Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group 19

Ieva Antonia, Palmieri Giorgio

September 9, 2018

Contents

CHAPTER 1

Introduction

The purpose of this project is to implement a processor based on the DLX architecture. The digital design requires different phases of design : RTL, logical and then physical design.
This report introduces the processor architecture and provides a technical analysis of the main components and used procedures.

1.1 Main features

The DLX is a RISC processor architecture and it is essentially a simplified MIPS CPU (an educational version). The implemented microprocessor has the following characteristics:

- Hardwired Control Unit;
- Maximum working frequency : $\simeq 160MHz$;
- ALU for integer operations;
- Register File windowed that allows the usage of different subroutines in the assembly code;
- Extended Instruction Set for integer operations;
- Five-stages pipelined processor;

1.2 Design steps

- **Digital design:** the entire circuit is described in VHDL language;
- **Simulation :** the entire design is simulated using Modelsim tool, in order to verify the correct functionality;
- **Logical Synthesis :** starting from an the RTL VHDL description, the tool Synopsys synthesizes the circuit mapping it to a specific library of cells, so giving as output the description of the circuit at the logic level in verilog or VHDL language;
- **Physical Synthesis :** Finally, the verilog description is used by the tool Innovus in order to generate a physical implementation of the microprocessor.

1.3 RTL top view

The main purpose of a microprocessor is the execution of a program (a particular set of instructions). The pipelined processor requires 5 clock cycles in order to complete each kind of instruction. A top view of the DLX at the architectural level is reported below, it shows the interconnection between the main blocks: the *Program Counter* register (*PC*) shows the address of the *Instruction Memory* (*IRAM*) to which it points, and the IRAM answers giving the instruction to be processed. Then the *Instruction Register* (*IR*) takes the instruction and allows to see it to the Control Unit (*CU*) and the *Datapath*. The CU analyzes the instruction and gives the right signals to the Datapath that is able to execute correctly that instruction.

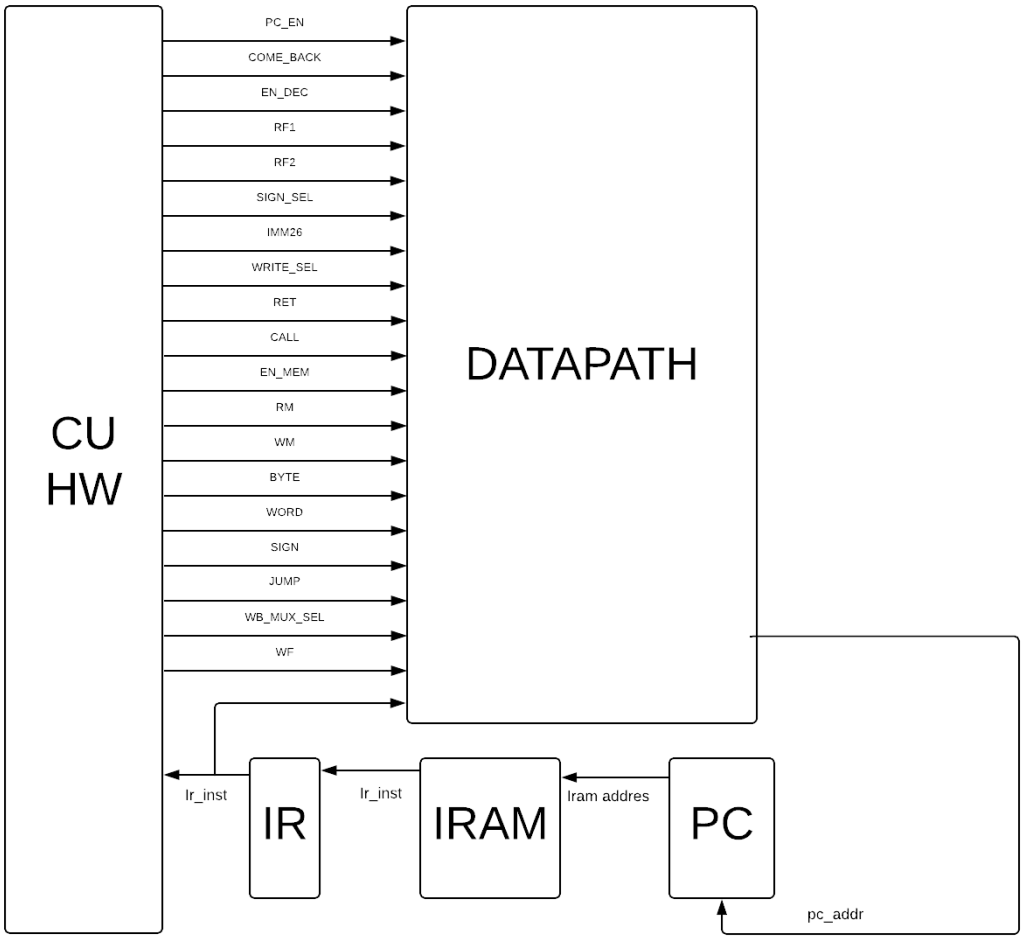

Figure 1: DLX top view

CHAPTER 2

Instruction set

The starting point of the design was the selection of the instructions that the processor should be able to perform. As said before this is a RISC architecture, so all the instructions have the same length (in this case equal to 32 bits). Since, each instruction requires a different type of action of the processor, it is coded with 6 bits (*OPCODE-field*) that are the most significant ones. Anyway, the DLX instructions can be grouped into 3 main types:

- **J-type:** Jump instructions, that can contain a 26-bit address or not;
- **I-type:** Loads/Stores and conditional branch instructions, these ones specify two registers one for reading and perform an operation with an immediate value of 16 bits, and one for writing the result of the operation.
- **R-type:** Register-register ALU operations. ALU operation is defined in the extra 11-bit field func,the least significant ones. Three registers references are involved, two for reading the values to use in the operation and one for writing the result. These instructions have the same opcode, so they differ according the *FUNC field*.

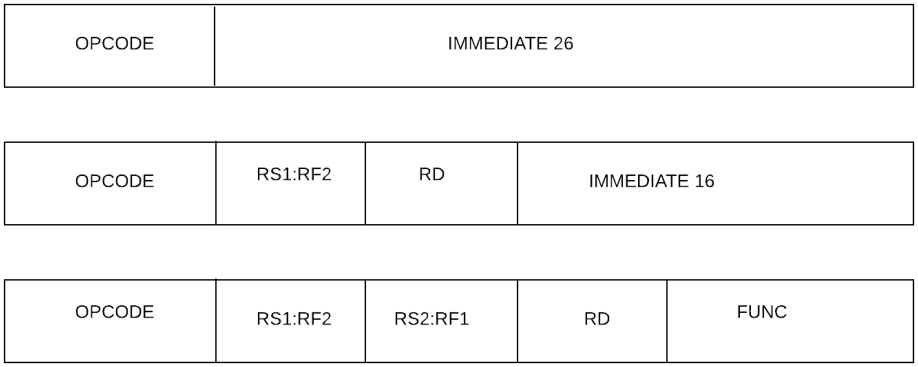

Figure 3: Types of instruction

The entire fulfilled *Instruction Set* is listed in the following table:

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

Figure 2: Table of implemented Instruction Set

CHAPTER 3

Program Counter, Instruction Register and Datapath

The PC, IR and the Datapath are the heart of the execution of an instruction. While the PC and IR are fixed blocks in a microprocessor, that allow its cyclic behavior, the Datapath is a variable component because the quantity and type of the blocks in this unit depend on the chosen *Instruction set* that the machine can execute.

3.1 Pipeline stages

The mentioned units are able to start and complete an instruction in 5 clock cycles but since the DLX is a pipelined structure the throughput is of 1 instruction per cycle. It means that the entire structure is split into 5 stages that work concurrently on 5 different instructions, taking 1 single clock tick, in other words each stage is scanned from one or more parallel registers. Anyway, since each stage can perform several actions (the Datapath expecially), the type of action is decided by the Control Unit that sends the needed control signals.

3.1.1 Fetch stage

In this phase literally the instruction is fetched toward the IR. In particular the *register PC* is pointing in the Instruction Memory that gives as output the new instruction to be execute. Normally the PC is enabled by the Control Unit by the signals *PC_EN* but it is disabled only during particular cases (FILL and SPILL operations) because of needed of stall of the entire pipeline in order to perform other operations (described later).

Control Signals

- *BRANCH_OUT*: it selects the input of the IRAM between the *PC register* and the Branch address;

Components

- *RCA1:* it uses the value of the current PC and computes the next PC value adding the fixed number 4, in order to point to the next location of the IRAM at the next clock cycle;
- *MUX-BRANCH* : This multiplexer is used in order to jump or not in case of Branch Instruction, so the Control Unit uses the *BRANCH_OUT* signal to select the address of the IRAM, that can be the one pointed by the PC or the one of the Branch Instruction, during its decode stage;

3.1.2 Decode stage

Control Signals

- *EN_DEC*: it activates the register of the next stage;
- *RF1*: it activates the first output of RF;
- *RF2*: it activates the second output of RF;
- *CALL*: in case of CALL operation the signal is assert;
- *RET*: in case of RET operation the signal is assert;
- *SIGN_SEL*: the signal is assert if the instruction expected a signed immediate;
- *IMM26*: the signal is assert if the instruction expected an immediate of 26 bit;
- *WRITE_SEL*: it is assert if there is an I-Type instruction arrives,to select the correct portion of IR signal that selects the writing address of the RF;
- *ZERO_SEL_OUT*: in case of comparison instruction the signal is assert, to send in input of the ALU a signal of 32 bits of zeros;

Components

- *NPC REGISTER* : This register always contain the same value that contains the PC, anyway it shows its output directly to the Decode stage, so while an instruction is in its Decode stage it is possible to see the value of the PC of the next instruction, that is in its Fetch stage;
- *INSTRUCTION REGISTER - IR* : The IR now is containing the instruction, that is sent to the Control Unit that immediately replies with the right control signals for this stage. Directly from the IR all the required signals are extracted and sent to the different components;
- *RF_WINDOWING + DELAY-REGISTERS + MUX* : It needs directly from the IR the addresses for its three ports. To be more specific the two reading address always correspond to the same bits of the IR, while the reading address changes according to the instruction type and so for this reason is used a multiplexer driven by the control signal *WRITE_SEL*. Although the write address is really used at this stage by the IR, the writing operation should be performed later, for this reason a sequence of registers is used in order to delay the address information without losing it.
- *SIGN-EXTENSION + MUX3*: In this stage for the I-type and J-type instructions the IR contains the information of the 'immediate' value, that can be of 16 or 26 bits, that can be interpreted as signed or unsigned value. This information will be stored into a 32-bit register in order to pass to the next stage, so an extension (signed or unsigned) is performed by a specific block, giving in output the four possible combinations. Since just one of this resulting 32-bit signals is needed, a multiplexer is required with two control signals coming from the CU, *IMM26* and *SIGN_SEL*.
- *MUX_ZERO_OUT*: This multiplexer is used in case of an instruction that in its execution stage need of a '0' as input of the ALU.
- *BRANCH_RCA*: This ADDER is used to compute the address of the next instruction just in case of Branch, that can be used or not to jump;

3.1.3 Execute stage

Control Signals

- *EN_EX*: it activates the register of the next stage;
- *MUXA_SEL*: it selects the output from *A* or *IN1* registers;
- *MUXB_SEL*:it selects the output from *B* or *IN2* registers;
- *COME_BACK*: it is the result of the evaluation of the prediction made by the Control Unit, in case of wrong prediction this signal is assert, in order to bring the PC to point at the Branch Instruction again, this signal should act at the end of the 5 cycles of the Branch and for this reason it is pipelined;
- *ALU_OPCODE*: it selects the type of operation that the Arithmetic-Logic Unit (ALU) should perform;
- *ALUsel*: it selects the correct output of the multiplexer that is included in the ALU, in case of comparison instructions;

Components

- *PC_2 REGISTER* : This register transports along the pipeline the value of the PC of the next instruction;
- *IN1 REGISTER* : This register contains the immediate value in case of I-type instructions;
- *A REGISTER* : This register contains the outputs of the port 1 of the Register File;
- *B REGISTER* : This register contains the outputs of the port 3 of the Register File;
- *IN2 REGISTER* : This register is connected to the *NPC register* of the previous stage, just to deliver to the ALU the value 'PC+4' used in many types of instructions;
- *MUX4* : According to the mentioned control signal *MUXA_SEL*, this multiplexer decide the *DATA2* entering to the MUX between the *IN1 register* and *A register*;
- *MUX5* : According to the mentioned control signal *MUXB_SEL*, this multiplexer decide the *DATA1* entering to the MUX between the *IN2 register* and *B register*;
- *ALU* : It is the calculator of the microprocessor, it can perform different type operations (addition, subtraction, bitwise operation, comparison) on two 32-bit inputs generally and only 16-bit inputs in case of multiplication. The result is always a 32-bit data;

3.1.4 Memory stage

Control Signals

- *EN_MEM*: it activates the register of the next stage;
- *RM*: it activates the reading operation of the DRAM memory;
- *WM*: it activates the writing operation of the DRAM memory;
- *BYTE*: it is needed to choose a memory operation using 1 byte of data;
- *WORD*: it is needed to choose a memory operation using 1 word of data;
- *SIGN*: it is needed to extend the data read from the DRAM, according to the required sign;

Components

- *PC_3 REGISTER* : This register transports along the pipeline the value of the PC of the next instruction;
- *OP_OUT REGISTER* : it contains the result of the ALU operation;
- *ME REGISTER* : it contains the input data for the memory that comes from directly from the register B;
- *DATA RAM - DRAM* : this is a RAM memory organized in 8-bit locations. It has one 32-bit input port and three different output port: 8-bit port, 16-bit port and 32-bit output port, and according the selection signals it gives the result through port or it store in one, two or four location the input data;
- *MUX6 + SIGN-EXTENSION* : Since the parallelism of the architecture is 32-bit, the output data of the memory needs to be extended according to the required signal and the same control signals that go into the DRAM, select the output of the multiplexer;

3.1.5 Write-back stage

In this stage is prepared the next PC, so the input of the *PC register* and the writing operations in the RF are performed.

Control Signals

- *JUMP*: it is activated in this stage only in case of Jump instructions, in order to select the data coming from the register *PC_JUMP*;
- *WB_MUX_SEL0 1* : it selects the data that should be written in the Register File;
- *WM*: it activates the writing operation for the RF;

Components

- *PC_4 REGISTER* : This register transports along the pipeline the value of the PC of the next instruction;
- *PC_5 REGISTER* : This register transports along the pipeline the value of the PC of the current instruction, because it is directly connected to the *PC_4 register*. It is used for the Branch instructions, in case of wrong prediction made by the Control Unit, in order to come back and execute again that Branch instruction;
- *PC-JUMP REGISTER* : it contains the address of the next instruction to be pointed in the IRAM, this content is selected by the *JUMP* signal;
- *MEM-OUT REGISTER* : it contains the data read from the memory;
- *ADDR REGISTER* : it shows the output of the ALU, just delayed of two clock cycles;
- *MUX7* : it allows to select from the register *ADDR, MEM-OUT* and *PC_4*;
- *MUX1* : even if the the block is placed apparently in the first stage of the pipeline, it gives the input of the PC;

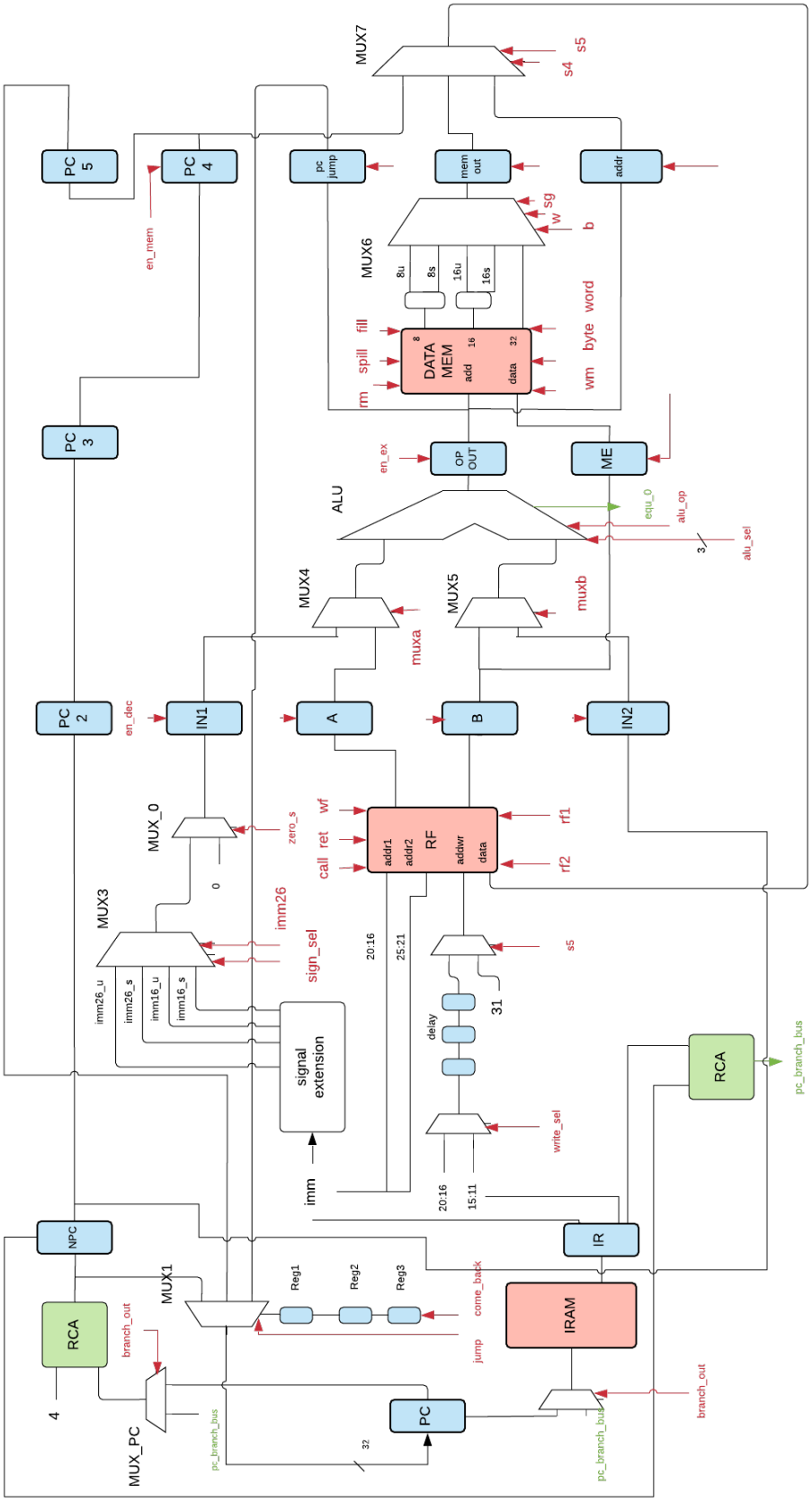

Figure 3: Datapath including IR and PC

3.2 Datapath

We described in this section the main components of the Datapath.

3.2.1 ALU

As is it possible to see in the figure below, the ALU is composed by several blocks:

- **Adder**
- **Boothmul**
- **Generic shifter**
- **Equal zero block**
- **Logic Elements**
- **Multiplexer**

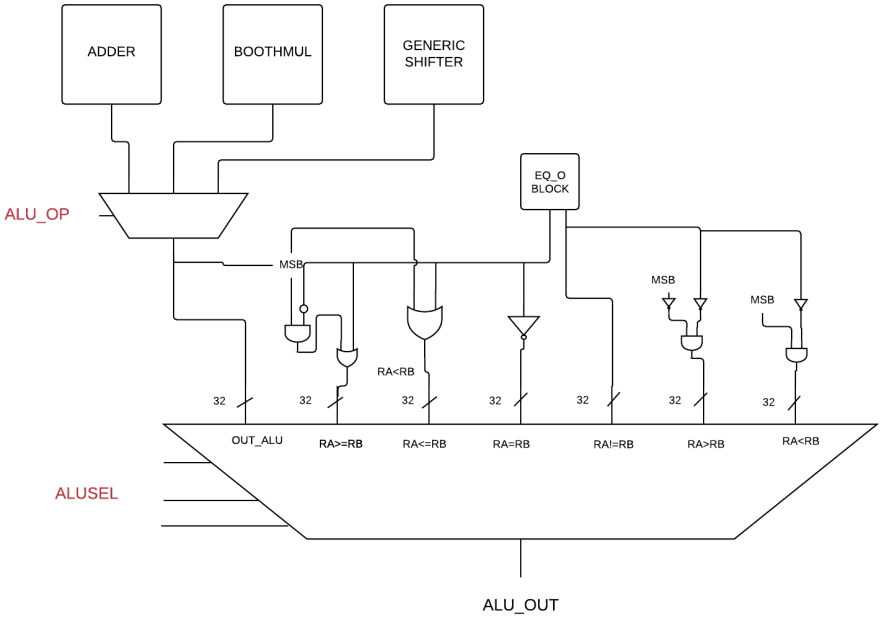

Figure 4: ALU

The entire components is used to execute every type of arithmetical operations,comparisons and logic operations. The signal *ALU_OP* is used to select the correct unit, then the other signal *ALU_SEL* is used to obtain the final result that is send to the *OP OUT* register.

Adder

The adder is a simple RCA that is used to execute the Sum operation, while Sub operation is executed using the rule of the complementation by 2, inserting the input carry.

Boothmul

An important improvement for multiplication consists in implementing the Booths Algorithm which is based on the recoding of multiplied numbers. The multiplier is based on Booths algorithm that is a multiplication algorithm that multiplies two signed binary numbers in 2s compliment notation. Booth

uses desk calculators that are faster at shifting than adding and creates the algorithm to increase their speed. In the figure below is reported a general stage of the multiplier, also in this case for the SUM operation is used an RCA.

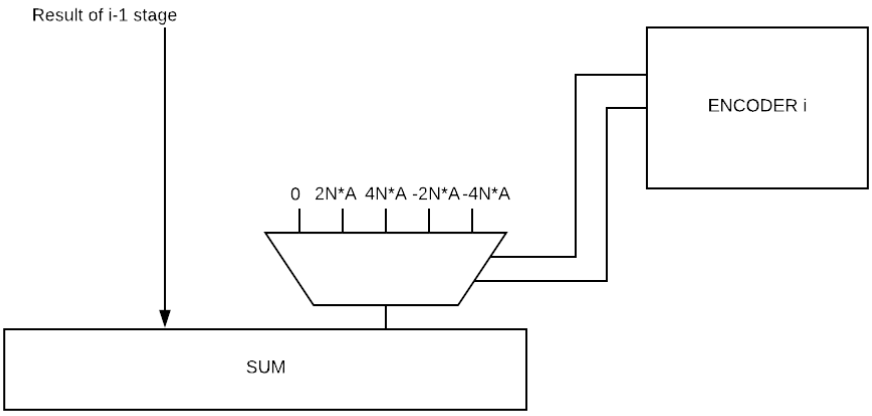

Figure 5: Boothmul generic stage

Generic shifter

The shifter that we used in the ALU is the same that is provide in the directory of the project. It can perform a shift operation up to 32 positions, both right and left, both logical and arithmetical.

Logic elements

The Logic Unit performs logic operations according to the input signal given by the Control Unit. It has an internal multiplexer that selects the output between the following bitwise logic operations between the two inputs: OR, AND, XOR. The default case for the output of multiplexer is the value of the rst input.

Equal zero block

The equal zero block is used to detect if the result of the SUB operation is equal to zero, this basic comparators composed by two generic nets:

- **xor net**
- **and net**

It possible to observe that this circuit gives the output equal 1 only if the result is equal to 0.

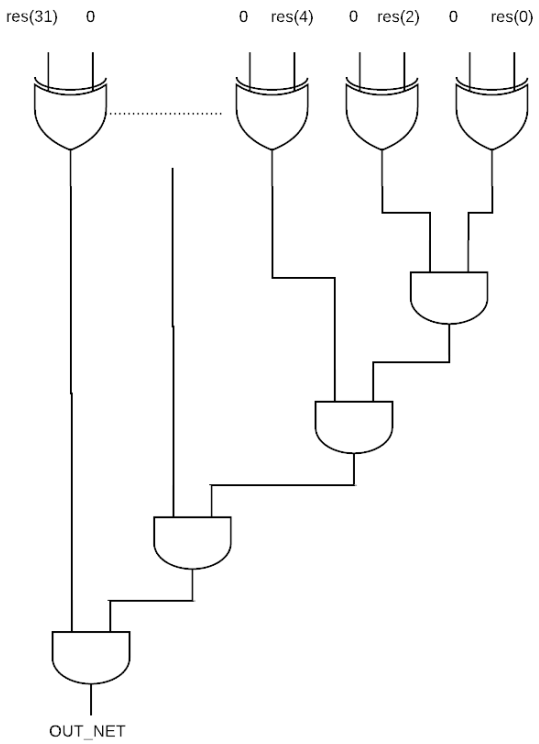

Figure 6: Equal zero block

3.2.2 Register File Windowing

A technique used to optimize overhead due to subroutine context switching is the windowing one. The basic idea is to allocate dierent parts of the RF (windows) to dierent routines of the program to run.

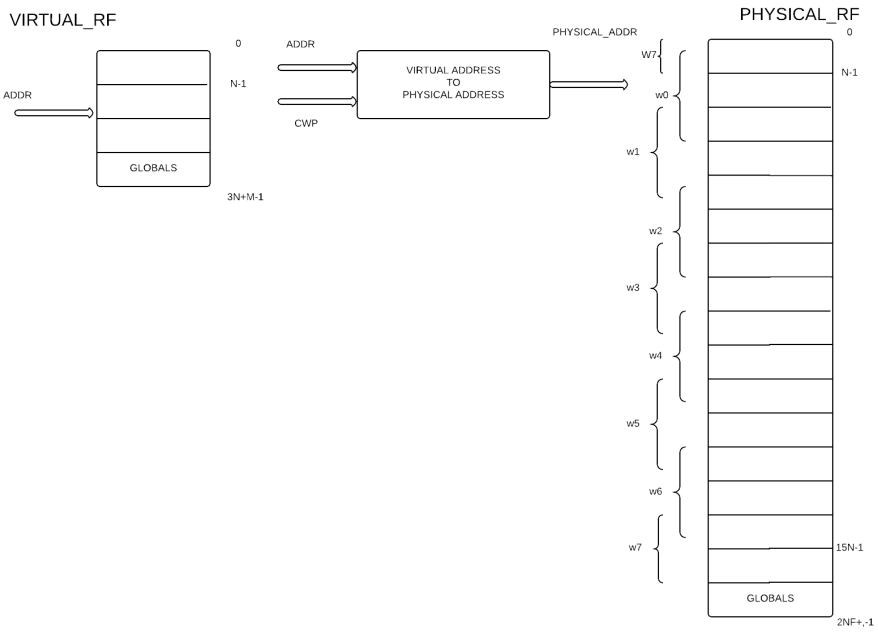

Figure 7 : Register File Windowing

The DLX includes a Register File Windowing with the following characteristics:

- 128 total registers for Integers;
- 8 register for Global section;
- The RF is divided in 8 windows, each window has 3 block (each one of 8 bits). The 3 blocks are : IN block, LOCAL block and OUT block, the last one is shared with the next window in a cyclic way;

Since there is not a limit to the number of subroutine that it is possible to use in a program, once the RF almost full it is necessary to move the data saved in the first window to the memory in order to make usable that window for another subroutine, it is a SPILL operation. The opposite operation is done when the data of that subroutine are needed so it is necessary to fill the RF window again with the old data, it is the FILL operation. To manage in correct way fill and spill operation of the register file, two pointer are used:

- **SWP:** Saved Window Pointer
- **CWP:** Current Window Pointer

CWP is incremented on call and decremented on ret. When the value of CWP equals the value of SWP, a spill or a ll is needed, depending on which instruction caused that condition.
Spill and Fill processes introduce a considerable delay in the program execution, because the pipeline is stopped.

CHAPTER 4

Memory

4.0.1 IRAM

The IRAM contains the instructions to be executed by the processor. Width and size of the IRAM are dened parametrically as generic constants. In our case each location contains 1 Byte. Basically the IRAM is represented by a bank of integer registers that is lled through the process FILL MEM P that reads an input les containing the instruction set. The address of the IRAM is represented by the output of the PC register.

4.0.2 Dram

The DRAM, like the IRAM, is represented by a bank of registers with width and size dened by parameters. As said before the DRAM is made of registers of 1 Byte. It is characterized by :

- 1 input port of 32 bits, activated from the signal WM, but it fills one, two or four memory locations according to the signals Byte and Word;
- 1 output port of 8 bits, activated by the signals RM and Byte ;
- 1 output port of 16 bits, activated by the signals RM and Word;
- 1 output port of 32 bits, activated by the signals WM;
- 1 input port of 32 bits for SPILL operation with the SPILL signal;
- 1 output port of 32 bits for FILL operation with the FILL signal;

Moreover we decided to set aside the last part of the memory for the SPILL and FILL operation, so just for the communication with the Register File, that involves specific bus.

CHAPTER 5

Control Unit

The Control Unit implementation follows the Hardwired style, it represent the most complex part of the entire processor. In this case the CU performs several task:

- **Instruction Coding**
- **Branch prediction control**
- **Stall management**
- **RF control**

Control signals associated to each instruction (Control Word) are stored into the Control Word memory. The Control Word of each instruction is set in Decode stage and it is propagated through the pipeline, according to the Instruction Registers pipeline. For each stage the correct signals are activated in order to execute the operation that is send by the IR. It is important to say that in this DLX there is not an Hazard detection, so for data dependency we must insert NOP operation in order to avoid this types of error.

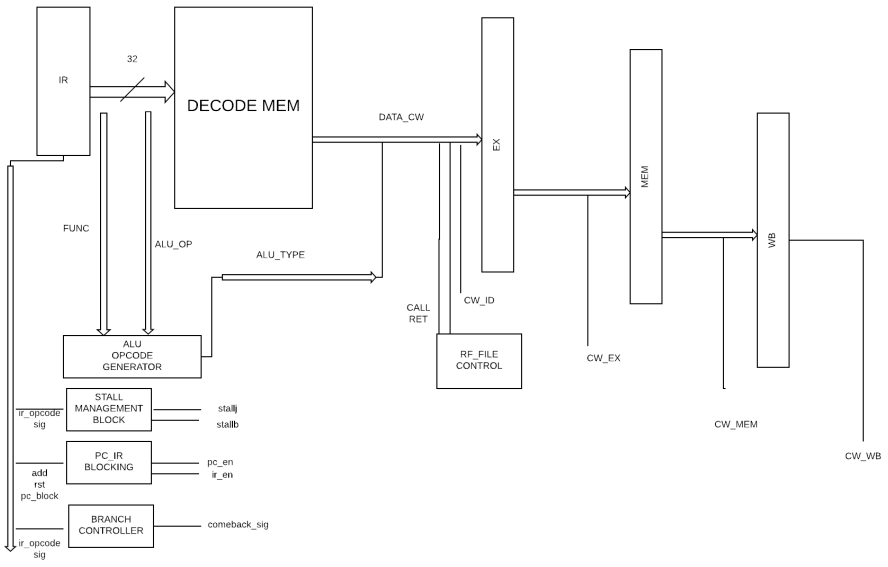

Figure 8: Control Unit Scheme

5.1 Instruction Coding

The main characteristics of the CU are the summarized below:

- The CU receives the instructions directly from the Instruction Register, so during the Decode stage. In this way the values of OPCODE and FUNC are obtained;
- The control word is obtained in this way: the IR instruction points to a specific cell of a memory that contains all the control words. In particular, the pointing address is the Opcode. If the address points to a not implemented instruction or to an empty location the result is a NOP instruction;
- The *ALU_OPCODE* and the *ALUsel* signal are obtained by a process that analyze both the OPCODE and the FUNC fields.
- To implement the pipeline there are 3 stage of registers and the control word signals are passed from one stage to the following one at each clock cycle;

5.2 Branch Prediction Unit

Branches can potentially impact in a very serious way the pipeline performance. It is possible to reduce the performance losses by predicting how branches will behave. They aim at correctly forecasting branches, thus reducing the chance that control dependencies cause stalls.
In order to avoid delay that concerns the branch operation we decide to use different circuit that act during the Decode and Execute stages:

- **Decode Stage Block 1**: This block analyzes only in the dedode stage of a branch instruction, the current state of a FSM, giving by the *HISTORY* signal and it decides to jump unconditionally showing as output the already mentioned signal *BRANCH_OUT* that control the next instruction to point.
- The flag that represents the branch instruction is delayed together with a significant bit of the opcode that differs for the two type branch instructions.
- A small combination block, analyzes the state signal *Equal_zero* coming from the ALU and sets the signal *SALTO* if the result of that instruction should be an effective branch.
- Then a branch history table is obtained using a simple FSM that have two states:
  - **S00: branch not taken**
  - **S11: branch taken**

  It verifies if the prediction has been done correctly just looking at the current state and checking the *SALTO* signals. The change of the state occurs in the case of wrong prediction and the signal *COME_BACK* is sent in order to give to the PC the value for IRAM that contains the branch instruction again. So a wrong prediction does not change the latency of the Branch instruction.

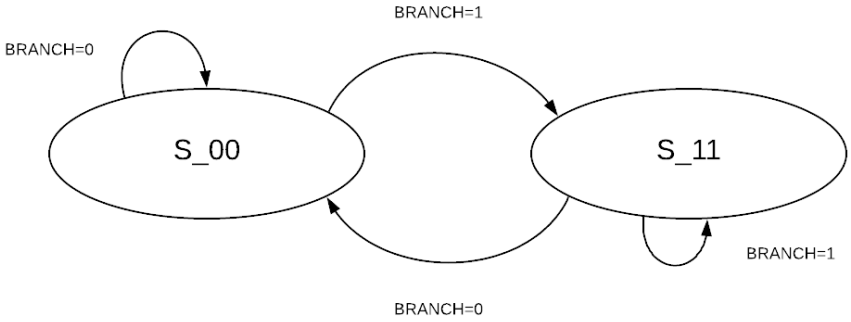

Figure 9: FSM used for branch prediction

5.3 Stall management block

Stall management block is used to avoid the execution of the next instructions in two cases:

- **Branch instruction in case of wrong prediction**
- **Jump instruction**

As a matter of fact, in case of an error of the BPU and in case of jump instruction the stall management block inserts several NOP operations to avoid errors (like permanent modification of data in the memory or RF). A net composed by several flip flops is used to count the number of stall periods.
The circuit for the branch stall receives in input the *COME_BACK* in case of branch and the one for jump receives a signal that indicates the jump instruction.

5.4 PC-IR Blocking

This block gives as output the enabling signals *PC_EN* and *IR_EN*, that are used to block or to activate the Program Counter or the Instruction register. In particular the Program Counter and the Instruction Register are disabled in three cases:

- **Rst = '0'**: if the Reset is active, the PC shows a zero at the output and it is blocked;
- **Address IRAM equal to the last address**: it means that the program has been executed entirely so the machine can be turned off;
- **PC block = '0'** this signal comes from the RF Control block of the Control Unit that is explained consequently.

5.5 Rf file controller

Register File Controller is a block that is used to manage the Register File windowing described previously. It receives in input: *call, ret, swp, cwp, fill, spill* and and gives as output the *PC_block* and the *Memory_Address*, a portion of the entire Address Bus of the DRAM dedicated to FILL and SPILL operations). This unit is organized into four main blocks:

- **Cansave_CU block:** it is a combinational circuit that generates a flag in case of the next operation should be a SPILL operation;
- **Canrestore_CU block:** it is combinational circuit that generates a flag in case of the next operation should be a FILL operation;

- **Counter SPILL/FILL**: it is a special counter implemented as a FSM, since different actions are required for FILL or SPILL operations. As the previous block set their signals, this special counter counts unconditionally maintaining the PC blocked for the numbers of clock cycle required for the entire RF-Memory operations;
- **Addressing Memory for FILL-SPILL :** it is a real up-down counter, that gives the address to the DRAM during these two particular operations;

CHAPTER 6

Logical synthesis and physical design

6.0.1 Logical synthesis

The final DLX has been synthesized trying different options and constraints:

- **Power**: The final dynamic power consumption is:
  - **without constraint:** 13.6762 mW
  - **with timing constraint:** 6.7404 mW
- **Area**: the total cell area obtained is
  - **without timing constraint:** 30605.428164
  - **with timing constraint:** 33121.788122
- **Timing**: The result obtained are the following:
  - **without constraint:**
    - **data arrival time**:6.18
  - **with timing constraint:**
    - **data required time**:5.97
    - **data arrival time**:-6.95
    - **slack(VIOLATED)**:-0.98

  As it possible to see, despite the constraint on the clock, the results for timing synthesis are worse with respect to the synthesis without constraints. We tried different timing constraints but all of them gave similar results. It seems that the tool cannot optimize the given circuit, as a matter of fact the result is violation of the slack.

6.0.2 Physical synthesis

- **Area**
  - **Area**:the total number of cells is 21213
  - **Gates**:the total number of gates obtained is 57544
  - **Area**:the total cell area obtained is 45920.4 um2

6.0.3 Memory synthesis

Both the memory are not synthesizable, for this reason they are not included in the datapath. The files has been modified in order to perform the synthesis steps, so the interface of the DLX has been changed.