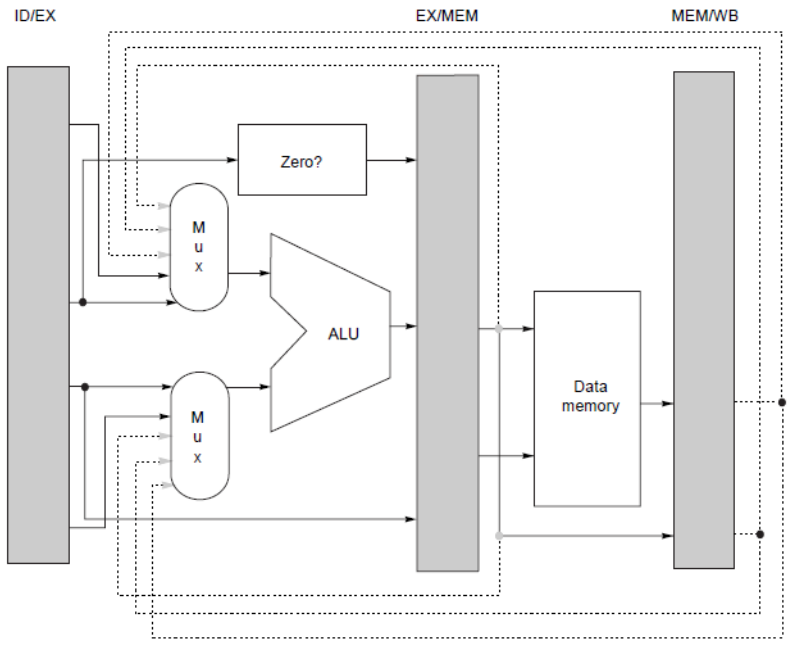

Figure 2.33: Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs [1]

Forwording logic is used to manage data hazards, in order to avoid stalls or wrong data. However there are some cases where a stall must be inserted, like a load followed by one instruction reading. In according to Patterson's description, in order to design a forwording control unit, it's necessary to perform a set of comparison between register addresses coming from IR fields. Since there are different type of instruction, where the $i-th$ bits of source and destination register are

not the same, this component should also recognize what typology of operation is at each step. Follows a table summorizing all comparations [1].

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR$_{16..20}$ == ID/EX.IR$_{6..10}$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR$_{16..20}$ == ID/EX.IR$_{11..15}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR$_{16..20}$ == ID/EX.IR$_{6..10}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR$_{16..20}$ == ID/EX.IR$_{11..15}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR$_{11..15}$ == ID/EX.IR$_{6..10}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR$_{11..15}$ == ID/EX.IR$_{11..15}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR$_{11..15}$ == ID/EX.IR$_{6..10}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR$_{11..15}$ == ID/EX.IR$_{11..15}$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR$_{11..15}$ == ID/EX.IR$_{6..10}$
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR$_{11..15}$ == ID/EX.IR$_{11..15}$

Figure 2.34: Forwarding of data to the two ALU inputs (for the instruction in EX) can occur from the ALU result (in EX/MEM or in MEM/WB) or from the load result in MEM/WB [1]

From the VHDL point of view, it's totally described by 3 behavioral processes

- Instruction analyzer: for each stage, FCU identifies the type of instruction (reg, imm, load, store, jmp and branch)
- Multiplexer selectors generator: receiving data from instruction analyzer process, it's able to drive forwording multiplexer in the datapath, in according to the content of the table above
- Stall process: it recognizes stall situation and sends a signal to the main control unit

Unfortunately after testing some bugs raised. **Failures occur when forwording interests branch or jump instructions**. I found two possiple solution could be adopted: an hardware and a software. The former expected rewrting VHDL code of forwording control unit and retesting again all assembly codes, the latter leaves to compiler the taskt to recognize critical scenarios and insert nop.

2.5 Control unit

This control unit is the main brain of the whole DLX. It's **hardwired**, it means that is composed of a big instruction decoder, which is implemented by a big VHDL process, and a chain of register in order to emulate the datapath pipeline.

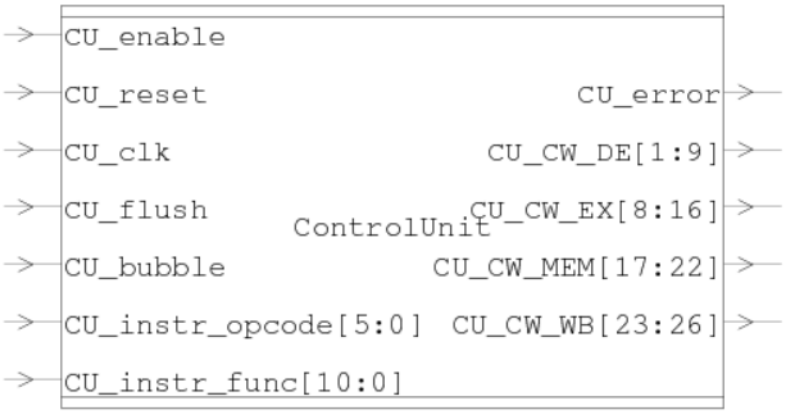

Figure 2.35: Control unit pinout

Signal	From	To	Note
CU_instr_opcode	IRAM	-	$IR_{31..26}$
CU_instr_func	IRAM	-	$IR_{10..0}$ Considered for reg-type instructions
CU_flush	Datapath, BTB	-	It's result of a boolean expression among DP_branch_taken, DP_restore and BTB_predition. A flush of the whole pipeline is required
CU_bubble	Datapath	-	Insertion of a NOP has been required
CU_error	-	-	Just for debugging, useless
CU_CW_DE	-	Datapath	Control word to decode stage
CU_CW_EX	-	Datapath	Control word to execute stage
CU_CW_MEM	-	Datapath	Control word to memory stage
CU_CW_WB	-	Datapath	Control word to write back stage

Table 2.10: Control unit signals

There are 26 control signals: 9 to decode, 9 to execute stage (2 are reused from the decode control word), 6 to memory stage and 4 to write back. In Documents directory there is ControlWords excel file, which lists in details all control words.

Rd1	Rd2	Wr	JMP_brch	SignextDE	HMBS_sel	EX_op	EX_enabl	EXsel_out	RD_wr	DRAM_en	MEM_red	BYTE_half	WB_sel	WB_red	Byte_half	SGN_usg
1	2	3	4-5	6-7	8-9	10-15	16	17-18	19	20	21	22	23	24	25	26

Figure 2.36: Control word structure

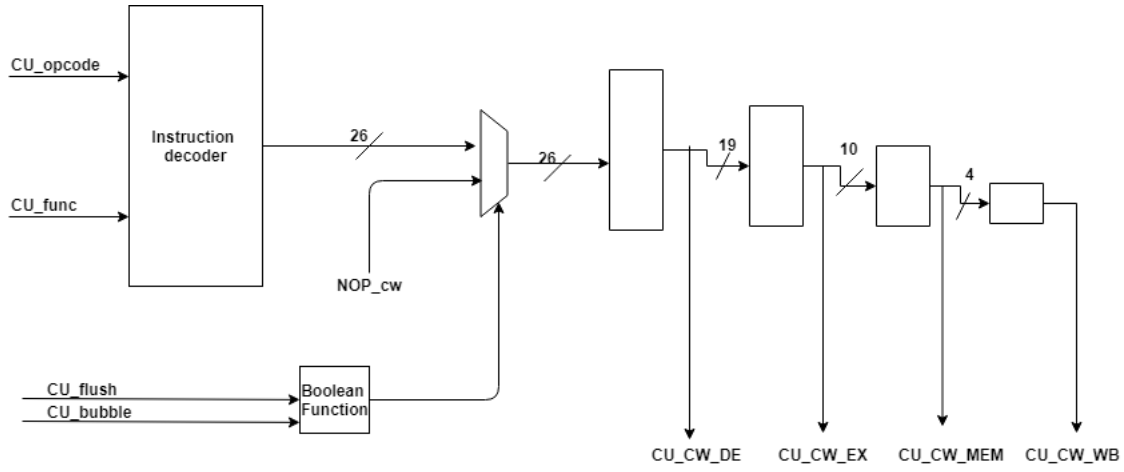

Figure 2.37: Control unit microarchitecture

Chapter 3

Simulation

I used 24 assembly codes to test the DLX. In order to run a simulation, it's necessary to write the path of the .asm.bin file in b-IRAM.vhd, here at line 46. Then you can use every simulator you want. I liked *ISim 2013*, as it's simple to use and very intuitive.
List of codes and functional units are involved into follows.

Code	**Functional units**
all_general_test	P4 adder, Logic unit, Shifter, Comparison logic, Multiplier, MHI and MFO
branch	Logic unit, FCU, BTB
branch_prediction_bench	Shifter, BTB
dlx_div	Logic Unit, Shifter, BTB
forwording	FCU
jr, j, JumpAndLink	FCU,Comparison logic, logic unit, jump
lhi	Shifter
LoadStore	DRAM
Mult0, Mult1, Mult3	Multiplier, MHI and MFO, Shifter, FCU, BTB(no Mult3)
rtype	Comparison logic
shiftAndImm	Shifter
test_arithmetic, test_arithmetic_comments	P4 adder, Shifter, Comparison logic, Logic unit
test_memory_no_hazard	Load reducer, Store reducer
test_pro_ins	P4 adder, Logic unit, Shifter, Comparison logic, Multiplier, MHI and MFO
xor_swap	FCU
fibonacci	BTB
fibonacci_cmplx	BTB, FCU, jump, Comparison logic
prodmatrix2x2	jump, Multiplier, MLO
vector_mul	Multiplier, MLO, Comparison logic, BTB, FCU

Table 3.1: Control unit signals

This DLX requires 2 `nop` instruction to start, so they must be reported every time.
For some codes I collected information on the execution time, in order to highlight the advantage of the dynamic branch prediction in a microprocessor system. With an average benefit around 11.43%

Code	CC BTB off	CC with BTB on	Saving [%]
fibonacci	404	351	13
branch	806	711	11.78
dlx_div	84	76	9.52

Table 3.2: Number of saved clock cycle by using BTB

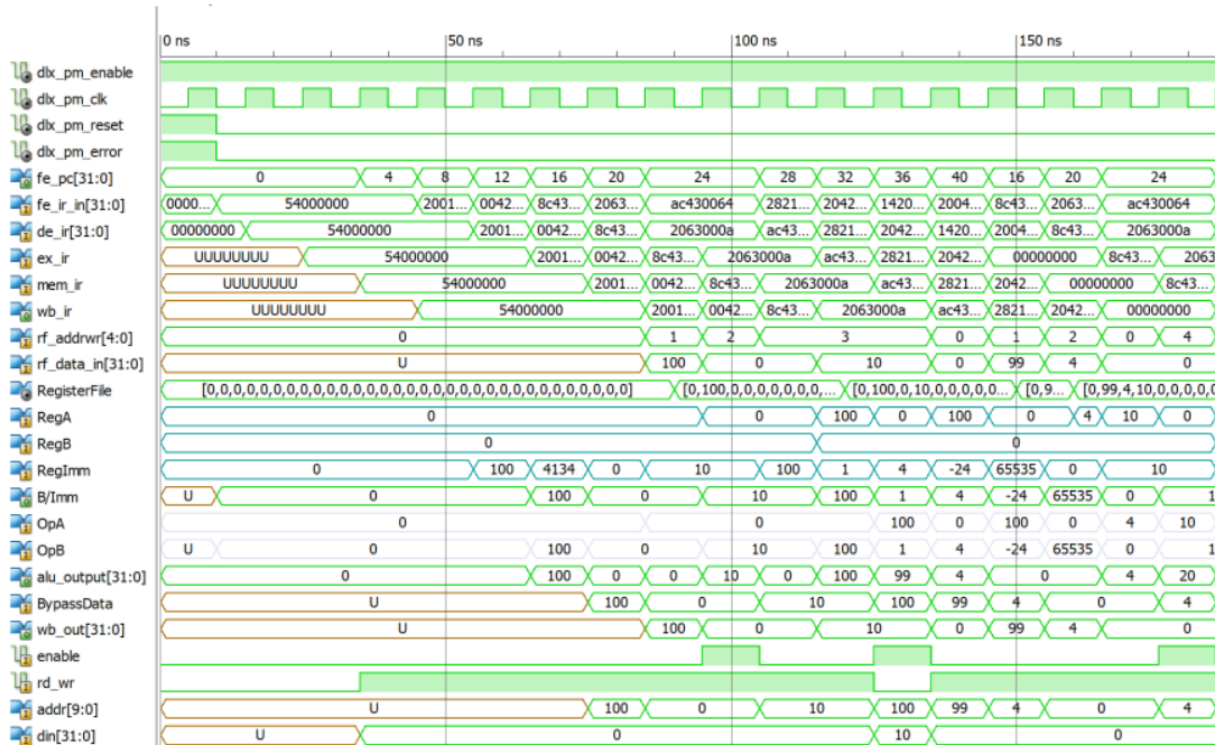

Figure 3.1: Datapath waveform executing Branch.asm

Just have a look over signals. You can immediate recognize the PC (*fe_pc*), which increments by 4 at every clock cycle. Then the IR propagation is shown by *fe_ir_in*, *de_ir*, *ex_ir*, *mem_ir* and *wb_ir*. You can notice the RF and the writing address and data, which arrive at the write back. At the DE data are prepared in *RegA* *RegB* and *RegImm*. EX reads operands *OpA* and *OpB* and computes the output on *alu_output*. Memory stage, for classic instruction, bypass data through *BypassData* register, otherwise *enable* DRAM, *rd_wr*, *addr* and *din* are involved to communicate. WB just leads data *wb_out* to register file.

Chapter 4

Synthesis and Physical layout

These are the last steps of the design flow. The provided library to synthesize is the **NanGate Open Cell Library**, with a technology process of 65 nm and one Single Threashold Voltage, in nominal conditions at 25řC I collected data from some blocks of the system, every time gathering timing, area, and power consuption information.
In the end, my DLX is able to run at a nominal frequency of 174 MHz, due to a critical path containing the behavioral multiplier.

Module	**Timing [ns]**	**Area** [um^2]	**Dynamic Power consuption [mW]**	**Leakage Power consuption [uW]**
DLX_core *(post_syn)*	5.74	59009	7.98	1216.2
Datapath	3.67	33495	11.77	673.78
Control_unit	0.41	700	0.577	15.17
BTB	1.91	23600	17.50	490.33
ALU	4.77	1231	3.30	31.06

Table 4.1: Synthesis result

In the end, I generated the post synthesis vhdl and verilog, which will be processed by **Cadence innovus** to perform the final physical layout. From the following pictures, it's possible to understand the real proportion among macroblocks making up the DLX

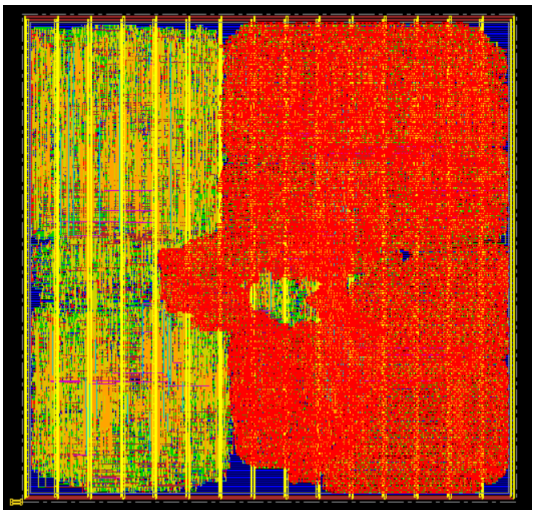

Figure 4.1: Datapath

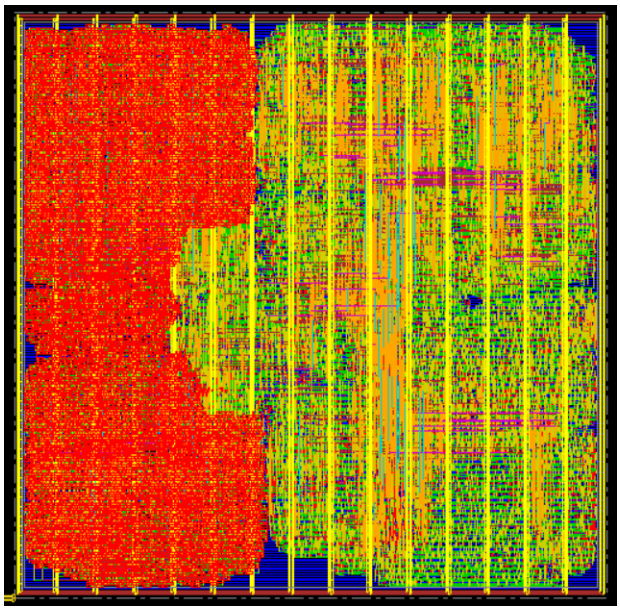

Figure 4.2: BTB

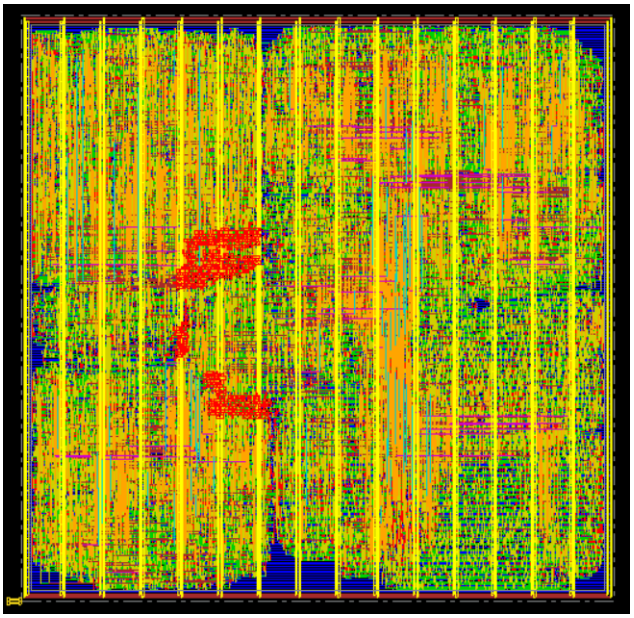

Figure 4.3: BMM

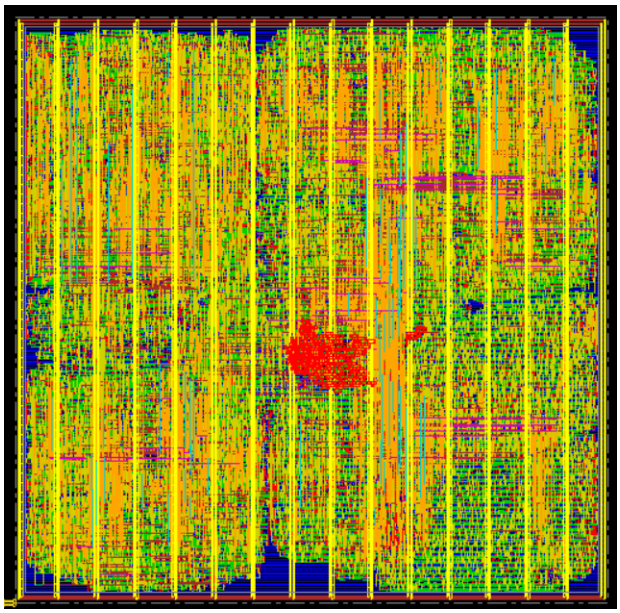

Figure 4.4: Control unit

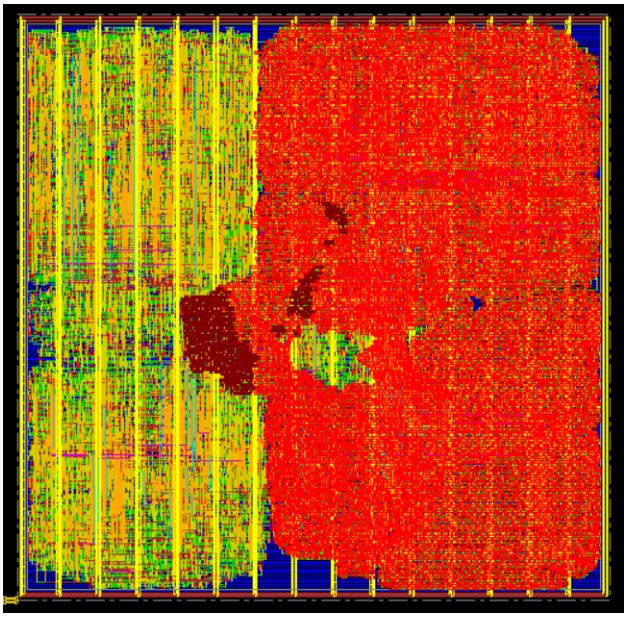

Figure 4.5: Fetch in Datapath

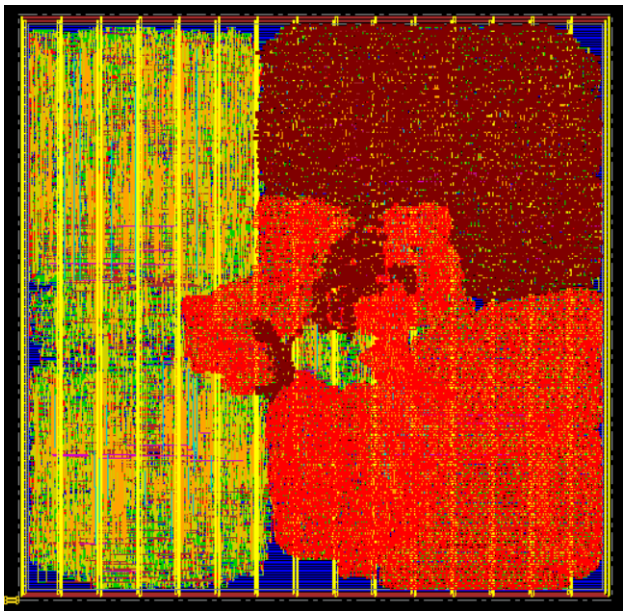

Figure 4.6: Decode in Datapath

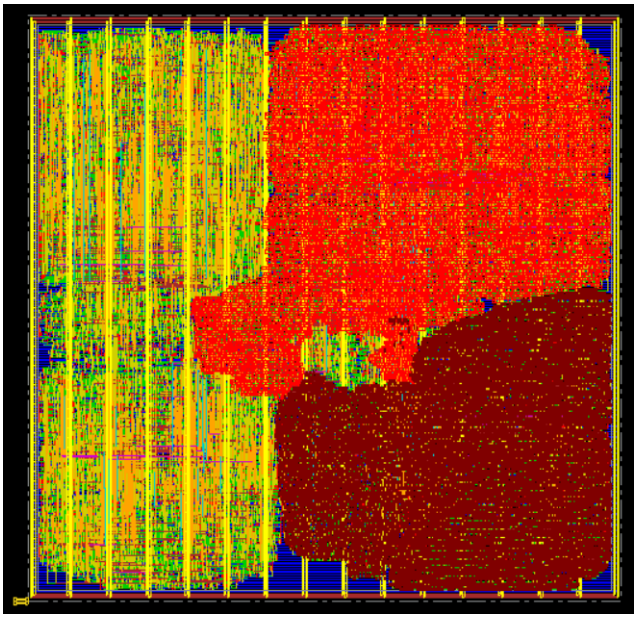

Figure 4.7: Execute in Datapath

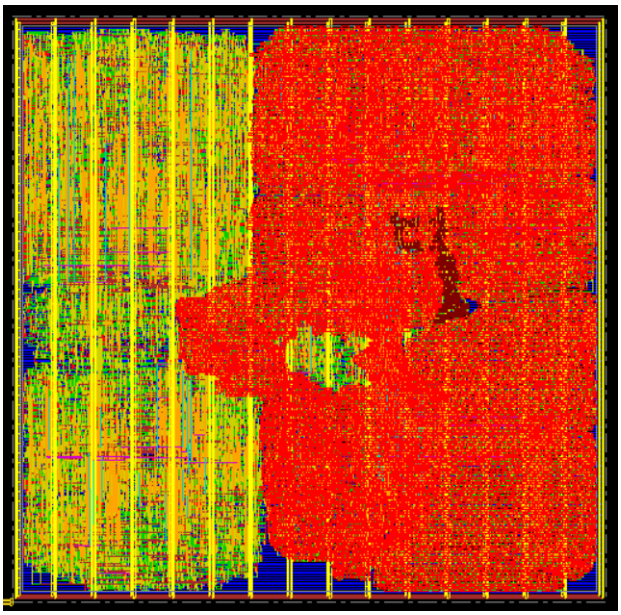

Figure 4.8: Memory in Datapath

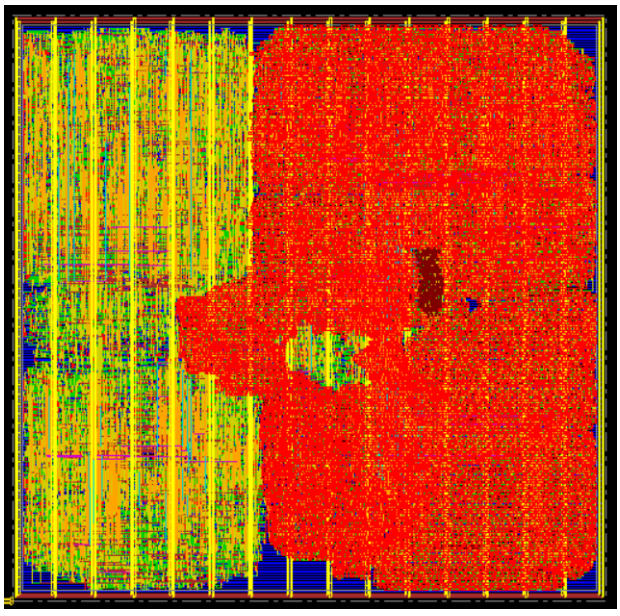

Figure 4.9: Write Back in Datapath

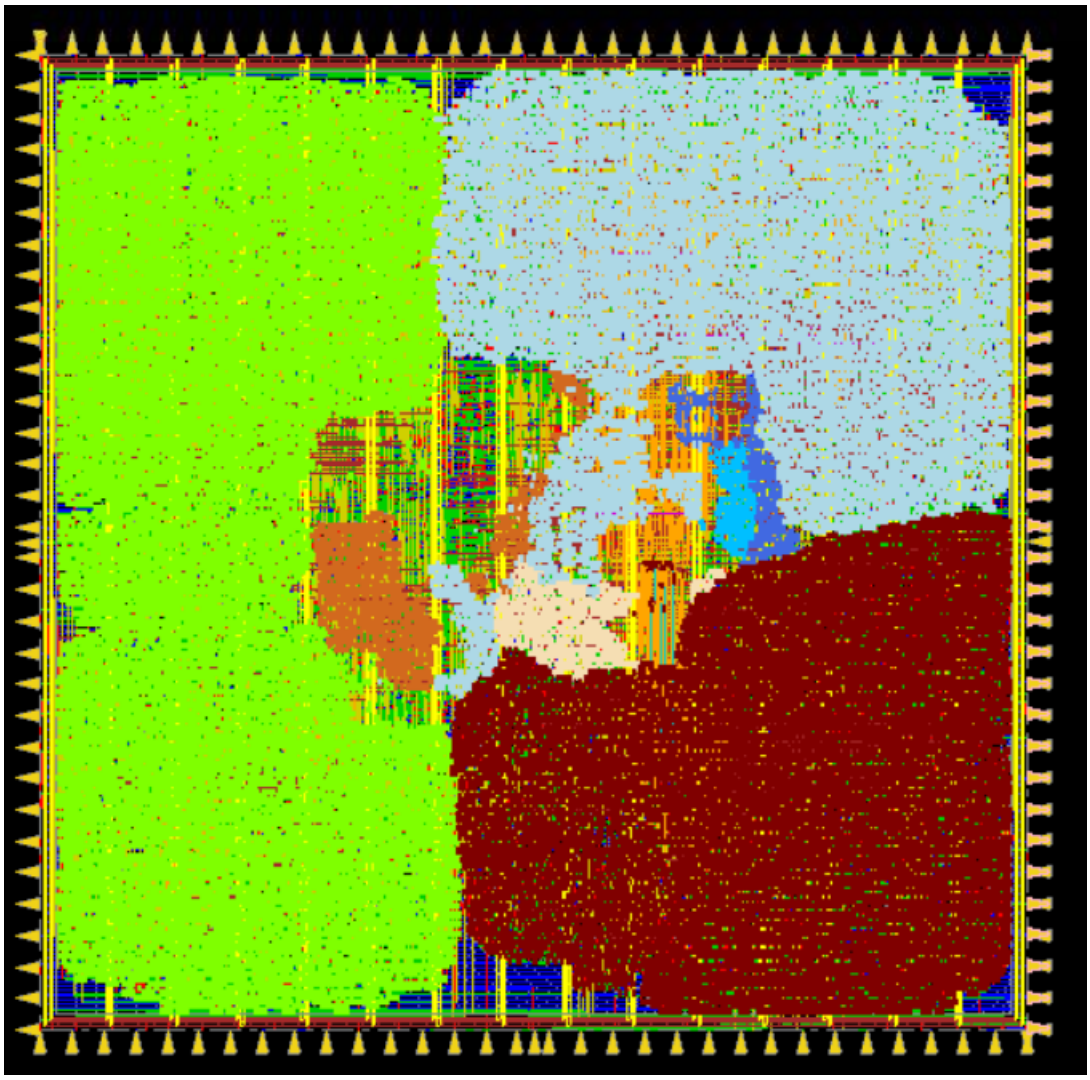

Figure 4.10: Final DLX physical design

Bibliography

[1] *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2011.

[2] *Ananlog and digital electronics for embedded systems*, chapter 2. CLUT, 2015.

[3] Mariagrazia Graziano. Microelectronic systems lecture notes. chapter 4.

[4] Giovanna Turvani Mariagrazia Graziano, Giulia Santoro. Design and development of dlx microprocessors. Microelectronic Systems Final Project Specification.