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 15, 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 [3], 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**

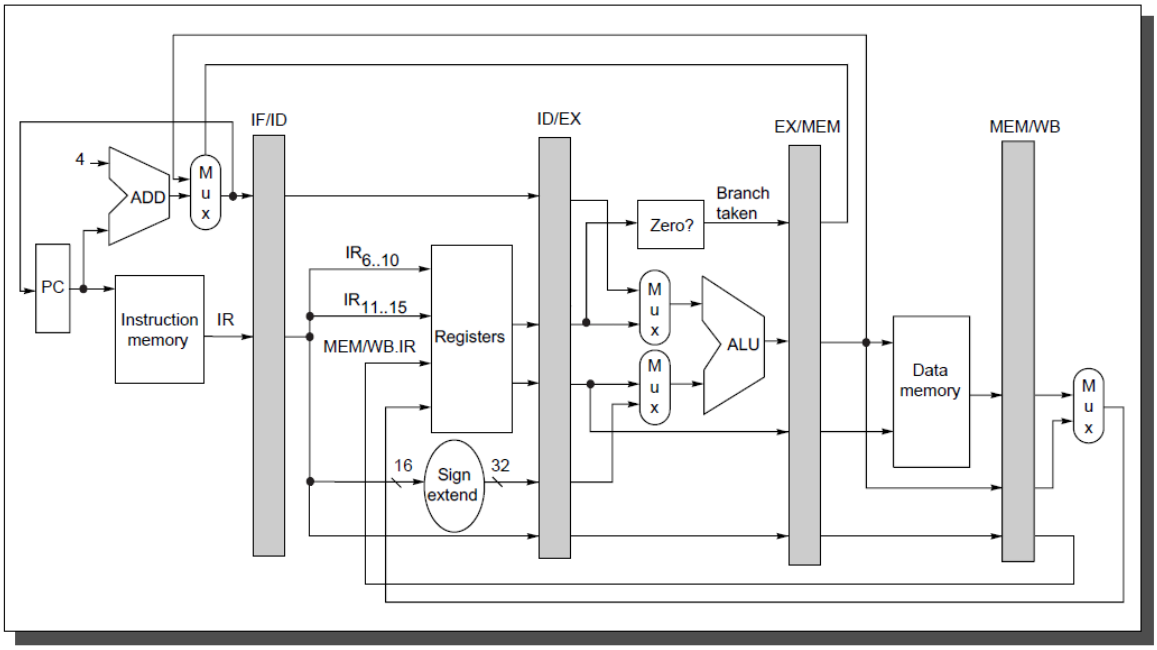

Figure 1.1: Basic DLX Datapath. Figure 3.4 from [1]

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 [3].

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

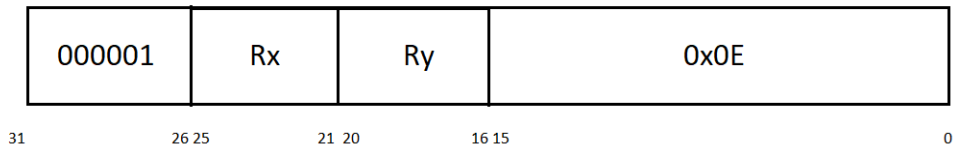

Figure 1.2: Mult format

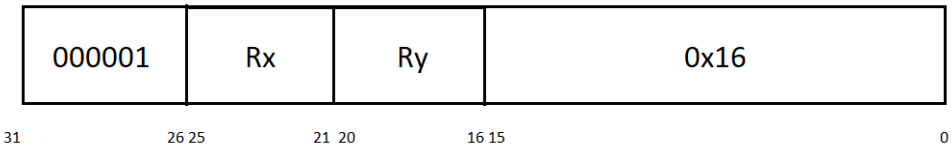

Figure 1.3: Multu format

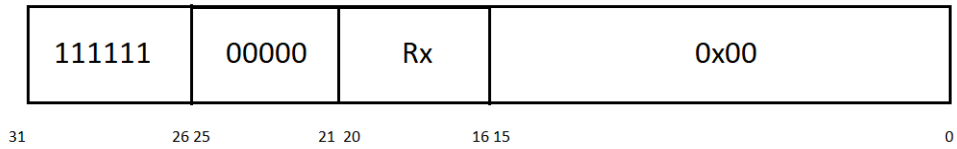

Figure 1.4: Mflo format

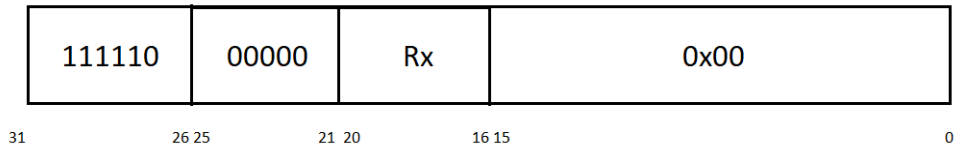

Figure 1.5: Mfhi format

More information on Basic and Pro DLX instruction in [3].

Chapter 2

Register Transfer Level

2.1 Core

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

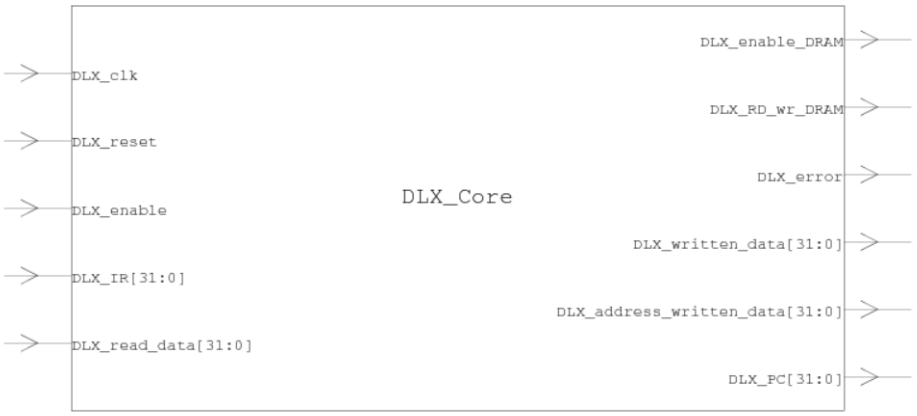

Figure 2.1: Core pinout

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

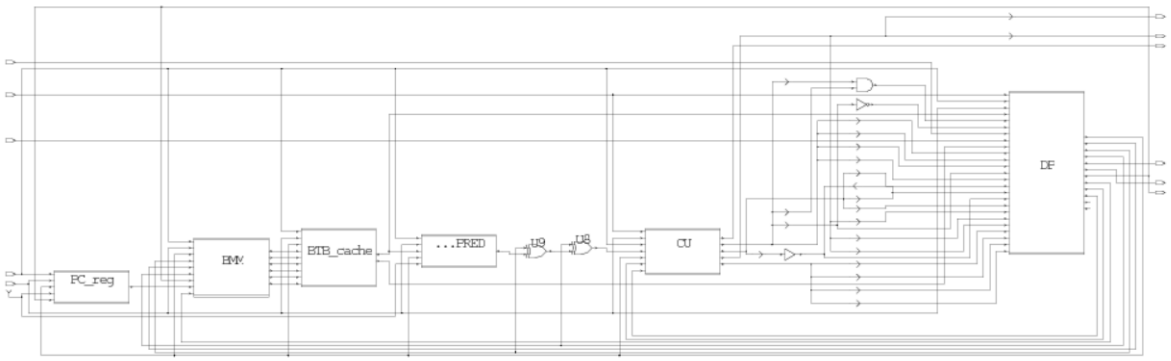

Figure 2.2: Core schematic - I

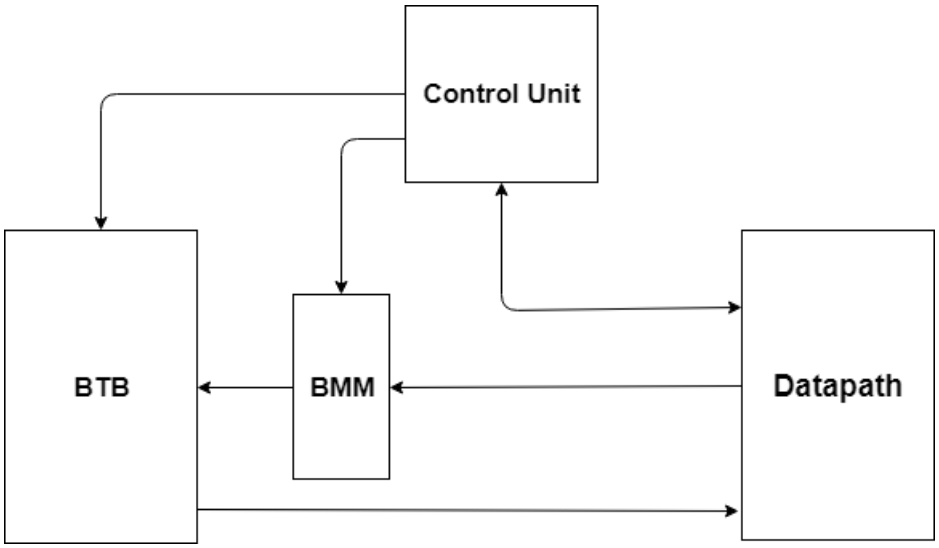

Figure 2.3: Core schematic - II

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.

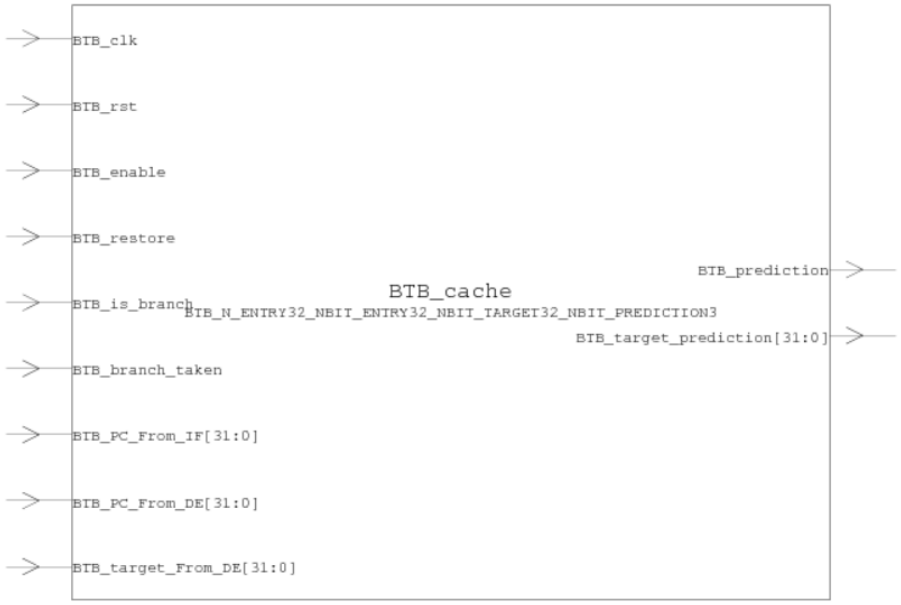

Figure 2.4: BTB pinout

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 provide 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 lead 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.

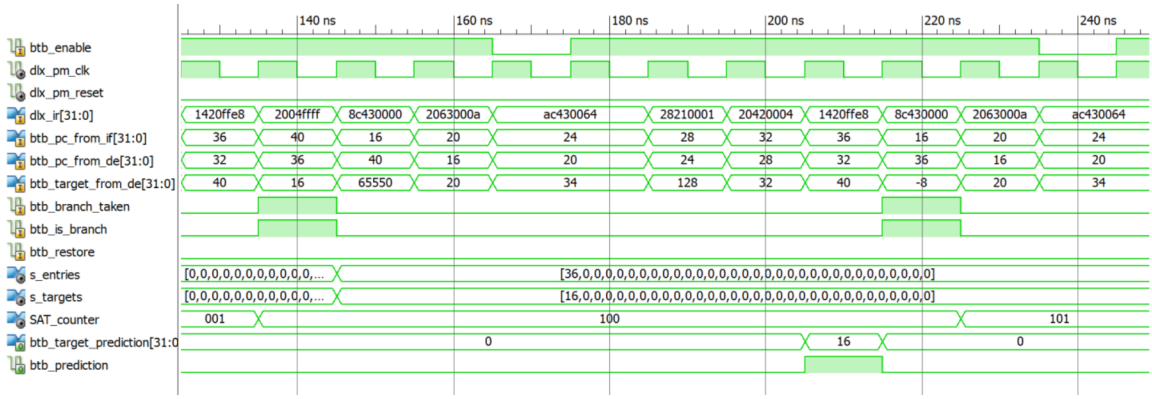

Figure 2.5: BTB waveform I

Restoring

Execution goings 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: $\phi_1 = 111_2 \rightarrow \phi_2 = 111_2 \rightarrow \phi_3 = 110_2 \rightarrow \phi_4 = 101_2$

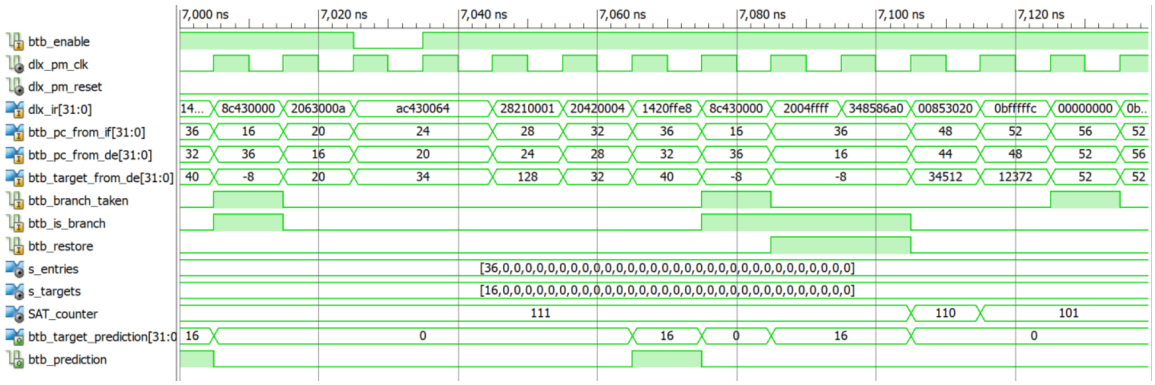

Figure 2.6: BTB waveform II

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

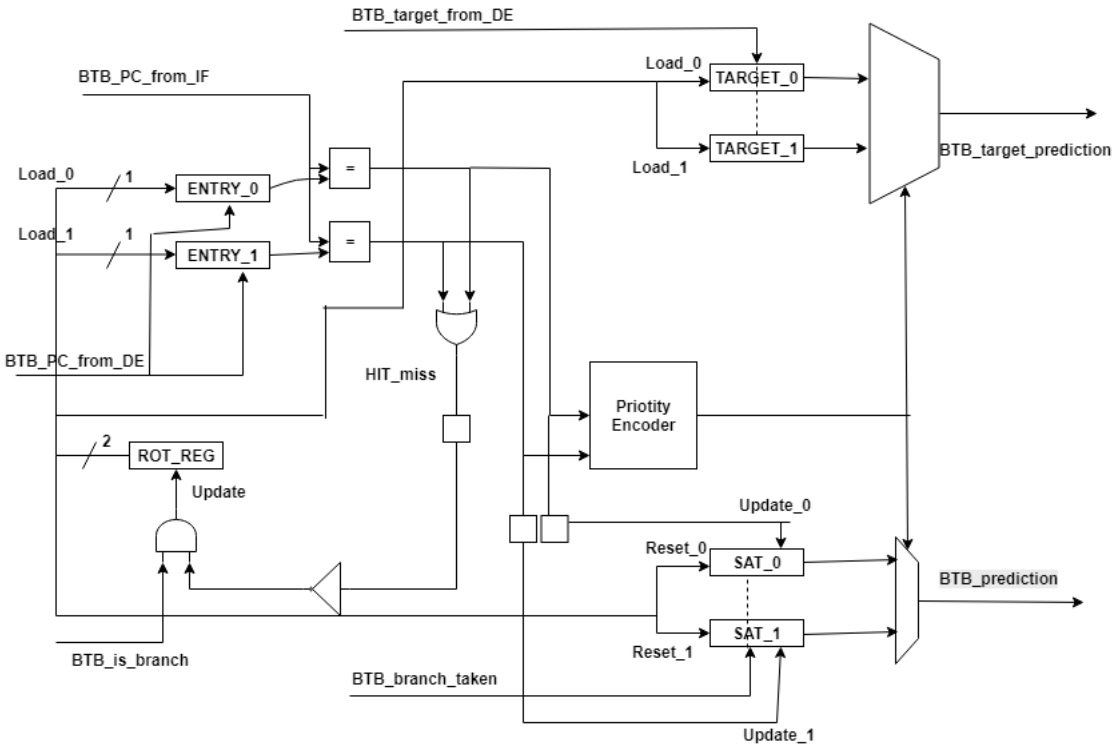

Figure 2.7: BTB simplyfied microarchitecture

2.2.1 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.

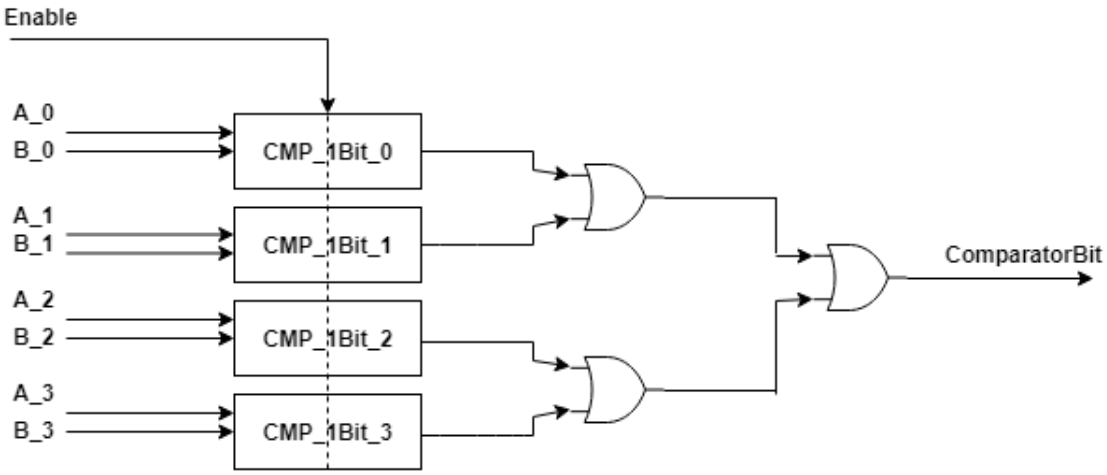

Figure 2.8: Comparator 4 bits schematic

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:

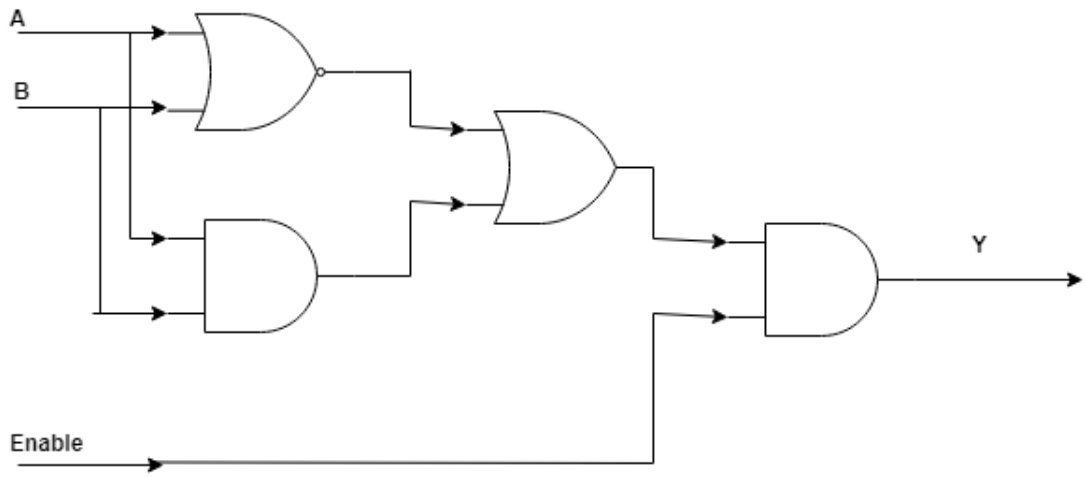

Figure 2.9: Comparator 1 bit schematic

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

2.2.2 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 are allowed. That single bit to 1 is the pointer I'm referring to.

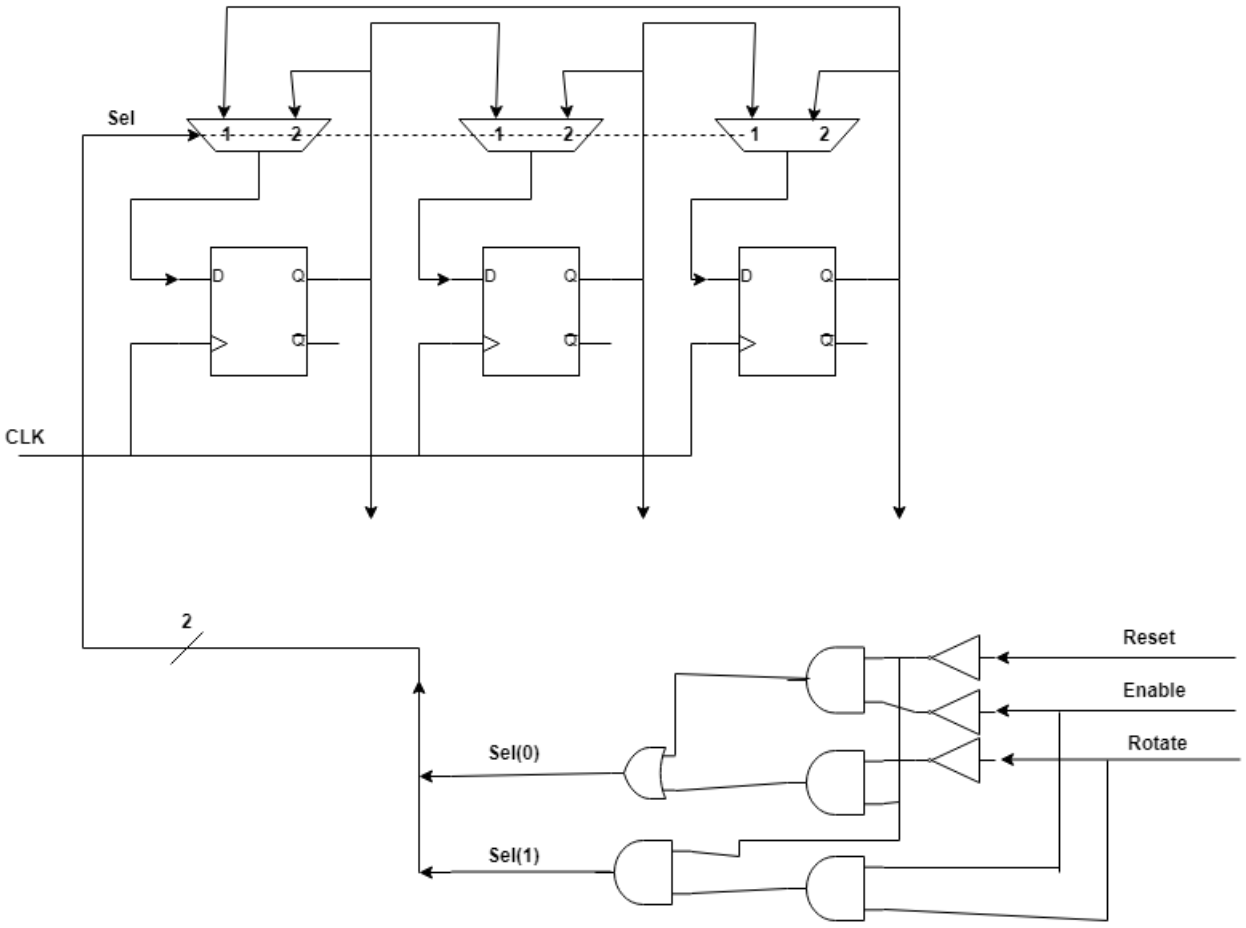

Figure 2.10: Rotate register schematic

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

2.2.3 Saturation counter

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

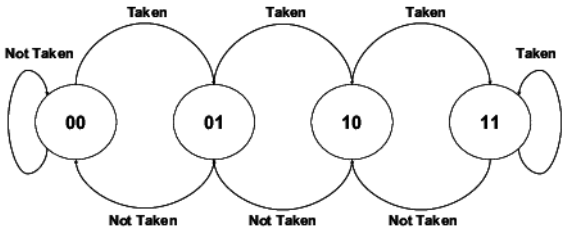

Figure 2.11: 2-bit SAT counter FSM

The usual architecture of a FSM is composed of a control unit and a datapath. This components ows this structure, where 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.

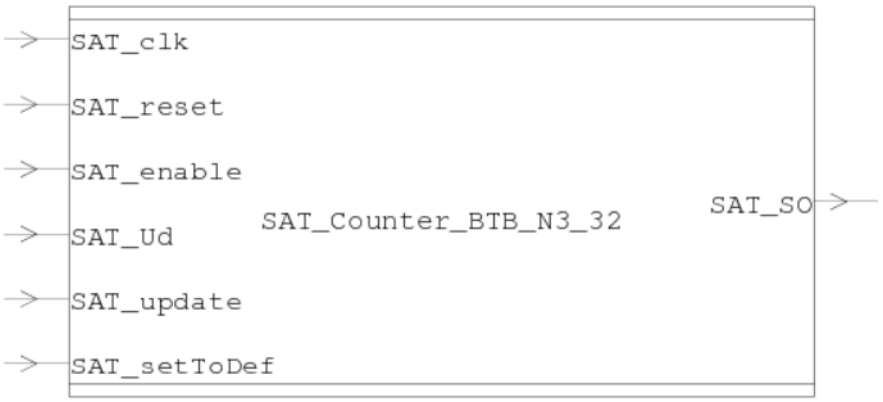

Figure 2.12: SAT counter pinout

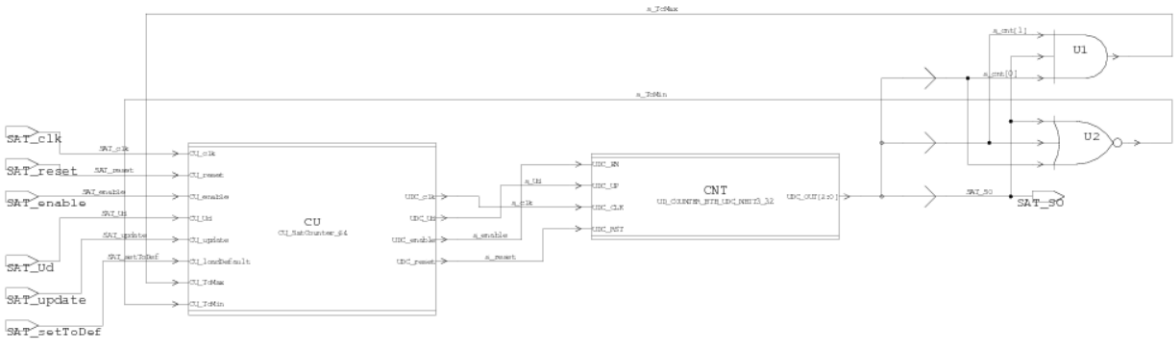

Figure 2.13: SAT counter microarchitecture

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

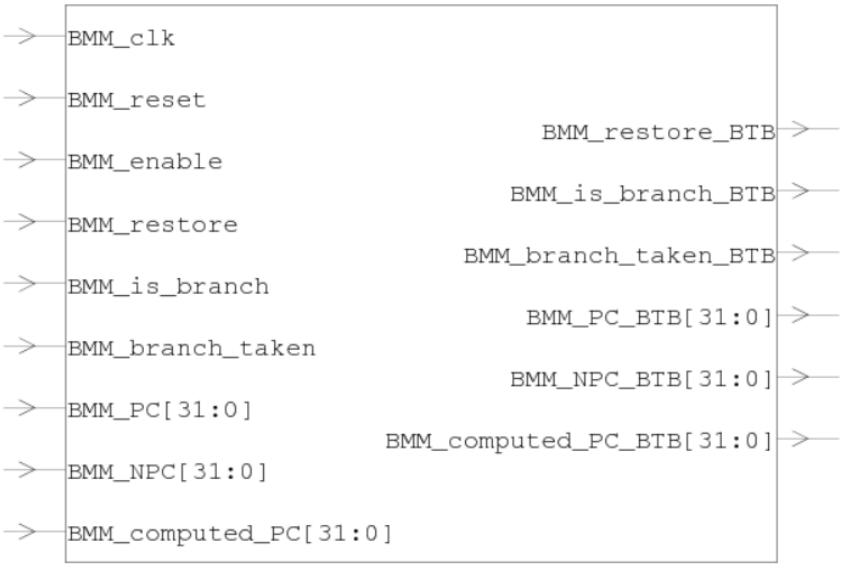

Figure 2.14: BMM pinout

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.

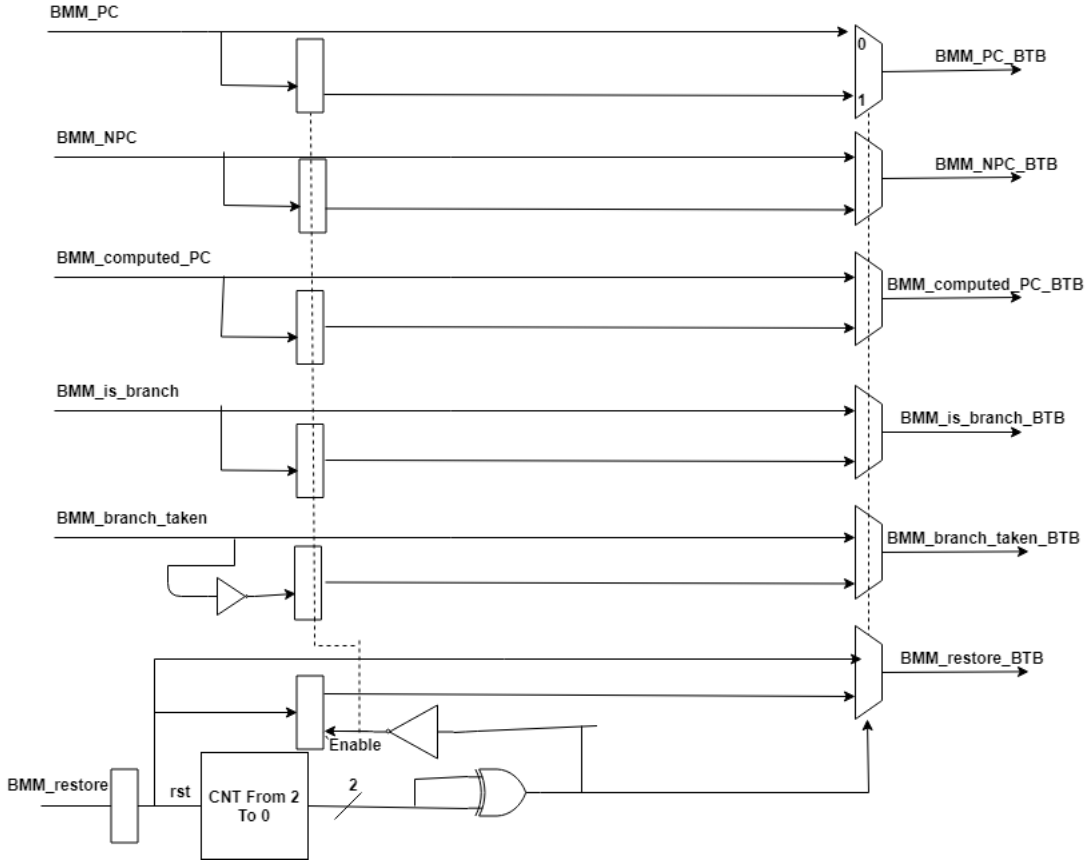

Figure 2.15: BMM microarchitecture

Bibliography

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

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

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