Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: Group 11

Fausto Chiatante, Alessandro Tempia Calvino

October 26, 2018

Contents

CHAPTER 1

Introduction

1.1 Architecture

The DLX (DELUXE) is a fully pipelined processor with a RISC architecture derived from notorious predecessors (AMD 29K, DECstation 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARCstation-1, Sun-4/110,Sun-4/260) / 13 = 560 = DLX (in roman number).
It is based on the Harvard Architecture which relies on two different memories for instructions and data, allowing simultaneous instruction-fetching and data transactions.
It has a simple 32-bits Load/store architecture with 32 integer registers (R0-R31) and 32 floating-point registers (F0-F31). It uses two main addressing modes:

- Immediate: OP RD, RS1, #56, in which one immediate operand is included in the operation
- Displacement: OP RD, 128(RS), in which the memory is addressed adding an immediate to the content of a register

It features a 5-stage datapath composed of the FETCH, DECODE, EXECUTE, MEMORY and WRITEBACK phases, inhereted from the MIPS architecture.
It executes three types of instructions:

- R-Type: pure registers instruction with two source registers and one destination register specified (ALU register operations); table 1.1
- I-Type: one immediate source, optional source register and destination register (Immediate ALU operations and Branches); table 1.2
- J-Type: Jump and Jump&Link instructions; table 1.3

OpCode (6)	RS1 (5)	RS2 (5)	RD (5)	FUNC (11)

Table 1.1: R-Type instruction.

OpCode (6)	RS1 (5)	RS2 (5)	IMMEDIATE (16)

Table 1.2: I-Type instruction.

OpCode (6)	OFFSET (26)

Table 1.3: J-Type instruction.

1.2 Implementation

R-type		**General Instructions**	
Operation	Code	Operation	Code
SLL	0x04	J	j,0x02
SRL	0x06	JAL	j,0x03
SRA	0x07	BEQZ	i,0x04
ADD	0x20	BNEZ	i,0x05
ADDU	0x21	ADDI	i,0x08
SUB	0x22	ADDUI	i,0x09
SUBU	0x23	SUBI	i,0x0A
AND	0x24	SUBUI	i,0x0B
OR	0x25	ANDI	i,0x0C
XOR	0x26	ORI	i,0x0D
SEQ	0x28	XORI	i,0x0E
SNE	0x29	JR	j,0x12
SLT	0x2A	JALR	j,0x13
SGT	0x2B	SLLI	i,0x14
SLE	0x2C	NOP	n,0x15
SGE	0x2D	SRLI	i,0x16
SLTU	0x3A	SRAI	i,0x17
SGTU	0x3B	SEQI	i,0x18
SLEU	0x3C	SNEI	i,0x19
SGEU	0x3D	SLTI	i,0x1A
		SGTI	i,0x1B
		SLEI	i,0x1C
		SGEI	i,0x1D
		LW	i,0x23
		SW	i,0x2B
		SLTUI	i,0x3A
		SGTUI	i,0x3B
		SLEUI	i,0x3C
		SGEUI	i,0x3D

Table 1.4: Supported instruction set

The processor developed in this project keeps unvaried the base architecture of DLX adding optional features such has:

- Hardwired Control Unit
- Expanded instruction set architecture (Jump&Link, Unsigned operations, shift operations, comparison operations)
- Advanced Datapath with reduced branch delay

- Efficient ALU logic (Pentium4 adder, Logic Unit and Shifter derived from SUN Microsystem SPARC T2)
- Forwarding
- Hazard Detection and stalls
- Branch prediction

The ISA supported by the final implementation is contained in table 1.4.

CHAPTER 2

Design

2.1 Top Level

The top-level during the design phase includes all the major components of the projects (figure 2.1), including the two memories, and it only needs to be inserted into a testbench that provides Clk and Rst signals.

```
entity DLX is
  generic (
    IR_SIZE          : integer := 32;        -- Instruction Register Size
    PC_SIZE          : integer := 32;        -- Program Counter Size
    ADDRESS_WIDTH_DM : integer := 5          -- Address Width DRAM
    );
  port (
    Clk : in std_logic;
    Rst : in std_logic;                      -- Active Low
    DataOut : out std_logic_vector(31 downto 0));
end DLX;
```

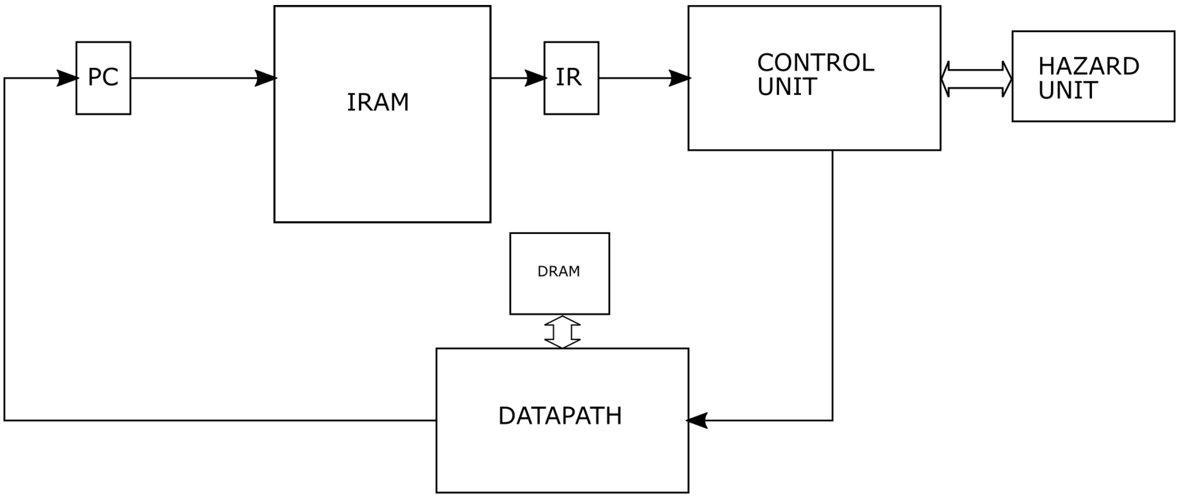

Figure 2.1: Schematic of the DLX top-level entity a-DLX.

At this level are present the connections between the upper-most components. Here also takes place the fetch stage of the pipeline. In this phase, at each clock cycle, the Program Counter (PC) is continuously

updated with the correct Next-PC, in order to fetch an instruction from Instruction RAM. The Instruction Register (IR) will contain the latest-fetched instruction.
The IR is parsed to deliver RS1, RS2, RD and the Immediate value to the datapath, according the type of instruction in execution.

2.2 Control Unit

The implemented Control Unit (CU) is of type hard-wired, which means that all the decision are the result of a combinational net. This decision has been taken to ensure the fastest and most easily readable realization.
The CU communicate with the datapath using a control word (CW) generated from the OpCode and eventual Func contained in the IR. This CW is composed of 29 single-bit signals, detailed in tables from 2.1 to 2.4, used to set the right configuration of the datapath. The CW is pipelined in four stages according to the architecture of the processor (the FETCH stage is already completed when an instruction enters the CU's domain). The CU also delivers a 5-bit Operation Code (ALU_OpCode) which determine the behavior of the ALU during the EXE stage.

1	2	3	4	5	6	7	8	9	10	11	12	13
I/R Type	RF Read1	RF Read2	RegA LEn	RegB LEn	RegImm LEn	RD1 LEn	Sign/ Unsigned	MuxImm Sel	Jump	Jump En	Eq Cond	MuxA Sel

Table 2.1: DECODE stage signals of CW

14	15	16	17
MuxB Sel	RegMe LEn	RALUout LEn	RD2 LEn

Table 2.2: EXECUTE stage signals of CW

18	19	20	21	22	23	24	25
PC+8 LEn	DRAM LEn	DRAM Read	DRAM Write	LMD LEn	RALUOut2 LEn	RD3 LEn	PC LEn

Table 2.3: MEMORY stage signals of CW

26	27	28	29
WBMux Sel	RF Write	ROut LEn	J&L Mux

Table 2.4: WRITEBACK stage signals of CW

There are also present the two signals STALL and Flush_BTB which will be described respectively in sections 2.6 and 2.7.

2.3 DataPath

The datapath contains the most important phases and critical components of a processor, detailed in the figure 2.2. Four of the five pipeline stages take place at this level.

The first one is the DECODE phase in which the instruction is interpreted by the CU. At this point the register file is accessed and the registers content read. The 16 or 26-bit immediate is extended as Signed/Unsigned and the correct one between the two is selected using a MUX (MuxImm).
Accessing the RF is done in parallel even before decoding the instruction thanks to the fixed location format of DLX instructions. This technique is known as fixed-field decoding.
In this stage are also included all the branching features, anticipated with respect to the standard DLX. This choice is intended to reduce the branch delay slot from three to one clock cycle, with respect to implementations which calculate branch targets through the EXE stage. Reducing the Branch Delay Slot comes with the trade-off of including an additional adder in the DEC stage, resulting in increased area.

In the EXECUTION stage the ALU elaborate the two operands coming from the previous cycle. It performs arithmetic and logical operations selected by the CU according to the current instruction. ALU is also used to calculate memory addresses to perform DRAM accesses.

In the MEMORY stage the data RAM is accessed either in Read or Write, if the instruction is a LOAD or a STORE. Otherwise, the stage is simply bypassed.

Finally, in the WRITE-BACK stage the result of the pipeline (either coming from ALU or DRAM) is saved to the Register File, if needed. At this stage also takes place the Jump&Link procedure which saves the original PC value in the register R31 for future use.

2.4 ALU

The ALU, in figure 2.3, contains 4 main components:

- A Pentium4 adder/subtractor which uses a sparse tree carry-generator and 4-bit carry-select blocks to improve the performance over traditional adders. This is the most critical component of the design as it has the biggest combinational delay in the system.
- A Logical Unit derived from the UltraSPARC T2 which features a two-level combinational logic, and is detailed in the schematic 2.4. The Logical Unit has been simplified with respect to the original implementation as the ISA of the DLX doesn't include NAND, NOR and XNOR. The two input lines A and B are composed of 32 bits, therefore is important to note that all the NAND gates are implemented internally as 32 3-input-NAND gates.
- A 3-stages Shifter derived from the UltraSPARC T2 as in figure 2.5. The first stage generates four masks on 40 bits, shifting at multiples of 8-bits depending on the type of shifting (Left, Right and Arithmetic Right). The second stage act a coarse grain shifting choosing the right mask between the available ones. At the end, a fine grain shifting translates back the operand to 32 bits with the right shift applied.
- A comparator able to evaluate $<, >, =, \neq, \geq, \leq$ between two+ operands. To perform these operations a subtraction between the two operands is needed, and it is performed by the Pentium4 adder. The result of the subtraction and the eventual carry out is taken as an input and a 1 or 0 on 32 bits is produced, depending on the wanted comparison. The comparator is able to perform operations both in signed and unsigned mode. In signed comparisons, is important to understand if the sign of the operands is different. If this case is verified then is very easy to calculate the result thus it depends only by the *MSB* of one of the two numbers. If the signs are equals then the numbers are treated as unsigned using the design in figure 2.6.

Talk about multi-cycle

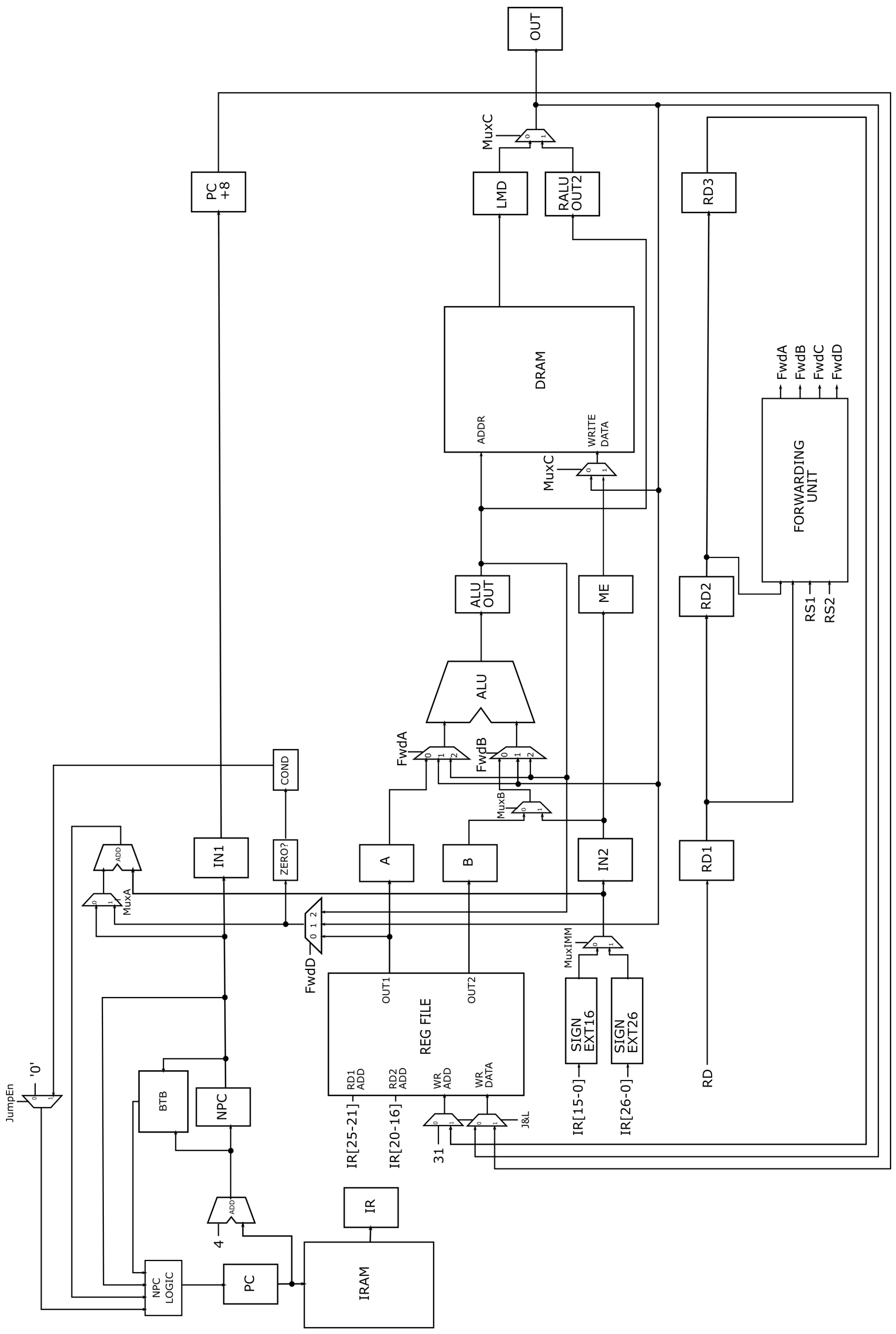

Figure 2.2: Schematic of the Datapath.

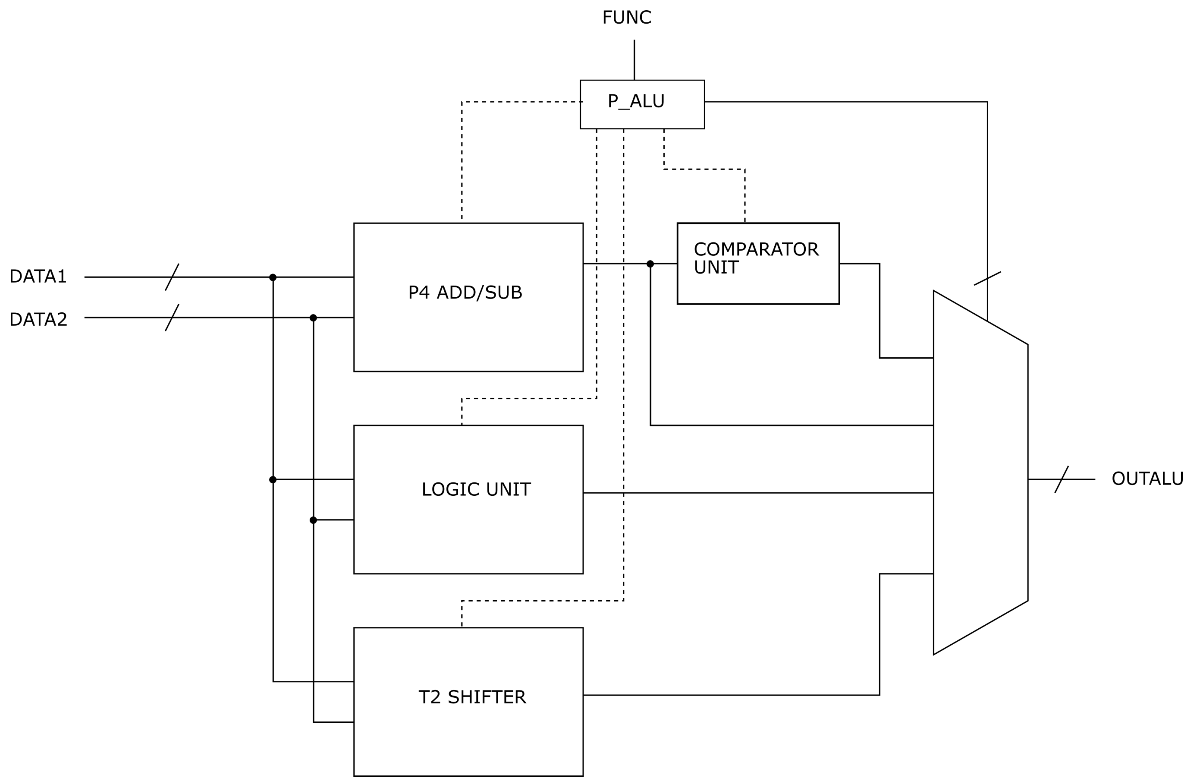

Figure 2.3: High-level schematic of the ALU's architecture.

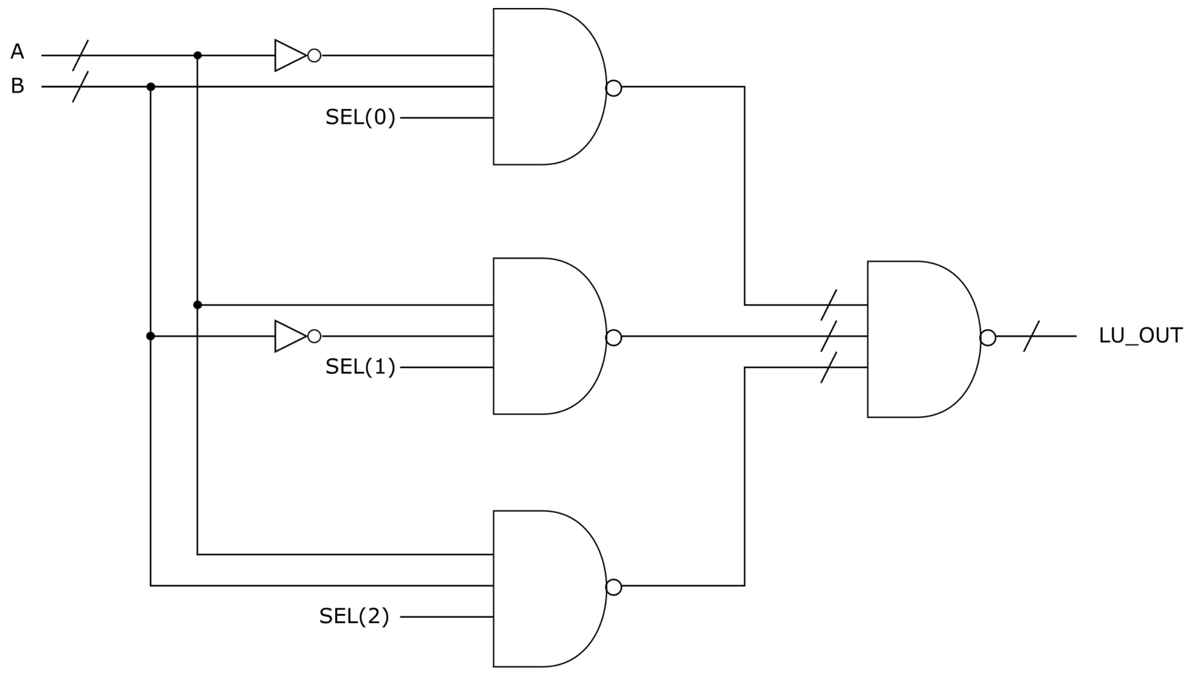

Figure 2.4: Schematic of the Logic Unit.

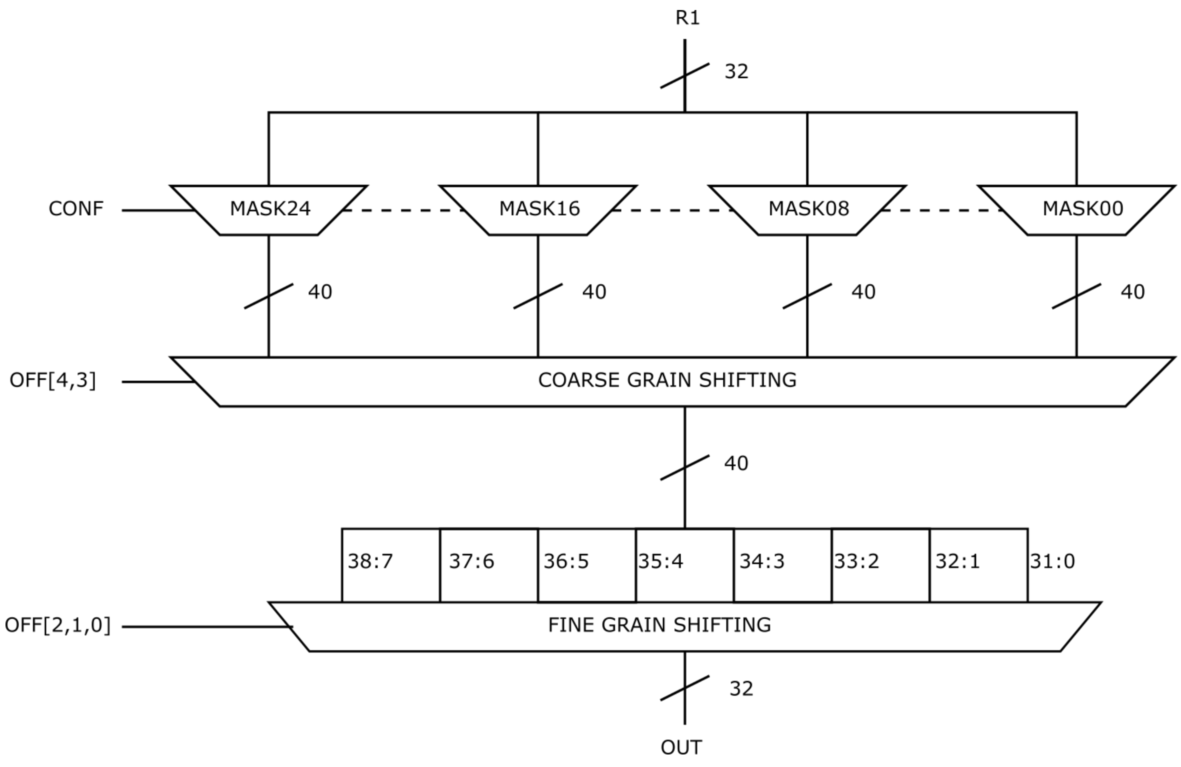

Figure 2.5: Schematic of the Shifter.

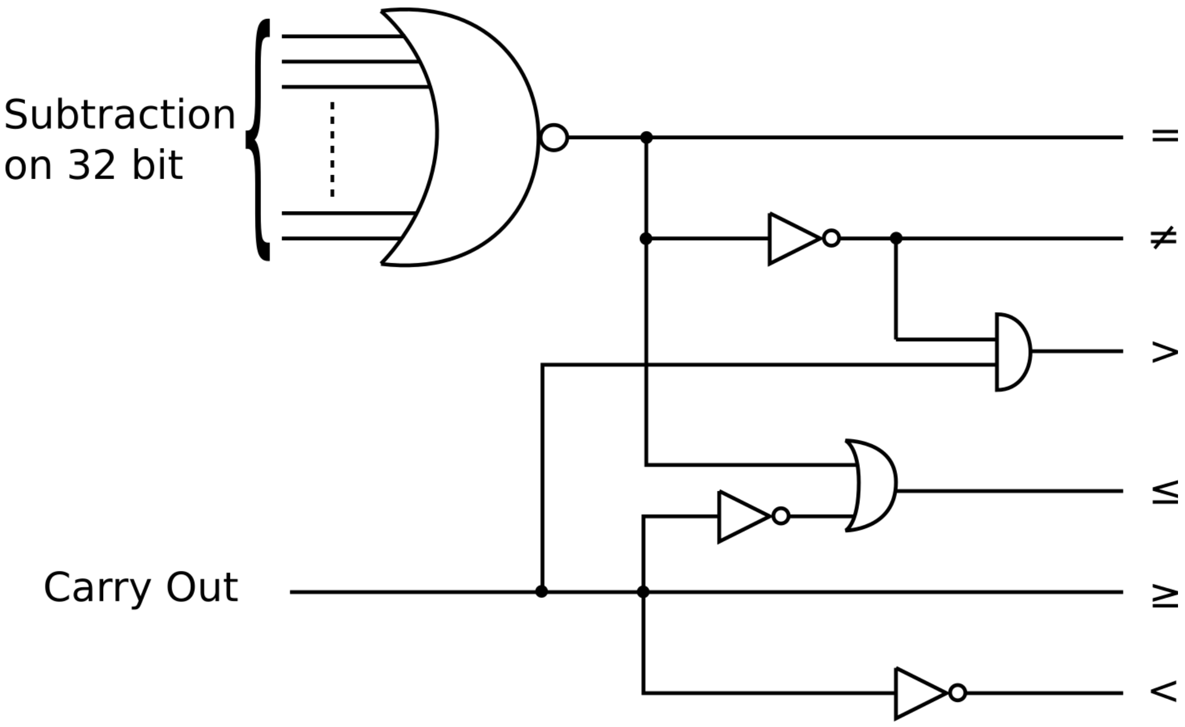

Figure 2.6: Schematic of the Unsigned Comparator.

2.5 Forwarding Unit

In a pipelined architecture, parallelization is exploited to achieve higher clock frequencies and performance. On the other side, though, having multiple operations in execution at the same time (five operations in this case) can lead to numerous data dependencies. Read-after-Write (RAW) dependencies can be solved effectively using the Forwarding technique. Forwarding allows data to skip the usual path ending with a WRITE-BACK in the Register File, and arrive directly where is needed. Data is produced mainly from:

- ALU: the output of RegALUout.
- DRAM: the output of MuxC, coming from RegLMD.

On the other hand, data is read from:

- ALU: Operand A and B taken as input from ALU.
- DRAM: the address targeted (doesn't cause problem because is directly connected to the ALU, which will be taken care of with forwarding) and the data to be written.
- Branching: the register values checked to either take a branch or not.

We need to select at each reading point which of the possible data in the pipeline are to be read, either coming form the previous stage, the ALU output or the DRAM output. The Forwarding Unit introduces four signals that control the MUXs inserted before each reading point. The most important information that the FU take as input, is the source and destination registers of each instruction in execution. The second needed information is wether the instruction in execution will write the result in the register file or not. This informations are pipelined (a similar pipeline to the datapath) inside the FU and compared at each stage to reveal possible data dependencies. Depending on the stage in which the dependency is discovered. data will be forwarded from ALU or from MEM stage. As an example, OperationX (just terminated the EXE stage) present a destination register which is equal to the source register A of OperationY (about to enter the EXE stage). The FU will recognize that operationX will write in the Register File, that the register is shared with OperationY and will react activating ForwardA on position 2 (refer to figure 2.2 for the MUX's positions).

2.6 Hazard Unit

Not all data dependencies can be solved using forwarding. Two particular situations are critical in the presented pipeline:

- A LOAD operation followed by an ALU operation which uses the data retrieved from memory: Data is retrieved in MEM stage and should be used in EXE stage. In two successive instructions this two stages are concurrent, making forwarding useless.
- A Branch checking a register that has been written in the previous instruction: the data to be written in the register is calculated in the EXE stage, while the (anticipated) branch is performed in the DEC stage, resulting in the same problem.

More in general, every time a data dependency involves two stage that would necessitate to "go back in time", forwarding cannot be used and other techniques are to be taken in consideration. Normal execution cannot be an option therefore a stall is used. Stalling means inserting a bubble in the normal program execution, in order to delay the problematic instruction just enough to resolve the data hazard. In both of the cases highlighted above, delaying the execution of 1 clock cycle is enough to resolve the problem.
A stall in this architecture is implemented transforming the faulty instruction, currently in the DEC stage, into a NOP which will have no effect on the system. At the same time, fetching has to be paused for 1 clock cycle, in order to be able to re-execute the same instruction in the following cycle. Fetch pausing is performed by disabling the LatchEn on the PC register.

2.7 Branch Target Buffer

Another idea in order to increase code execution performances is to insert a branch prediction unit in the processor.
Branches are a very critical part of the code. Take a wrong branch path means to undo the incorrect instructions executed, creating a bubble in the pipeline. That happens because the jump direction is unknown during the fetch as it is calculated in the decode stage.
The goal of this unit is to minimize the possibility of take wrong branches. The classical and easier branch decision is the static one. In this case branches are considered always taken or not taken but this is efficient in term of performance. A little improvement could be realized considering that usually forward branches are more often not taken and that backward branches as more often taken due to loops structures.

The BTB is a type of dynamic prediction. It takes decisions depending the previous history of the instructions. It is composed by a small direct-associative memory in which each entry contains the address of the considered branch and the target value to be loaded in the PC. The access to the memory is realized trough the lower bits of the instruction address depending the number of lines of it the cache. The PC is updated at the end of the fetch stage, before that the branch instruction is decoded, following the criteria indicated in the table 2.5.

Instruction in buffer	**Prediction**	**Actual branch**	**Penality c.c.**	**Action**
Yes	Taken	Taken	0	Nothing
Yes	Taken	Not taken	1	Remove from BTB
No	Not taken	Taken	1	Add in BTB
No	Not taken	Not taken	0	Nothing

Table 2.5: BTB penalty and action table

A basic architecture can be seen in figure 2.7.
In order to undo instructions, in the case of wrong prediction, a BTB logic raises a flush signal to the control unit in the decode stage. Then the CU substitute the wrong instruction with a NOP to guarantee a correct program flow. In particular, a flush is raised when:

- In the previous clock cycle there was not a flush, in order to not allow a comparison on a wrong instruction;
- The previous prediction of the branch was wrong i.e. it differs from the one calculated in the decode stage;

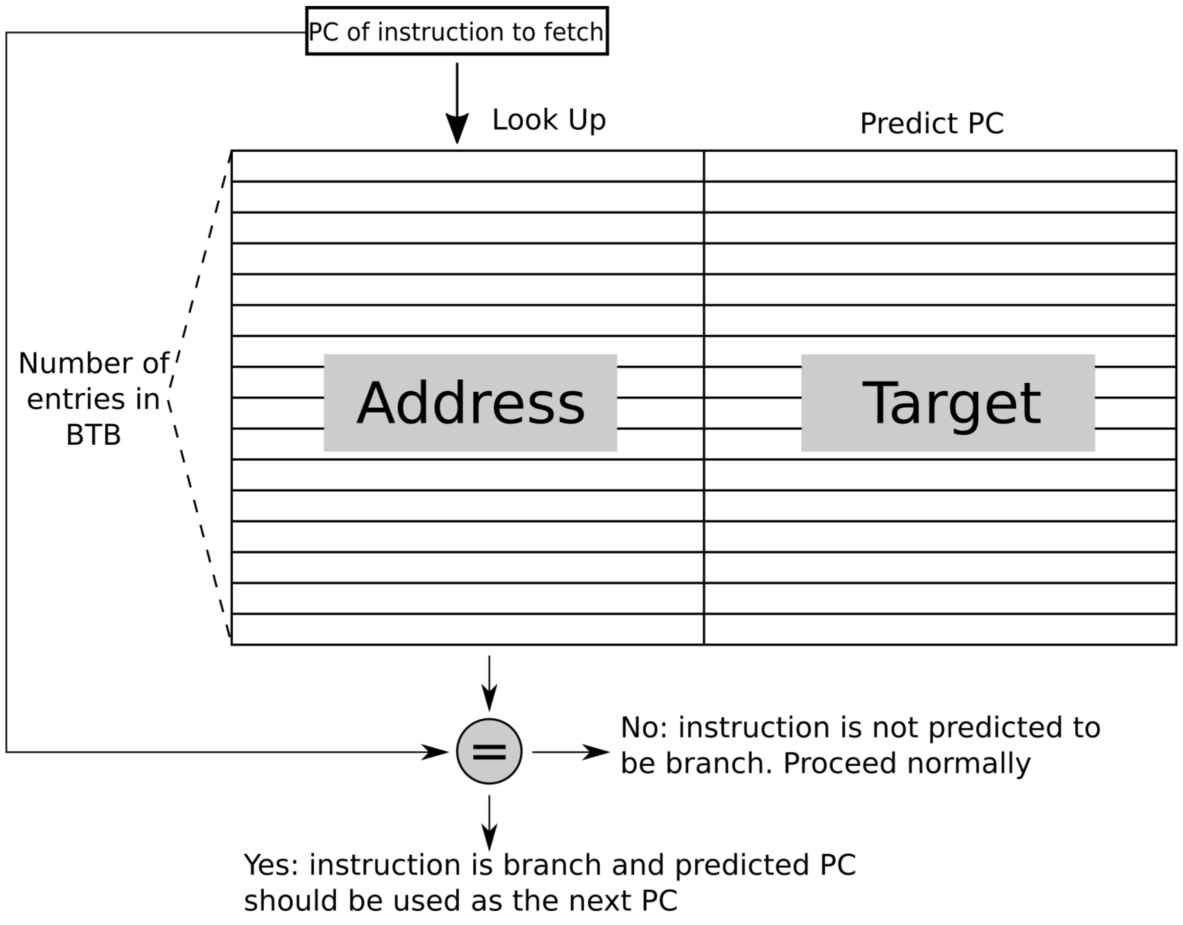
PC of instruction to fetch
Look Up
Predict PC
Number of entries in BTB
Address
Target
=
No: instruction is not predicted to be branch. Proceed normally
Yes: instruction is branch and predicted PC should be used as the next PC

Figure 2.7: Branch-target buffer architecture.

CHAPTER 3

Physical Synthesis

3.1 Synthesis

Once the modeling of the DLX microprocessor is concluded, we shift from simulation to physical synthesis, performed in **Design Vision**. We modified a predefined TCL script to speed up the synthesis process and to fit our needs. Also the synthesis is done optimizing separately the critical components and using them "as is" when building up the higher-level parts.
At the beginning of the script all the file are analyzed hierarchically to check for any coding error. To be able to go more in depth in each component, find the best performance and reuse the implementation in every instance of the component, the following Design Compiler command are used:

- **Characterize** and **current_design**: used to select the current module to optimize.
- **set_max_delay**: used to fix the constraint of the current module.
- **set_dont_touch**: used to save the module design and prevent any future modification of it during the next steps of the synthesis.

All the report file are saved in the report folder, while the synthesized netlists are stored in the arch folder. Finally, the timing and power reports are produced, obtaining the following values:

Cell internal Power	0
Net Switching Power	0
Total Dynamic Power	0
Cell Leakage Power	0

Table 3.1: Timing analysis results

Data Required Time	0
Data Arrival Time	0
Slack	0

Table 3.2: Power analysis results

3.2 Place and Route

Using the netlists obtained from the physical synthesis, we move to the final step of the project. The place and route is performed in **Cadence Encounter**. The steps followed are similar to the one given in the laboratories of the course Microelectronic Systems.

CHAPTER 4

Combinational logic insights

4.1 P4 adder

This is a particular type of adder, on 32 bits, based on a Sparse-Tree carry generator logic that generates the carry every four bits. The carries go into a carry select adder block that generates in parallel the sum for a carry in of zero and of one 4 bits using Ripple-Carry adders, see figure 4.1.

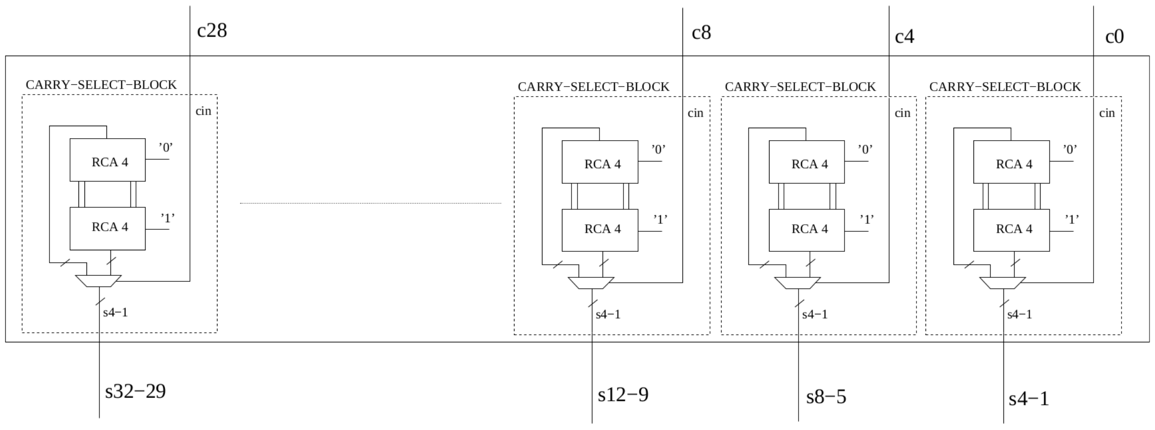

Figure 4.1: Carry Select Block.

This adder carry logic is based on the concept of generation and propagation.

- A generation happens when a carry bit is generated by the sum of two bits, so they are both one.

$$g_i = a_i \wedge b_i.$$

- A propagation happens when one of the two bits are 1 so that a carry can propagates through.

$$p_i = a_i \vee b_i.$$

Using generation and propagation the carry can be written in this way:

$$C_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i) = g_i \vee (p_i \wedge c_i)$$

If $g_i = 1$ then the carry is generated at step i, while if $p_i = 1$ then input carry is propagated. Iteratively:

$$C_{i+1} = g_i \vee (p_i \wedge (g_{i-1} \vee (p_{i-1} \wedge c_{i-2})))$$

These values are calculated in the first step of the Carry Sparse-Tree in the PG network.
Other two types of blocks should be defined in the carry net. To do that two more definitions are needed:

- The general generate of a stage i is equal to 1 if there is a generation of a carry in i or if there is a propagation in i and a generation in $i-1$:

$$G_i = G_i \vee (P_i \wedge G_{i\_1})$$

- The general propagate of a stage i is equal to 1 if there is a propagate both in stage i and $i-1$.

$$P_i = P_i \wedge P_{i\_1}$$

With these two definitions we can define two types of blocks: PG that implements both the general generation and the general propagation and G that implements only the general generation. They have a particular topology and disposition that can be seen in figure 4.2.

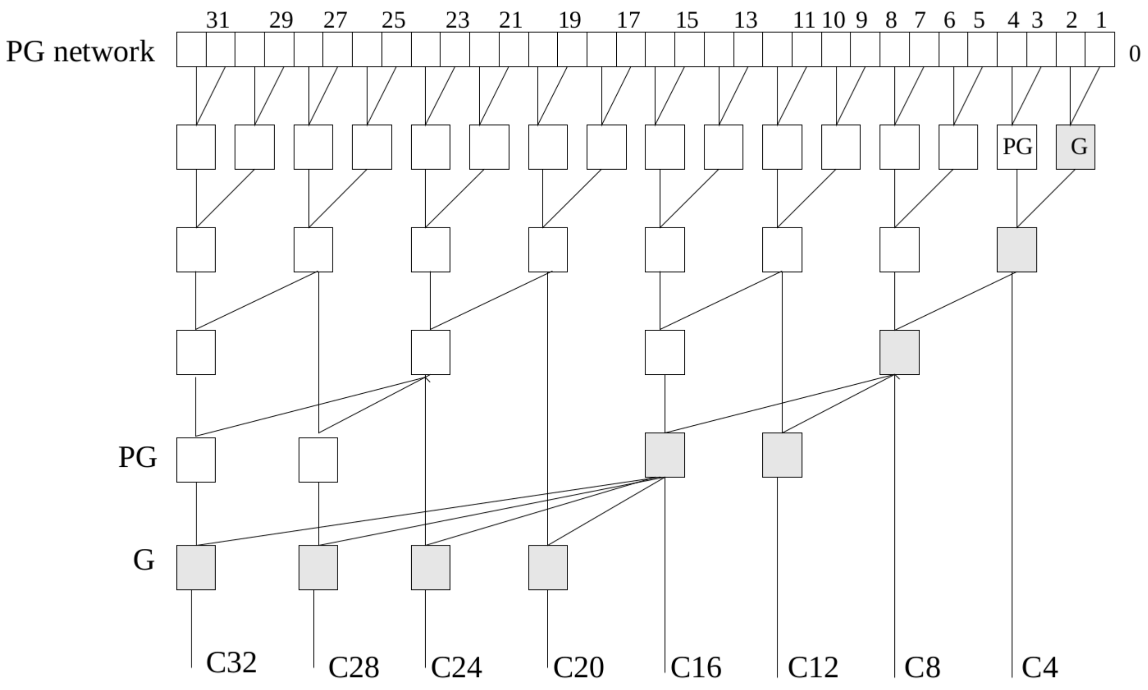

Figure 4.2: Carry Sparse-Tree: PG in white and G in grey.

Could be necessary adding a Carry in to the adder. To do that we need to modify the first block of the PG network adding the general generate and a Cin input. So the resulting generate of the block will be:

$$G = (a_0 \wedge b_0) \vee (C_{in} \wedge (a_0 \oplus b_0));$$

This can be very useful in order to allow the adder also to do the subtraction. To do that just put a xor port between b and the carry in for every b input. Cin now is the control SUB/ADD.

4.2 Floating Point Adder

The IEEE 754 is a technical standard for floating-point computation. The standard specifies:

- Formats for binary and decimal floating point data for computation and data interchange
- Different operations (such as addition, multiplication etc.)
- Conversion between integer and floating point formats
- Properties to be satisfied when rounding numbers
- Indications of exceptional conditions (such as division by zero, overflow, etc.)

Numbers are represented in the format $x = s \times b^e$ in which s is the significand, b is the base (2 for binary) and e is the exponent.
In single-precision arithmetic, numbers are stored in 32 bits: 1 for the sign, 8 for the exponent and 23 for the fraction, table 4.1. The exponent is a signed number represented using a bias of 127, so the general representation for normal numbers become $x = (-1)^{sign} \times 1.f \times 2^{e-127}$.

S(1)	EXPONENT(8)	FRACTION(23)

Table 4.1: Single-precision number

Different types of numbers are defined in the standard (table 4.2):

- **Normal numbers**: the significand is on 24 bit but only the fractional part is stored in 23 bits in the format $1.f$. In this way the total precision remains of 24 bits, equivalent to $\log_{10}(2^{24}) \approx 7.225$ decimal digits.
- **Denormal numbers**: also known as *subnormal numbers*, fill the underflow gap around zero in floating-point arithmetic. If E_{min} is the smallest exponent, a number less than $1.0 \times 2^{E_{min}}$ cannot be represented. The standard defines what is called *gradual underflow*, numbers less then the minimum normal number are represented using significand less than 1. Thus, as numbers decrease in magnitude, they gradually lose their significance and their precision and are only represented by 0 when all their significance has been shifted out. So when the exponent is equal to E_{min} the number is configured as $0.f \times 2^{E_{min}}$.
- **Zero**: when both exponent and fraction are equal to zero.
- **NaN**: standing for *Not a Number* it means that the number value is undefined or unrepresentable. For istance $0/0$ and the sum between $+\infty$ and $-\infty$ are NaN. These numbers are represented with exponent equal to E_{max} and the fraction different form zero.
- **Inf**: when the exponent is equal to E_{max} and the fraction is equal to zero. The sign is decided by the sign bit.

Type	Sign	Exponent	Fraction	Number
Normal	s	$E_{min} < e < E_{max}$	f	$(-1)^s \times 1.f \times 2^e$
Denormal	s	E_{min}	f	$(-1)^s \times 0.f \times 2^{Emin}$
Zero	s	E_{min}	0	0
NaN	s	E_{max}	$\neq 0$	*NaN*
+Inf	0	E_{max}	0	$+\infty$
-Inf	1	E_{max}	0	$-\infty$

Table 4.2: Numbers representation

In a floating point addition is important to identify these types of numbers at the first step because when a *NaN* or a $\pm\infty$ are present the addition is useless.

The adder realized is divided in 3 stages pipelined.
In the first stage:

1. It extracts signs, exponents and the fractions, looks for special cases (*NaN* and $\pm\infty$) and finds out what types of numbers are given (*Normal*, *Denormal*, *Mixed*). In case of special cases the result is immediately calculated.
2. If $Exp_1 < Exp_2$ then it swaps the operands and sets the result exponent to $Exp1$
3. It standardizes the denormal number in the case of mixed numbers in a way that can be treated as a normal case. To do that the subnormal number is shifted to the left by the number of zeros before the first 1 and the new exponent is upgraded with the correct value.
4. It shifts the lower exponent number fraction to the right by $Exp1 - Exp2$

In the second stage:

1. Depending on the numbers' sign and if the operation is an addition or a subtraction, it calculates the output sign and the operation to do
2. Addition or Subtraction of the numbers using a P4 adder
3. Replace the result with the two's complement if the operands were swapped, the operation was subtraction and if the result is negative
4. It groups the result into a single vector

In the third stage:

1. It shifts the result left until the MSB is different from zero and it sums the carry out to the exponent
2. It rounds the result
3. It selects the result between the adder and the special cases