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 05

Lezzoche Davide, Grottesi Lorenzo

September 17, 2020

TABLE OF CONTENTS

Introduction 5

Specification 5

Functionalities 6

Functional schema and blocks description 8

Control unit 9

Datapath 10

Fetch stage 10

Decode stage 11

Execute stage 12

Memory stage 14

Writeback stage 15

ALU 16

Integer ALU 18

Adder subtractor unit 19

Multiplier unit 19

Shifter 21

Logic unit 21

Comparator unit 22

Floating-point units 23

Floating point additions, subtractions and comparisons 24

Floating point multiplication 26

Floating-point division 27

Data forwarding 29

Implementation 33

Netlist simulation 33

Synthesis 34

Physical design 35

Discussion and conclusions 36

A. DLX VHDL 37
B. CONTROL UNIT VHDL 42
C. FETCH STAGE VHDL 50
D. DATAPATH VHDL 52
E. ALU VHDL 62
F. FORWARDING UNIT VHDL 65
G. JUMP LOGIC VHDL 71

Figures and tables

Figure 1 - I-TYPE (load, store, reg. immediate ALU op.) 5
Figure 2- R-TYPE 6
Figure 3- J-TYPE 6
Figure 4 - DLX block diagram 8
Figure 5 - Control Unit functional scheme 9
Figure 6 - Fetch stage block diagram 10
Figure 7 - Decode stage black diagram 11
Figure 8 - Execute stage 13
Figure 9 - Memory stage 14
Figure 10 - Writeback stage 15
Figure 11 - ALU general structure 17
Figure 12 - Integer ALU 18
Figure 13 - Pentium IV adder 19
Figure 14 - Multiplier 20
Figure 15 - Operand A rotate extension 21
Figure 16 - Logic unit 21
Figure 17 - Unsigned comparator 22
Figure 18 - Floating-point representations 23
Figure 19 - Float single unit 23
Figure 20 - Float double unit 24
Figure 21 - Floating point add/sub network 25
Figure 22 - Floating-point multiplication 26
Figure 23 - Floating-point division 27
Figure 24 - Non-restoring cellular array divider 28
Figure 25- Forwarding multiplexer 29
Figure 26 - Stall example 30
Figure 27 - Data forwarding 31
Figure 28 - Datapath 32
Figure 29 - ASM test 33

Figure 30 - Waveform 34
Figure 31 - Routed DLX 35
Table 1 - Instruction set 7

INTRODUCTION

Specification

The aim of the project is the design and synthesis of a *MIPS-64* like general purpose processor, also named *DLX*.

This is 5 stage pipelined processor and so architecture and control must be organized following the five stages, that are:

- *Fetch*, Instruction form IRAM is read and Instruction register (IR) is loaded.
- *Decode*, the IR is decoded and operands are prepared.
- *Execute*, ALU (Arithmetic logic unit) is feed with the operands and the output is stored.
- *Memory*, access the memory if necessary.
- *Writeback*, result of the computation is stored back to the register file.

To perform this architecture a dedicated *control unit (CU)* and *datapath* must interleave. The datapath performs all the functional blocks and computation, the CU feed the control word of the datapath according to the value stored in the IR, to use the correct functional block in order to perform the right operation.

The control unit sets the control word accord to the read instruction, in particular is possible to classify instructions into 3 main groups:

- **I-type,** operation performed between an operand and an immediate.
- **J-type,** jump operation.
- **R-type,** registers operation.

The DLX is a *RISC* processor so all instructions are mapped using the same bit length (32 bit). The encoding of all operations is specified by 6-bit *OPCODE* field, that classify the type of instruction.

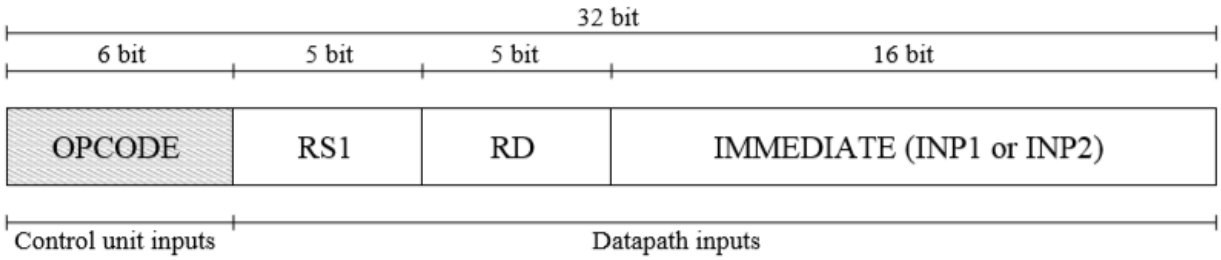


Figure 1 - I-TYPE (load, store, reg. immediate ALU op.)

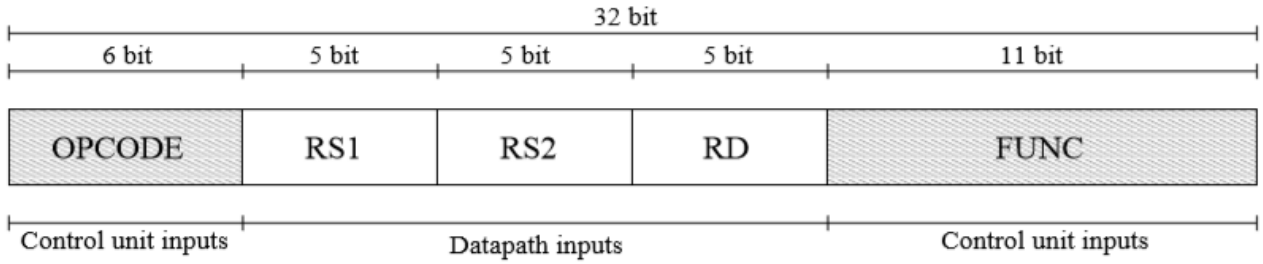


Figure 2- R-TYPE

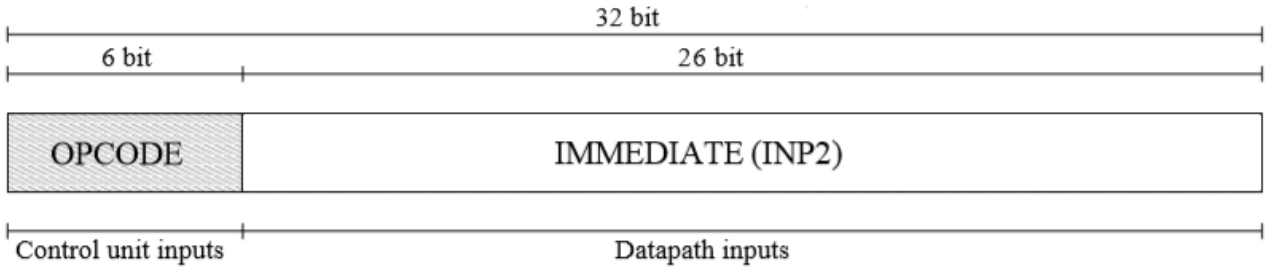


Figure 3- J-TYPE

Functionalities

The developed DLX presents an *ISA (Instruction Set Architecture)* that handles operations between integers and two kind of precision floating-points: single and double.

Furthermore, to optimize the program execution time a *forwarding unit* has been implemented to avoid read after write (RAW) hazards between operands.

Table 1. contains all the supported assembly instructions.

Table 1 - Instruction set

Operation	OPCODE	FUNC
J_TYPE		
J	0x02	X
JAL	0x03	X
I_TYPE		
BEQZ	0x04	X
BNEZ	0x05	X
BFPT	0x06	X
BFPF	0x07	X
ADDUI	0x09	X
SUBI	0x0A	X
SUBUI	0x0B	X
ANDI	0x0C	X
ORI	0x0D	X
XORI	0x0E	X
LHI	0x0F	X
JR	0x12	X
JALR	0x13	X
SLLI	0x14	X
NOP	0x15	X
SRLI	0x16	X
SRAI	0x17	X
SEQI	0x18	X
SNEI	0x19	X
SLTI	0x1A	X
SGTI	0x1B	X
SLEI	0x1C	X
SGEI	0x1D	X
LB	0x20	X
LH	0x21	X
LW	0x23	X
LBU	0x24	X
LHU	0x25	X
LF	0x26	X
LD	0x27	X
SB	0x28	X
SH	0x29	X
SW	0x2B	X
SF	0x2E	X
SD	0x2F	X
SLTUI	0x3A	X
SGTUI	0x3B	X
SLEUI	0x3C	X
SGEUI	0x3D	X
ADDED BY US		
RSLI	0x30	X
RSRI	0x31	X
NANDI	0x32	X
NORI	0x33	X
XNORI	0x34	X

Operation	OPCODE	FUNC
R-TYPE		
SLL	0x00	0x05
SRL	0x00	0x07
SRA	0x00	0x08
ADD	0x00	0x21
ADDU	0x00	0x22
SUB	0x00	0x23
SUBU	0x00	0x24
OR	0x00	0x26
XOR	0x00	0x27
SEQ	0x00	0x29
SNE	0x00	0x30
SLT	0x00	0x2a
SGT	0x00	0x2b
SLE	0x00	0x2c
SGE	0x00	0x2d
MOVF	0x00	0x32
MOVD	0x00	0x33
MOVFP2I	0x00	0x34
MOVI2FP	0x00	0x35
SLTU	0x00	0x3a
SGTU	0x00	0x3b
SLEU	0x00	0x3c
SGEU	0x00	0x3d
ADDED BY US		
RSL	0x00	0x40
RSR	0x00	0x41
NAND	0x00	0x42
NOR	0x00	0x43
XNOR	0x00	0x44

Operation	OPCODE	FUNC
R-TYPE(FP)		
ADDF	0x01	0x00
SUBF	0x01	0x01
MULTF	0x01	0x02
DIVF	0x01	0x03
ADDD	0x01	0x04
SUBD	0x01	0x05
MULTD	0x01	0x06
CVTF2D	0x01	0x08
CVTF2I	0x01	0x09
CVTD2F	0x01	0x0a
CVTD2I	0x01	0x0b
CVTI2F	0x01	0x0c
CVTI2D	0x01	0x0d
MULT	0x01	0x0e
DIV	0x01	0x0f
EQF	0x01	0x10
NEF	0x01	0x11
LTF	0x01	0x12
GTF	0x01	0x13
LEF	0x01	0x14
GEF	0x01	0x15
MULTU	0x01	0x16
DIVU	0x01	0x17
EQD	0x01	0x18
NED	0x01	0x19
LTD	0x01	0x1a
GTD	0x01	0x1b
LED	0x01	0x1c
GED	0x01	0x1d

FUNCTIONAL SCHEMA AND BLOCKS DESCRIPTION

The DLX structure (code in appendix A) can be seen in two main blocks:

- Control Unit
- Datapath

The datapath presents some selection signals used to be controlled by the control unit, the instruction is read from the *instruction memory* using the *program counter* as address, the instruction is then stored in the *instruction register.* The above stage is the fetch stage that feeds the control unit and the datapath. Externally the DLX communicates with the IRAM for the program instructions and with a RAM for the operand storing and loading. The DLX instantiates only the control signals for these two memories, as interface.

Specific description of the blocks will be treated in the following sections.

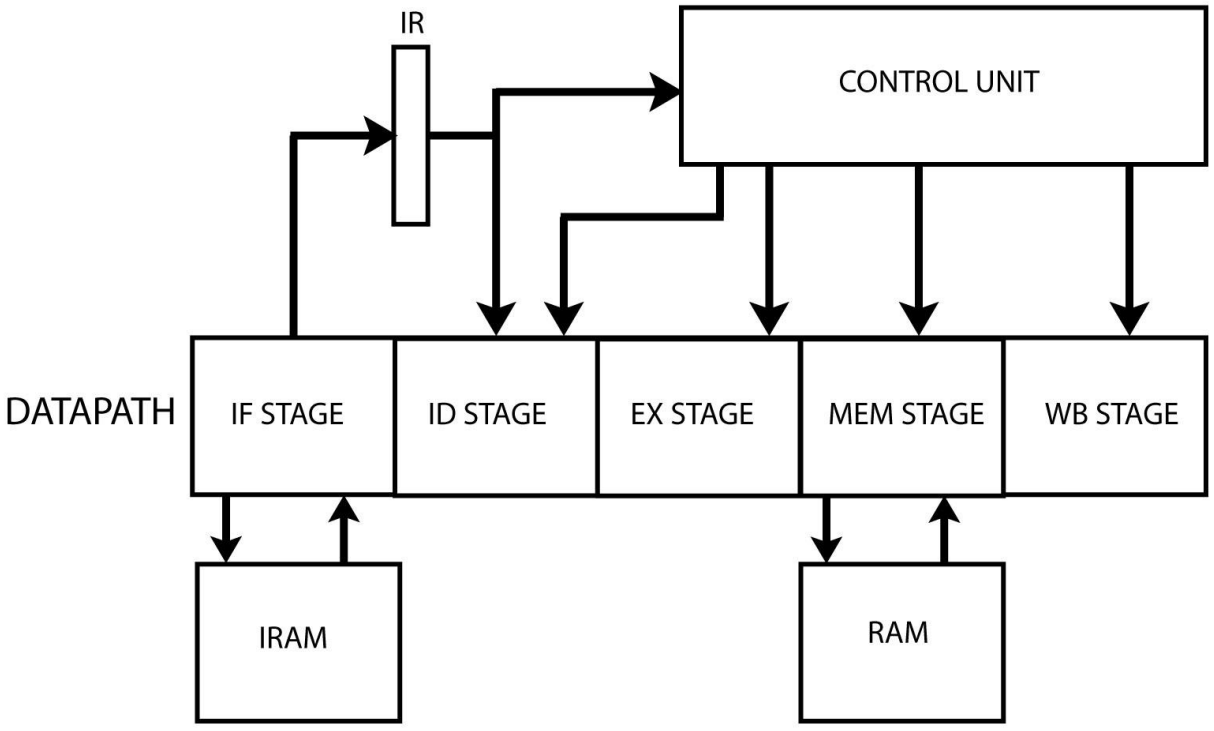


Figure 4 - DLX block diagram

Control unit

The control unit must control the datapath selection signals through a control word, this is done using a hardwired system that takes as input the OPCODE and FUNC field of the instruction register. Due to the fact that the DLX is a pipelined machine the CU splits the control word in four main *sub-word*, as consequence the overall control word is made by four different control *sub-word* each delayed by one clock cycle from the previous. The delaying mechanism has been performed by 3 sequential registers.

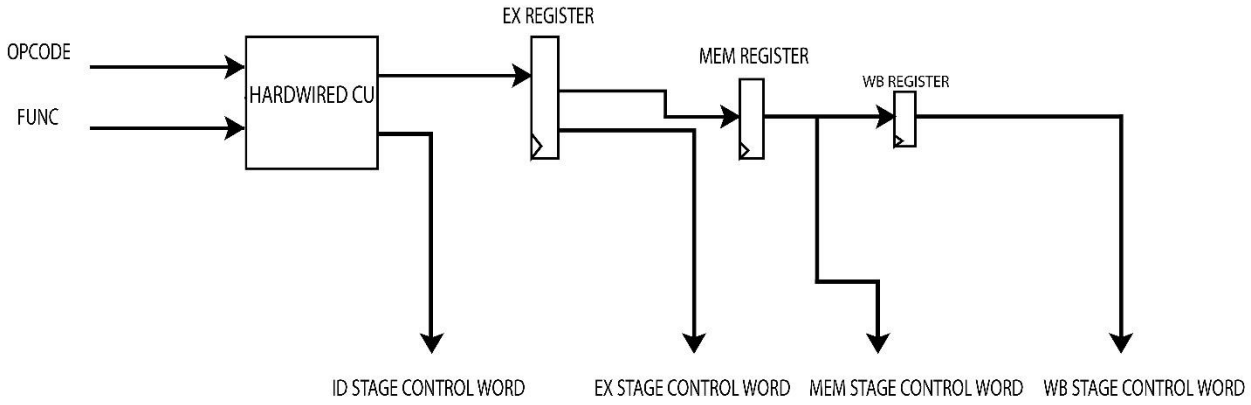


Figure 5 - Control Unit functional scheme

In the VHDL implementation (see appendix B) there are also some signals outputted by the cu for the datapath from the EX stage pipe used for the implementation of the data forwarding, better explanation will be given in the forwarding unit section.

DATAPATH

The datapath is the part of the DLX that performs the operations among data, this is the huge part of the processor. As explained in the previous chapter the datapath can be viewed in several stage, one per pipe stage. The description of this component will be done accordingly with this division.

Fetch stage

The fetch stage is the first part of the datapath that takes care of the instruction pick, this stage has the goal of fill the IR with the right instruction. This stage will feed both the control unit and the following datapath stage. In order to pick the instruction from the IRAM, a counter is required called *program counter (PC),* this will be used as address of the instruction memory and will be updated at each clock cycle incrementing its value four by four. The value of the next PC is stored in a specific register *(NPC).*

In order to perform jumps it's possible to choose the PC value between the NPC and another value coming from the ALU, this selection is done by a multiplexer. The control bit of the mux is generated by the jump logic at execution stage.

In order to correctly feed the decode stage a logic is required to prepare the values of addresses for the registers and immediate. The logic block recognizes the type of instruction (I, J, R) and acts in an according way generating the signal to feed the next stage.

The VHDL code of this part (appendix C) is described apart from the rest of the datapath (appendix D).

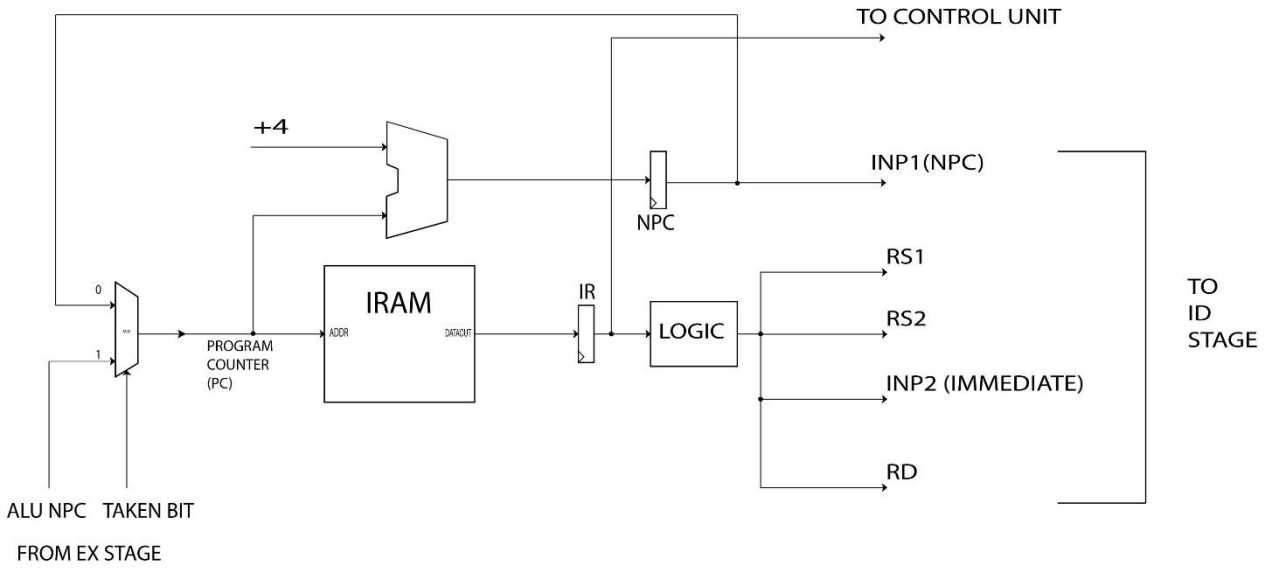


Figure 6 - Fetch stage block diagram

Decode stage

The decode stage prepares the operands for the execution stage, due to the fact that the DLX supports operations between integers and floating-point numbers three register files are instantiated: integer, float single-precision and double-precision. All the words inside the register file are 32 bit long, so 5 bits of address is required. In case of double precision numbers, a variable takes 2 register to be stored. Due to the fact that the RFs are read from decode stage and written by the writeback stage the synchronization of the read and write has been improved for both operations in the same clock cycle, the read is performed on the rising edge of the clock and the write on the falling edge of the clock. In this way is possible to correctly read and write the RF in the same clock cycle without occurring in read after write hazards.

The two 5 bits address signals are provided by the fetch stage and are in common among all register files, the enable of the output it's up to the control unit through the generation of 2 bits control (RD1, RD2).

To provide a unique representation of data to next stages all the operands of the register files and the NPC immediate are extended to 64 bits, except for the double-precision RF. Also, the immediate need to be extended, so a dedicated network is inserted in this stage. The network is driven by two bits of the CU to select the extension from 16 or 26 bit in signed or unsigned notation. The extended values of the immediates are stored in two registers in order to feed next stage (EX).

The WRs controls and the DATAIN word of the registers are handled by the writeback stage. The addresses for write comes from fetch stage and are delayed of 3 clock cycles through registers.

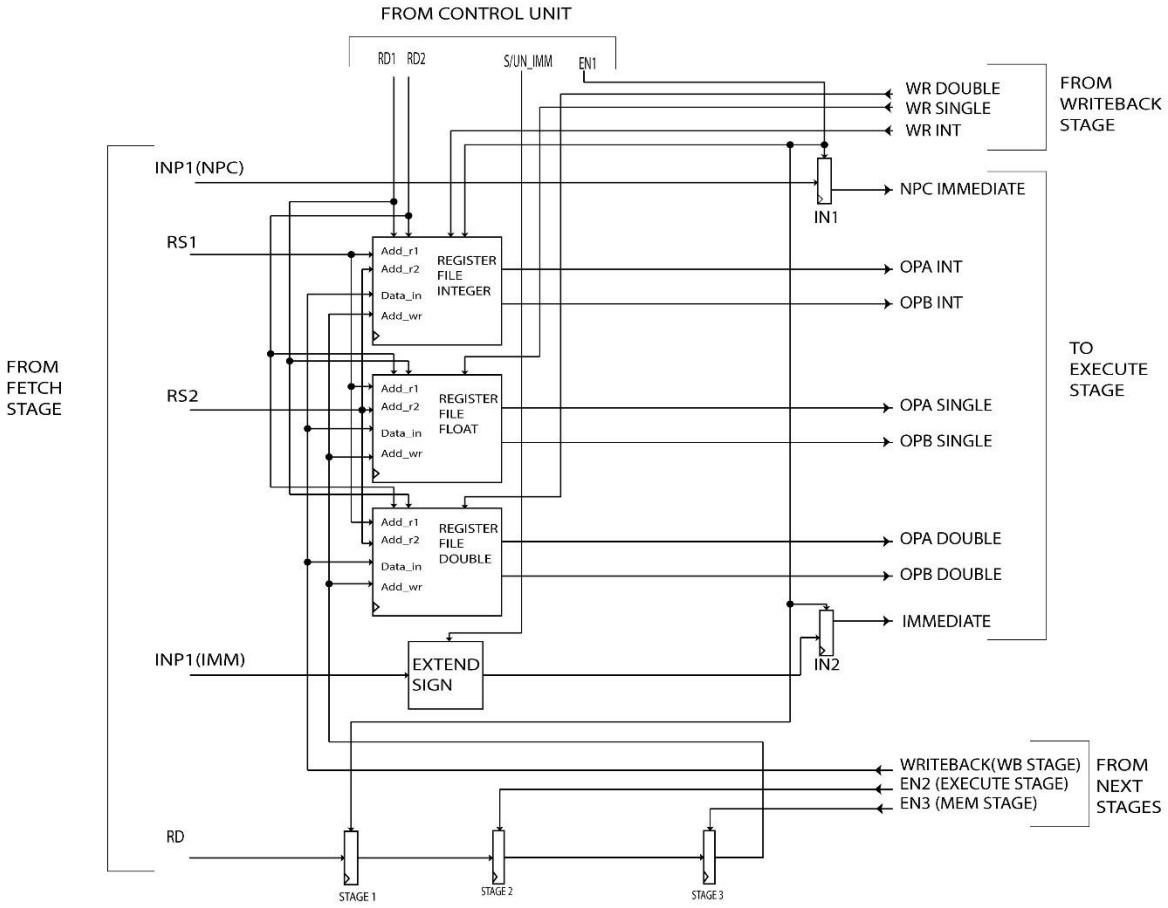


Figure 7 - Decode stage black diagram

Execute stage

This stage executes one or more operation between the data. The core of this stage is the arithmetic logic unit (ALU), which performs the computation. In order to correctly feed the ALU according with the current instruction an array of mux is needed to select the correct operands. In particular there are three sets of mux the first one (two muxes, top left) selects the operands between the tree registers coming from the decode stage, the control is up to the CU using RF_SEL1 and RF_SEL2 (both 2 bits length). Then the second couple of muxes selects the entry between the selected operands and the two immediate (NPC and IMMEDIATE) from decode stage. Finally, the third stage of muxes is a set of four muxes used to implement the forwarding. These are used to feed back the operands coming from next stages (MEM or WB) or from the ALU out. The control of these muxes is up to the forwarding unit, which is able to recognize a RAW hazard and solve it setting the correct configuration of the muxes. A more accurate description of this unit will be treated in a dedicate chapter due to its complexity. After the array mux the operands has been selected and ready for the computation. The ALU presents some dedicated control signals that allows the CU to select the right operation according to the current instruction. Also, the ALU will be better described in a dedicated chapter due to its complexity. The arithmetic unit presents three outputs that are the ordinary ALU output, the ALU NPC and the FPS bit. The ordinary output is connected to a register and used to feed the next stage, in particular this out could be used for a normal computation or to compute the effective address of the RAM. The ALU NPC is directly connected with the integer ALU and is used to compute a new value of program counter in case of complex jumps, this output will be used to feed back the fetch stage. The FPS bit is used to recognize comparisons among floating-point numbers. In particular, this bit will be set if the result of a comparison is positive. The FPS bit is stored in a flip-flop to save its value, this FF is driven by a subpart of the FP_CONTROL word in order to enable the reading if necessarily. Concurrently the zero detection checks if the value of the operand A is equal to zero and feeds the jump logic (code in appendix G). This entity is driven by three control bits given by the CU, that specify if which type of jump is required, in particular six situation could occur:

- No jump , PC = NPC
- Conditional jump between integers, check zero detection equal zero or not (two case).
- Conditional jump between float, check FPS is zero or not (two case).
- Unconditional jump.

The jump logic generates the bit that decide if the branch is taken or not. This bit will be used to feed the mux of the fetch stage, and decide the value of the PC. Concurrently feeds also the CU to decide the flush of the control word.

The operand B is directly connected to the DATAIN memory through a register in order store its value when required.

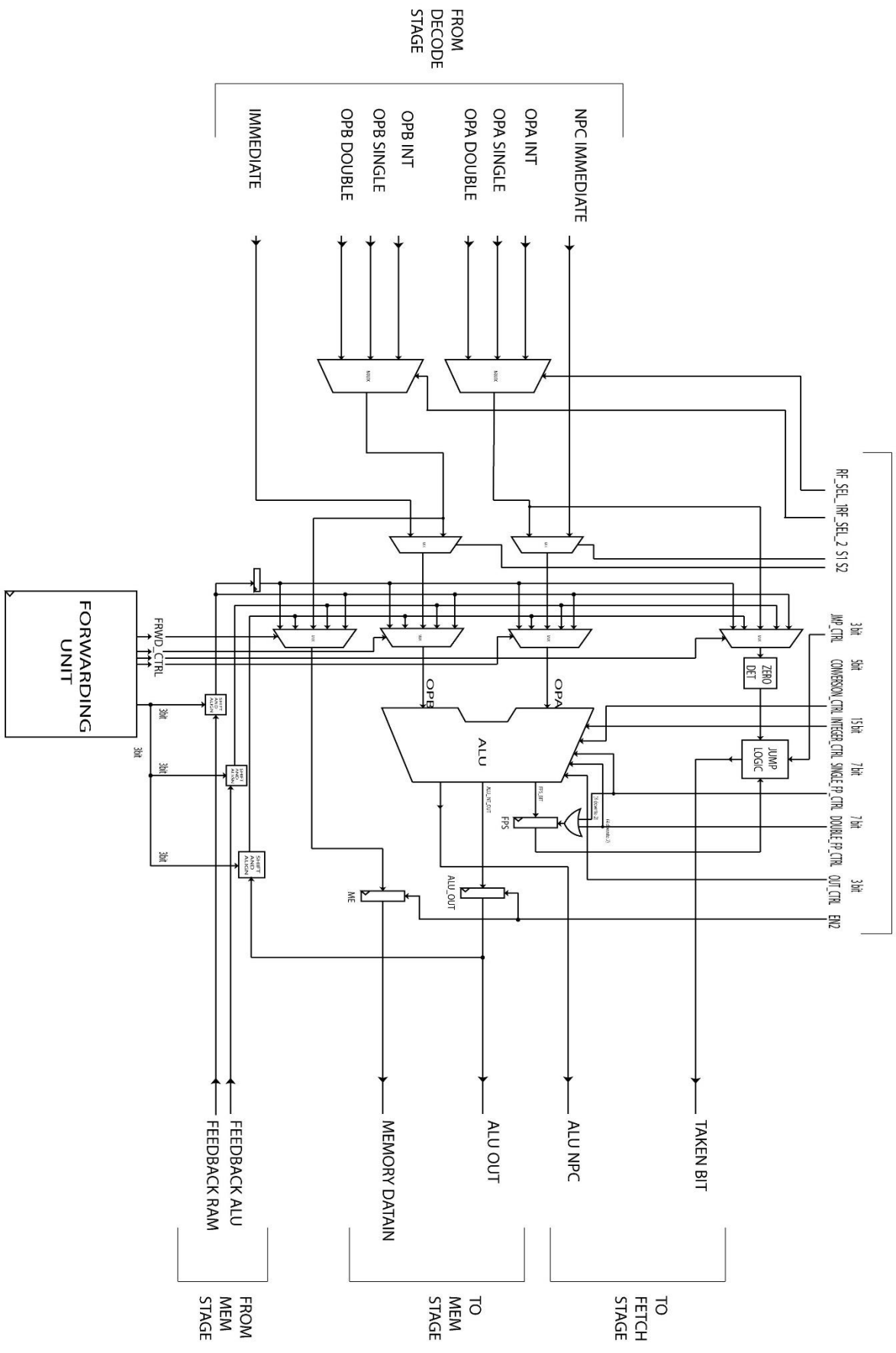


Figure 8 - Execute stage

Memory stage

This stage accesses the memory, if required by instruction. The base address is computed by the ALU from the previous stage. The memory is controlled by the control unit through signals:

- RD, specify the reading operation.
- WM, specify the writing.
- RD_double, access two consecutive words (8 cells) in reading mode.
- WR_double, write two consecutive words (8 cells).
- ALIGN_AND_CTRL: specify if the number of bytes to be written (byte, half-word, word).
- EN: Enable the memory.

In order to read in the right clock cycle, the RAM reads and writes the data in the falling edge of the clock. This choice will negatively affect the critical path (more detailed discussion will be treated in the following chapters).

Simultaneously two register are used to delay the signals to the next stage, in particular the ALU out and the shared control signals with the WB stage.

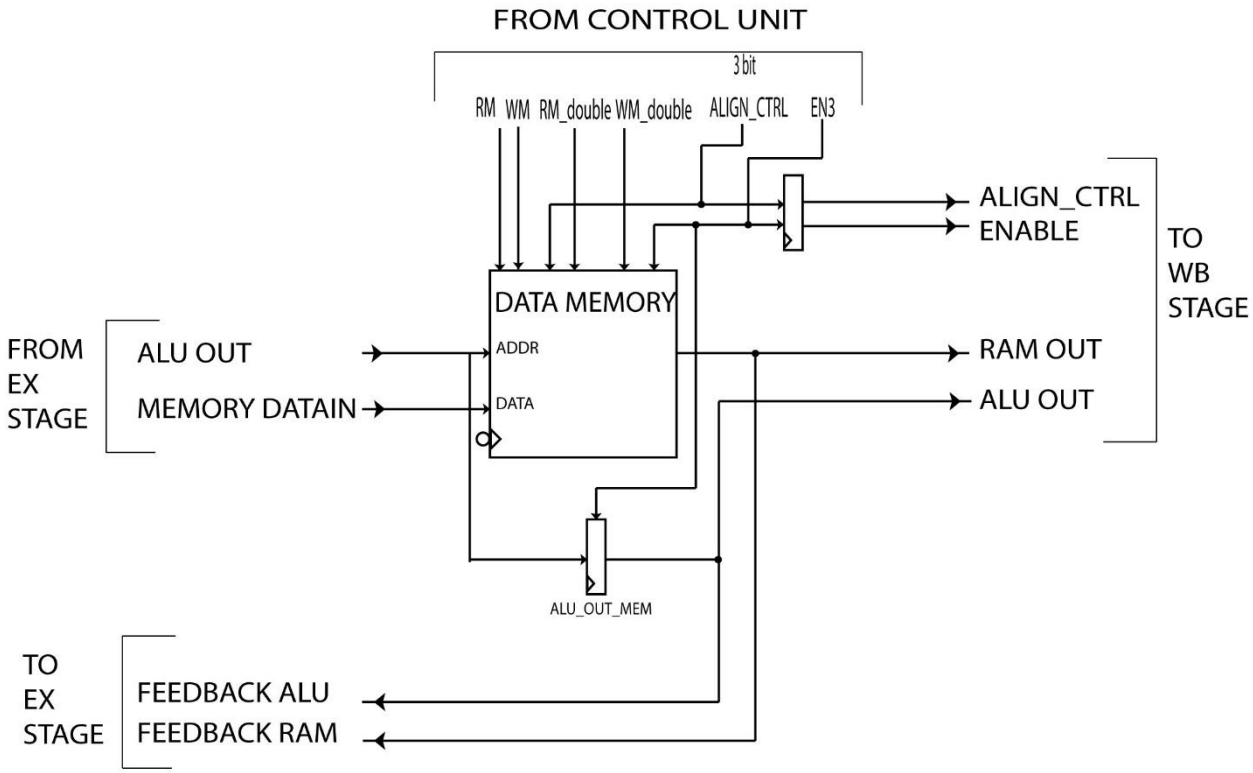


Figure 9 - Memory stage

Writeback stage

This stage writes in the specific register the value of the computation obtained. The multiplexer selects the RAM or the ALU operand. Before the feedback, an align block is applied to re-align operand coming from memory or apply shift to the value. The register out is used to store the value of the computed operand stable for the output of the DLX.

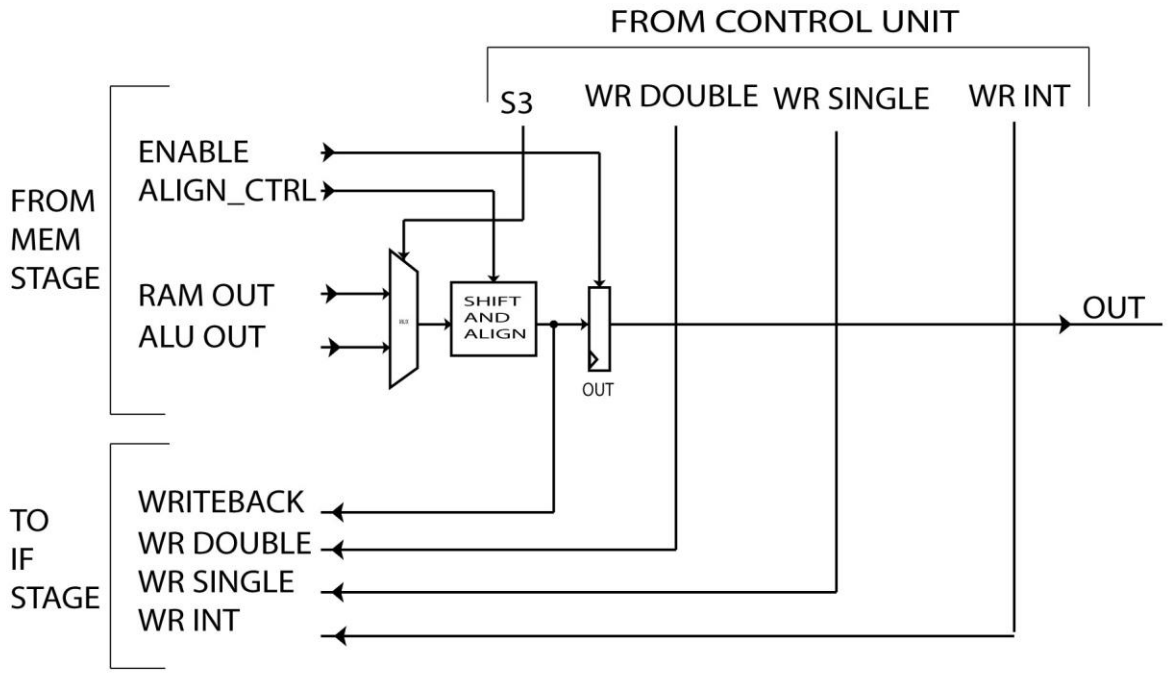


Figure 10 - Writeback stage

ALU

This chapter aims the description of the Arithmetic Logic Unit (see appendix E). This unit is composed of three main blocks, the integer ALU, float-single ALU and float-double ALU. The last two are very close from a structural point of view indeed the only difference stays in the length of the data. Consequentially, the description of the floating-point units will be unique. The main units are controlled by the signals generated from the CU.

The two input operands pass through two conversion units in order to implement conversions among the different representations. Another conversion block that is placed at the output of the single floating point. This choice has been taken in order to perform the integer division through floating ALU. In particular the integer operands are converted firstly to float representation by the converters. Then the division is performed by the floating divider, finally quotient is reconverted back to integer. This choice has been taken in order to reduce the amount of area, using just one divider.

The conversions supported by the converter units are:

- FP single to integer.
- FP single to double.
- FP double to integer.
- FP double to single.
- Integer to double.
- Integer to single.
- No conversion (for other operations).

In order to reduce the switching activity, the three units have an all zero pattern to feed the inputs and the output when no operation has been required by the control unit.

The global ALU has three outputs:

- NPC OUT
- ALU OUT
- FPS

NPC OUT is directly connected with the integer ALU, this is used to feed the program counter as explained before.

FPS out bit feeds the Floating Point Status flip-flop, this is set when one of the two floating unit has a true comparison. So, an or gate is placed.

ALU OUT is the main output of the unit, is connected to a multiplexer driven by 3-bit control that selects the output between:

- Integer out.
- Floating-point single out.
- Floating-point double out.

- Floating-point single to integer converter (for division).
- Converted OPA.
- Converted OPB.
- Converted OPA + 4.
- Extended integer comparison.

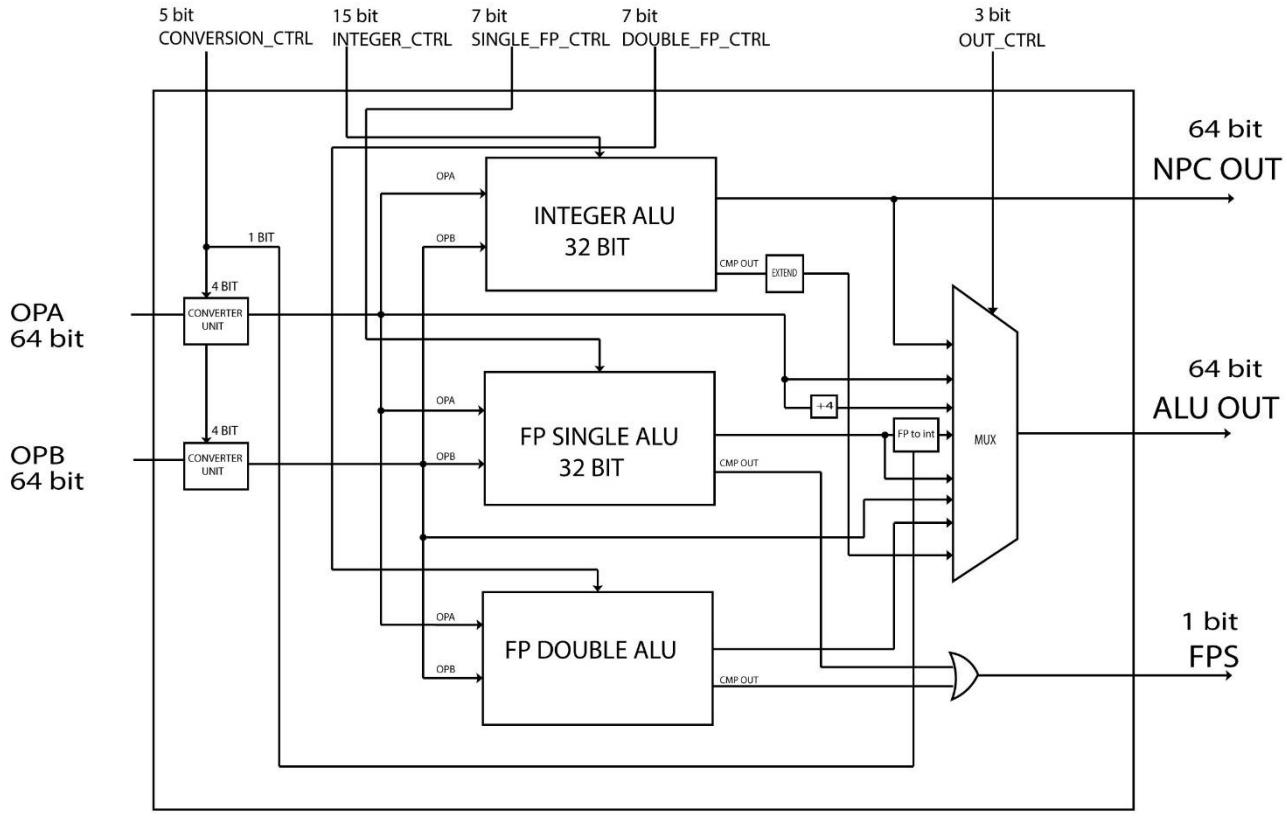


Figure 11 - ALU general structure

The following sections explains deeply the detail of the ALUs.

Integer ALU

Integer unit performs all the operation between integer numbers. In particular the supported operations are:

- Addition unsigned.
- Addition signed.
- Subtraction unsigned.
- Subtraction signed.
- Multiplication unsigned.
- Multiplication signed.
- Logic operations.
- Shifting operations.
- Comparisons.

Division is implemented through floating unit.

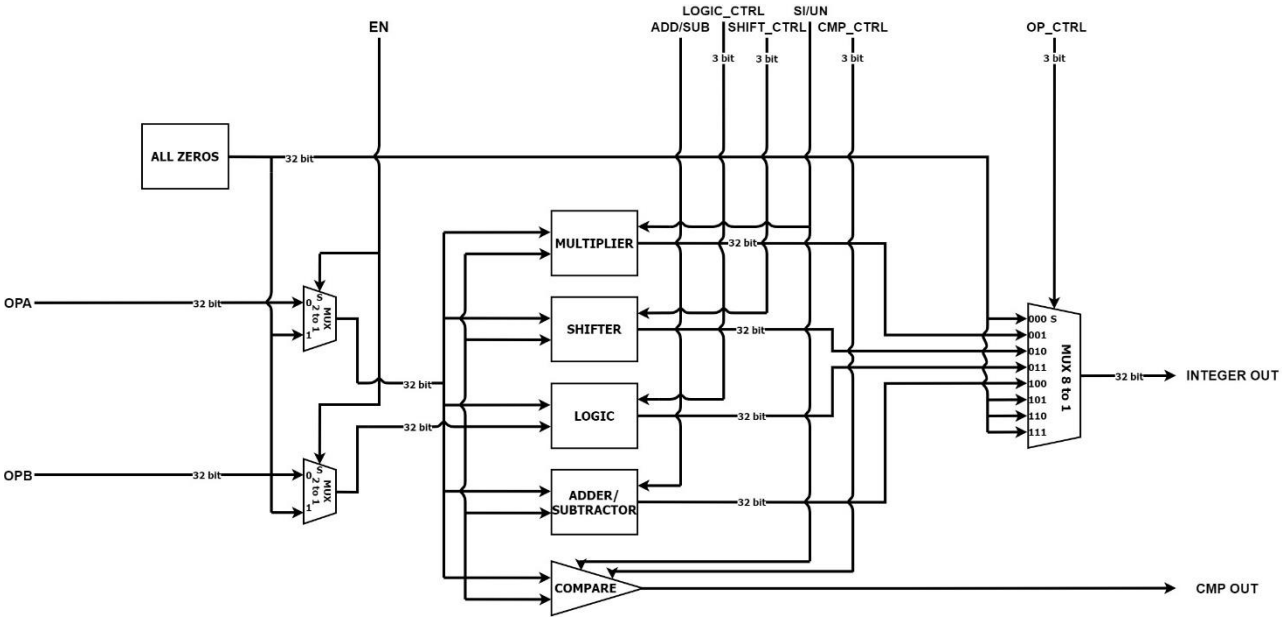


Figure 12 - Integer ALU

The input operands are shared among all units and are the lower 32 bits of the given words.

The two muxes connects the all zero patterns when the ALU is not used. This is done to avoid propagation of data when this unit is not used for execution (the control is given by the CU).

The units present two output signals:

- Comparison out
- Integer out

The comparison output is connected to the comparator unit that through 3-bit selection performs a specific comparison.

Integer out selects the output among:

- ADD/SUB unit.

- MUL unit.
- Logic unit.
- Shifter.
- All zero pattern.

Adder subtractor unit

This unit is implemented using the Pentium IV adder. The selection between add and sub is performed using the carry in bit.

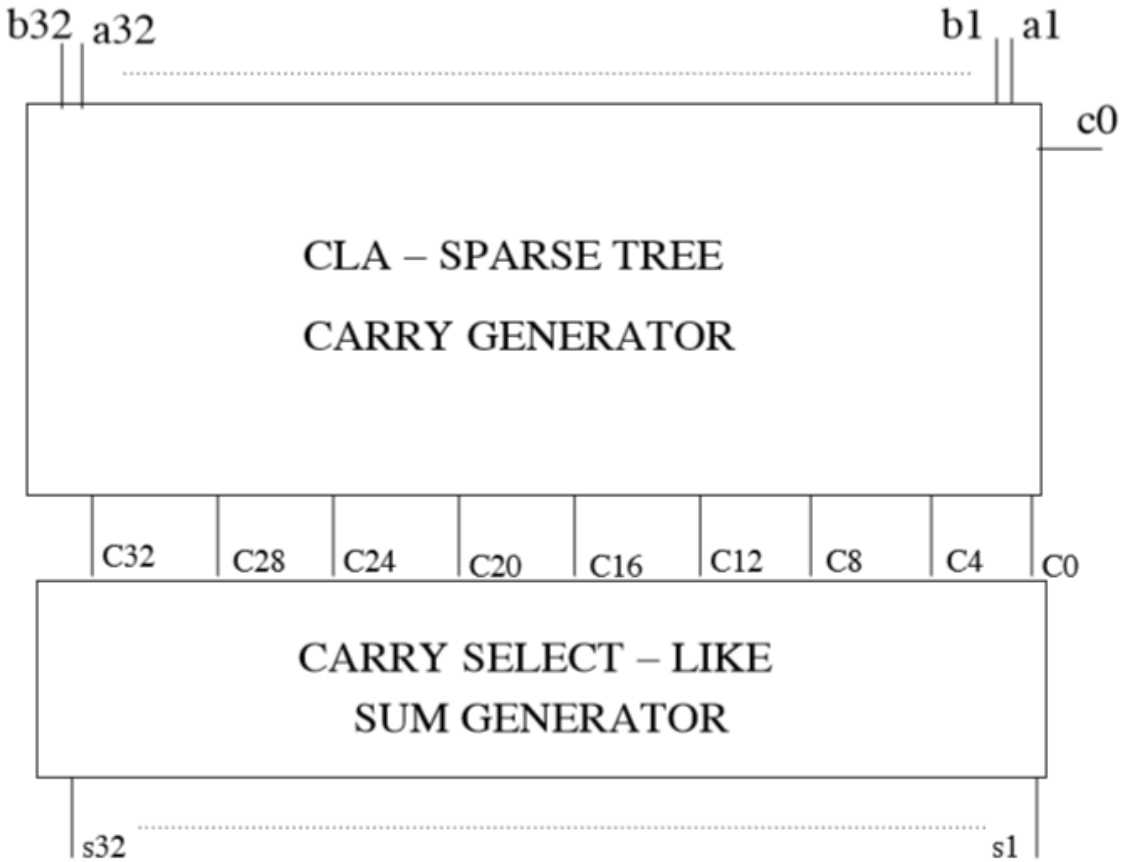
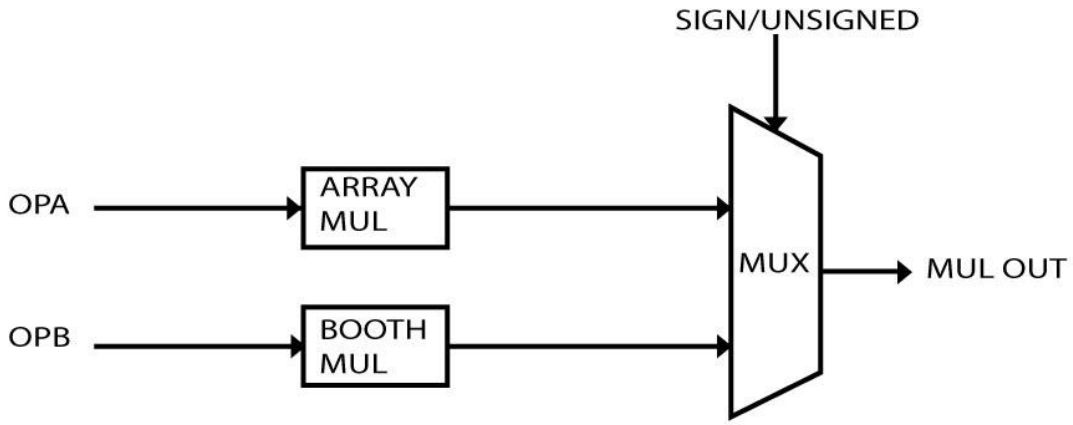
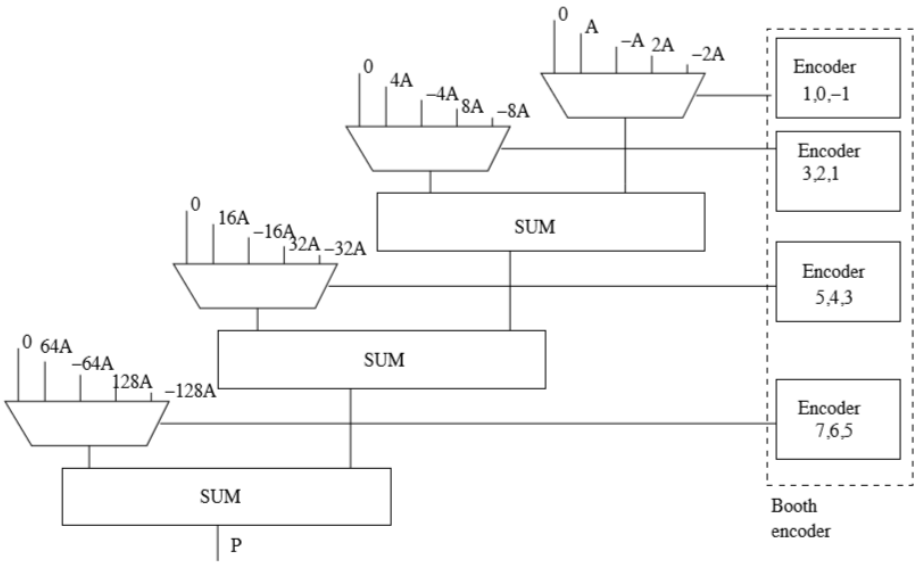


Figure 13 - Pentium IV adder

Multiplier unit

This unit performs multiplications between signed or unsigned numbers, a selection bit is used to identify the type of multiplication. In particular for the signed numbers a booth multiplier performs the operation. The unsigned numbers instead pass through an array multiplier. The two inputs operands are shared between the two multipliers and a multiplexer selects the output.





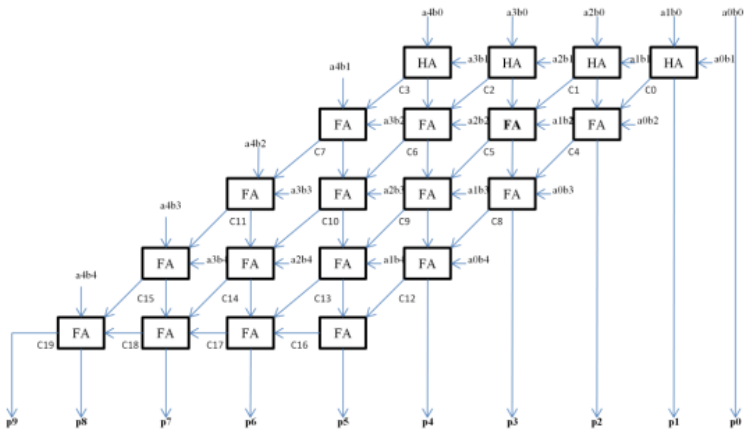


Figure 14 - Multiplier

Shifter

The shifter performs the shifting of operand A by the value of operand B, the supported shifts are:

- Logical shift left.
- Logical shift right.
- Arithmetical shift right.
- Rotate left.
- Rotate right.

A 3-bit signal selects the output.

This unit is behaviourally described except for the rotate operations, which are implanted through the logical shift. Starting from the OPA word, an extended version is created as shown in figure.

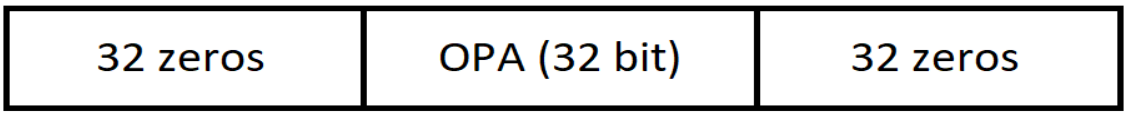


Figure 15 - Operand A rotate extension

Then the shift (right or left) is applied. Finally, a bitwise AND is applied between the central word and the left one (rotate left) or right one (rotate right). Therefore, the maximum accepted value for a rotate is 32 (this modulation will be up to the compiler).

Logic unit

This unit performs the logical bitwise operations. Supported operations are:

- And
- Nand
- Or
- Nor
- Xor
- Xnor

The inputs are share among all bitwise operations and, through 3-bit selection signal, the output is connected.

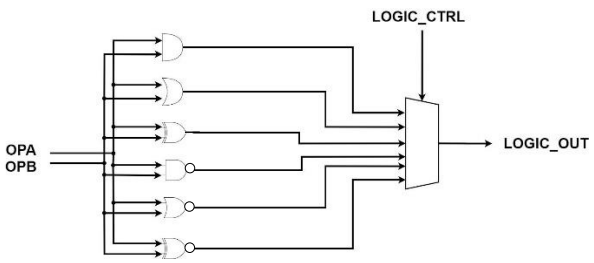


Figure 16 - Logic unit

Comparator unit

The integer comparator unit is able to recognize both signed and unsigned comparisons. In particular is possible to perform the following comparisons:

- A>B.
- A>=B.
- A<B.
- A<=B.
- A=B.
- A!=B.

Both the signed and unsigned comparisons base on the same hardware device, structure as in figure 18.

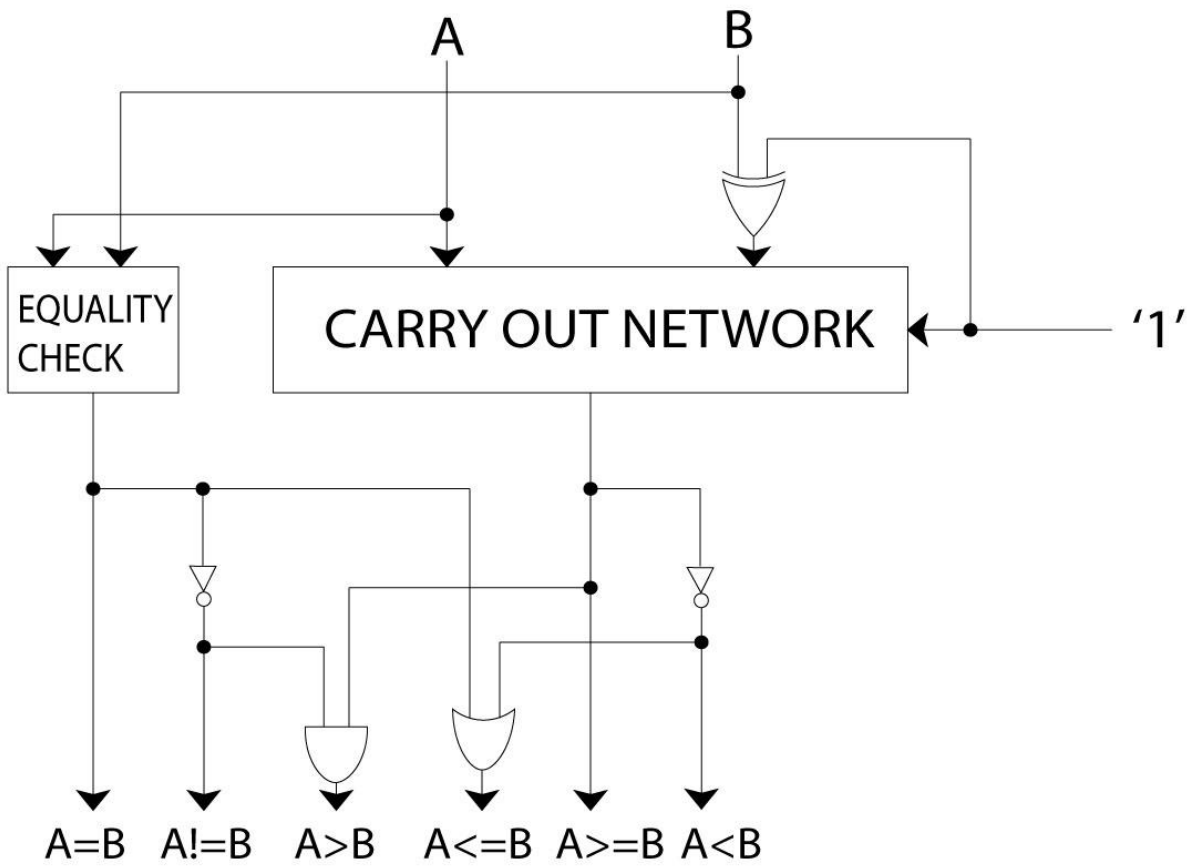


Figure 17 - Unsigned comparator

This comparator works for the unsigned numbers, and bases the comparisons on the difference through a carry network. The carry out-bit is used to compute the inequality and an equality network is instantiated for the others.

This network has been extended also for the signed numbers using a sign/unsigned control bit. The control on signed numbers is behaviourally described for discording signs. For according signs, this unit is used.

Through a multiplexer driven by 3-bit control a specific comparison is connected to the output.

Floating-point units

This section will treat the structure of both floating-point single precision and double precision units. This units follow the IEEE 754 standard. In particular, the representations for double and single precision numbers are shown in figure.

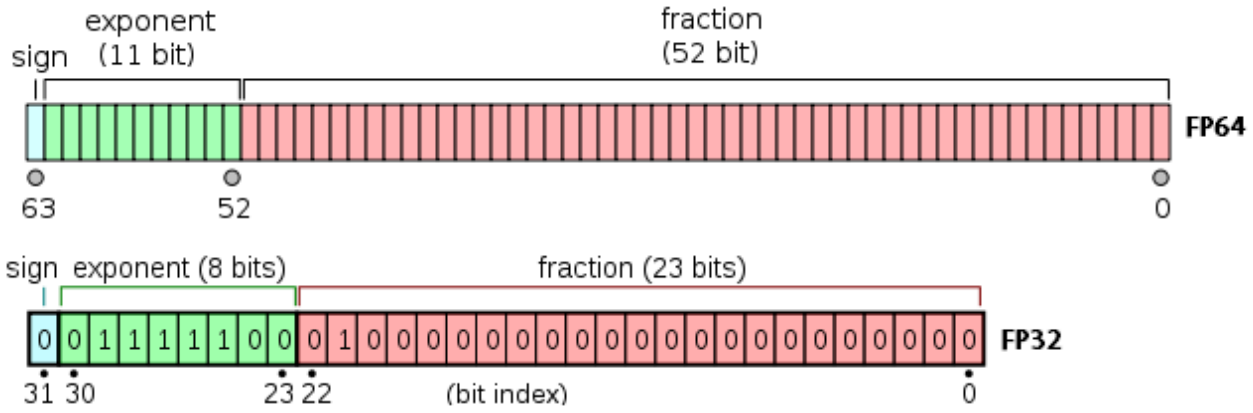


Figure 18 - Floating-point representations

The first MSB is the sign bit, then the rest of the word is used to store the exponent and the mantissa. For both, the representation is sign-module one. So, the mantissa is assumed to be always positive. The exponent is considered biased so an offset has been applied.

Due to the fact that floating numbers could have more representations for the same value, a normalization is required. The standard assumes and hidden '1.' as the first bit of the mantissa (real mantissa is 1.|M|).

Therefore, a floating-point number can be written as:

$$X = (-1)^{sign} \cdot 1.|M| \cdot 2^{EXP-BIAS}$$

The single and double units differ for the bit-length and for the divider unit that was not implemented in the double due to its complexity. All the operations for both units are performed in a single clock cycle.

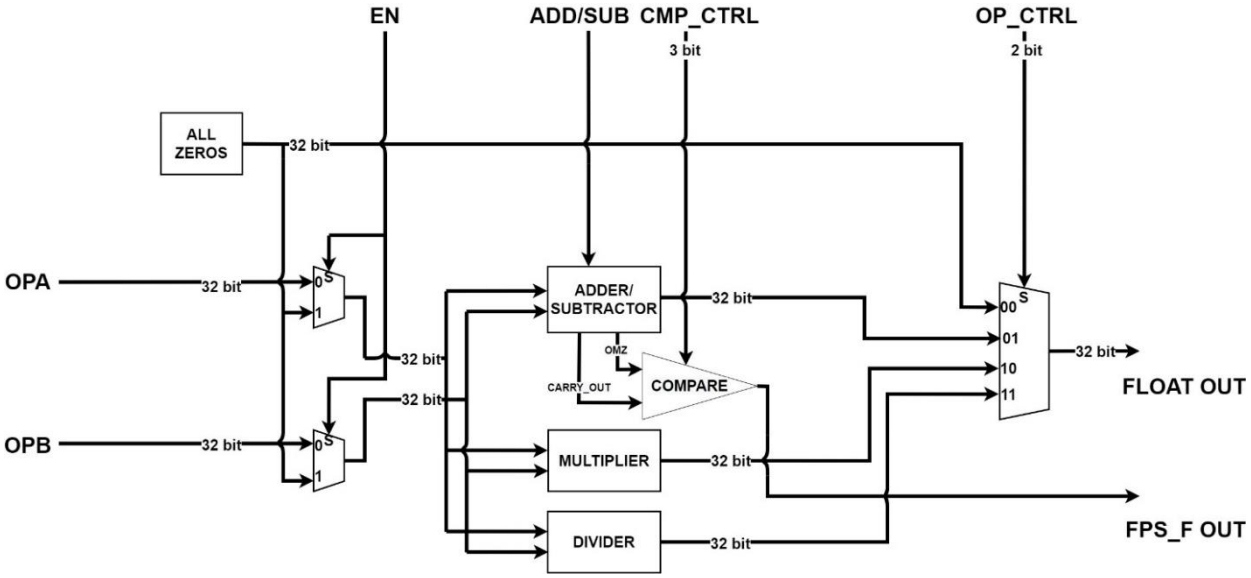


Figure 19 - Float single unit

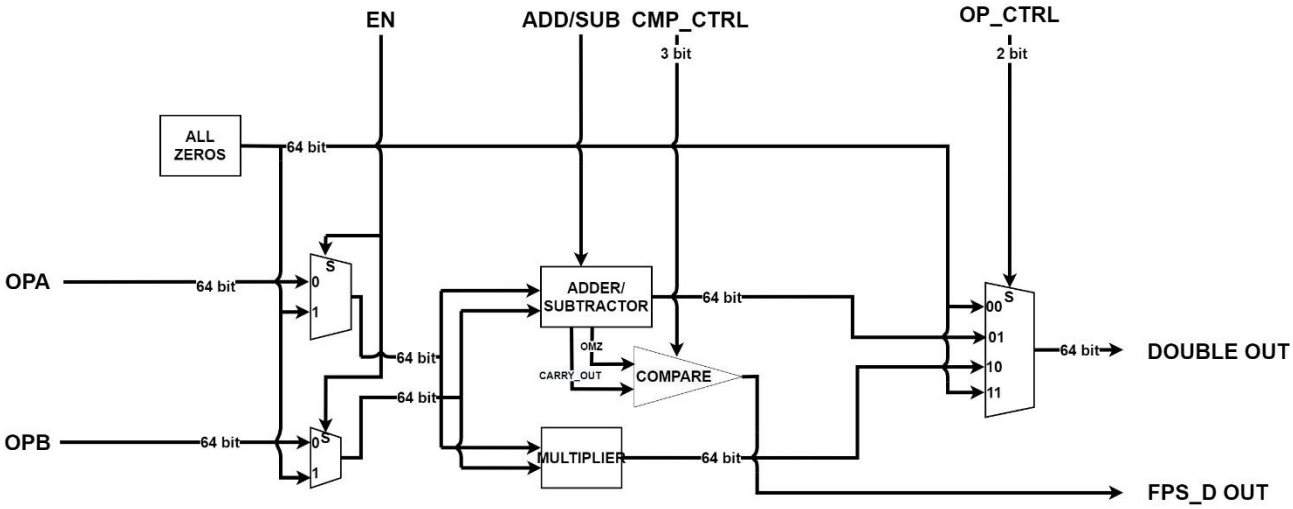


Figure 20 - Float double unit

Both units have an enable signal that connect all zeros to the inputs to avoid switching propagations when the unit will not be used.

Beyond the divider the two units presents exactly the same structure so the description will be unique.

Floating point additions, subtractions and comparisons

Giving the definition of floating-point numbers using the IEEE 754 standard is possible to build a network for add or subtract floating-point numbers. In order to sum two numbers a first alignment is required. First stage is done using a comparison network between the exponents, this network will define the shift amount for the mantissas. The greatest exponent will be used for the final result.

After the alignment, the mantissas are summed up as fixed-point numbers. In this stage the computation can be performed as sum or as subtraction, in this last, a two's complement is required due to the sign-module notation.

Finally, the mantissa the exponent and the sign are attached according to the standard and a post normalization network performs the normalization of the numbers shifting the hidden-bit in the right position and changing the exponent accordingly. This stage is implemented as a combinational network generating all possible shifts and checking the right one using a complementary network.

All the given operations are performed in one clock cycle.

A general scheme of the add sub is left.

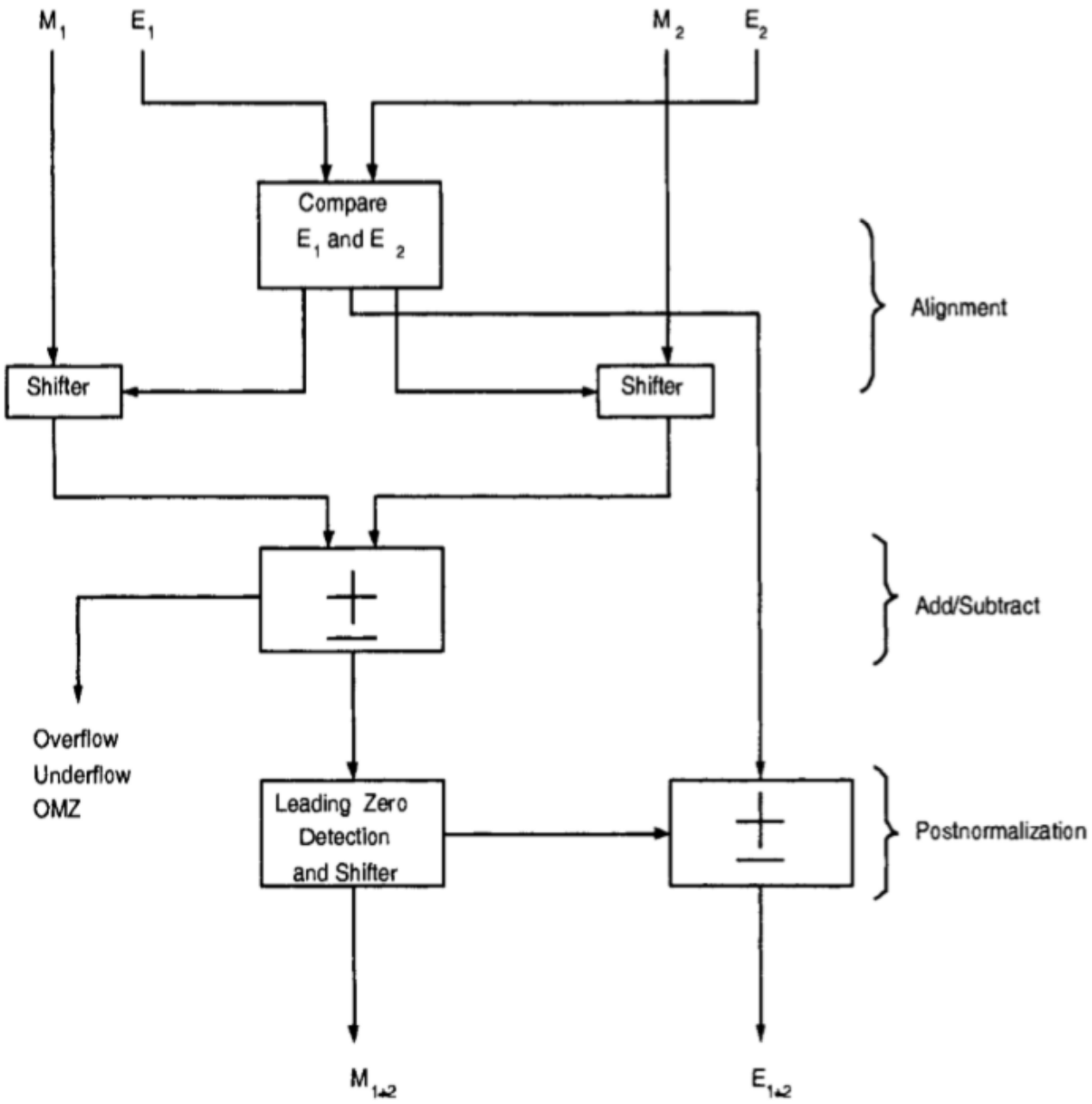


Figure 21 - Floating point add/sub network

The computation stage is implemented through a carry look-ahead adder. In this stage a carry-out is generated and also an OMZ bit, which flags the zero mantissa for the normalization stage. These two bits are also used to feed the comparator unit, when a subtraction is performed (the sub operation is set by the CU). Using these two bits, the float comparator can recognize floating-point comparisons. The principle is very similar to the integer comparator but, this time, the bits are produced by the sub stage. As before a multiplexer selects the specific comparison that goes in a specific output of the floating-point unit.

Floating point multiplication

The floating multiplication is performed summing the exponent as integers and multiplying the mantissas as fixed point. No alignment is required for this operation. After the computation, a normalization stage is required.

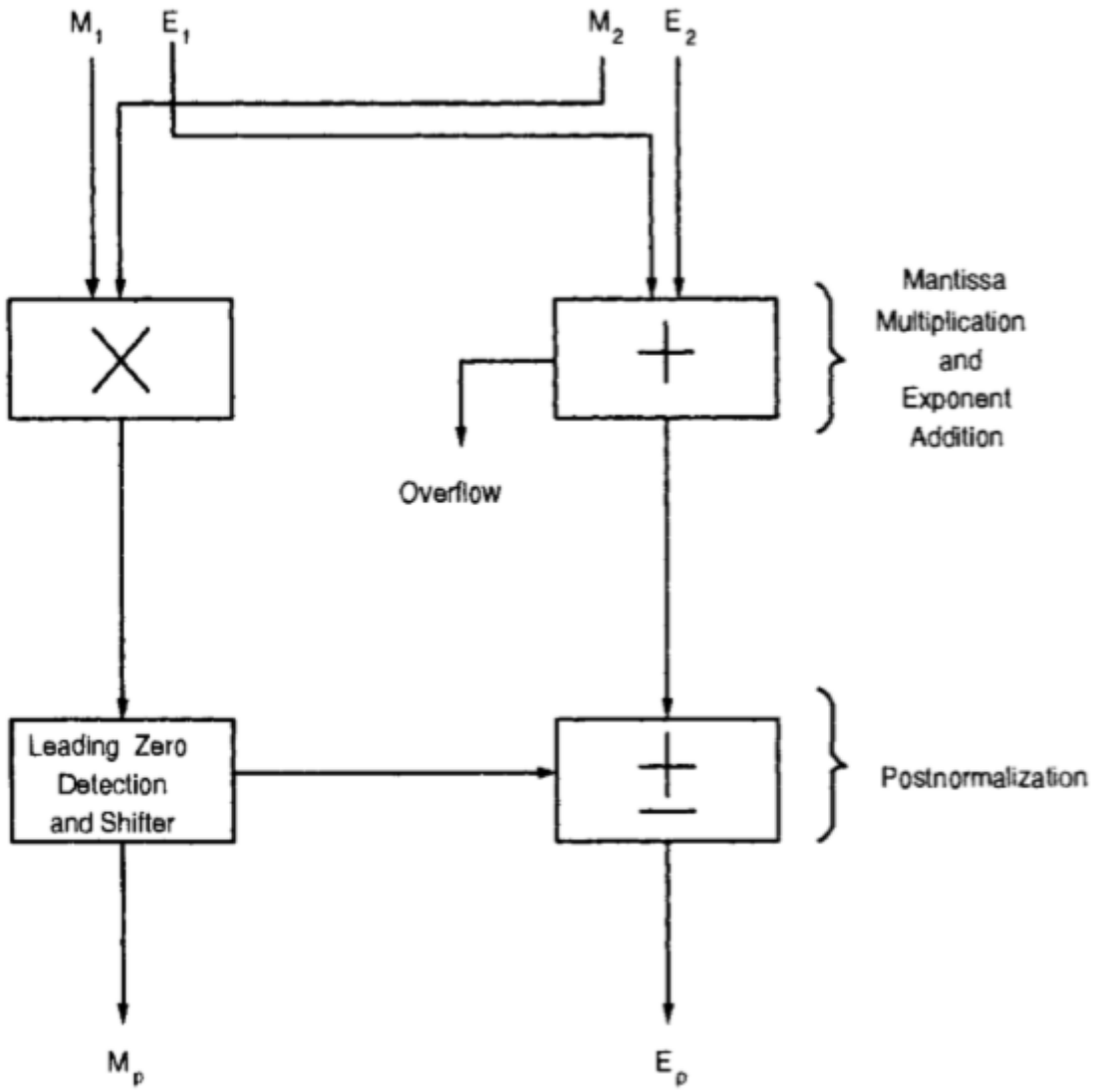


Figure 22 - Floating-point multiplication

The multiplication stage is performed through an array multiplier.

Floating-point division

The floating-point division is performed subtracting the exponent as integers, and dividing the mantissa as fixed point.

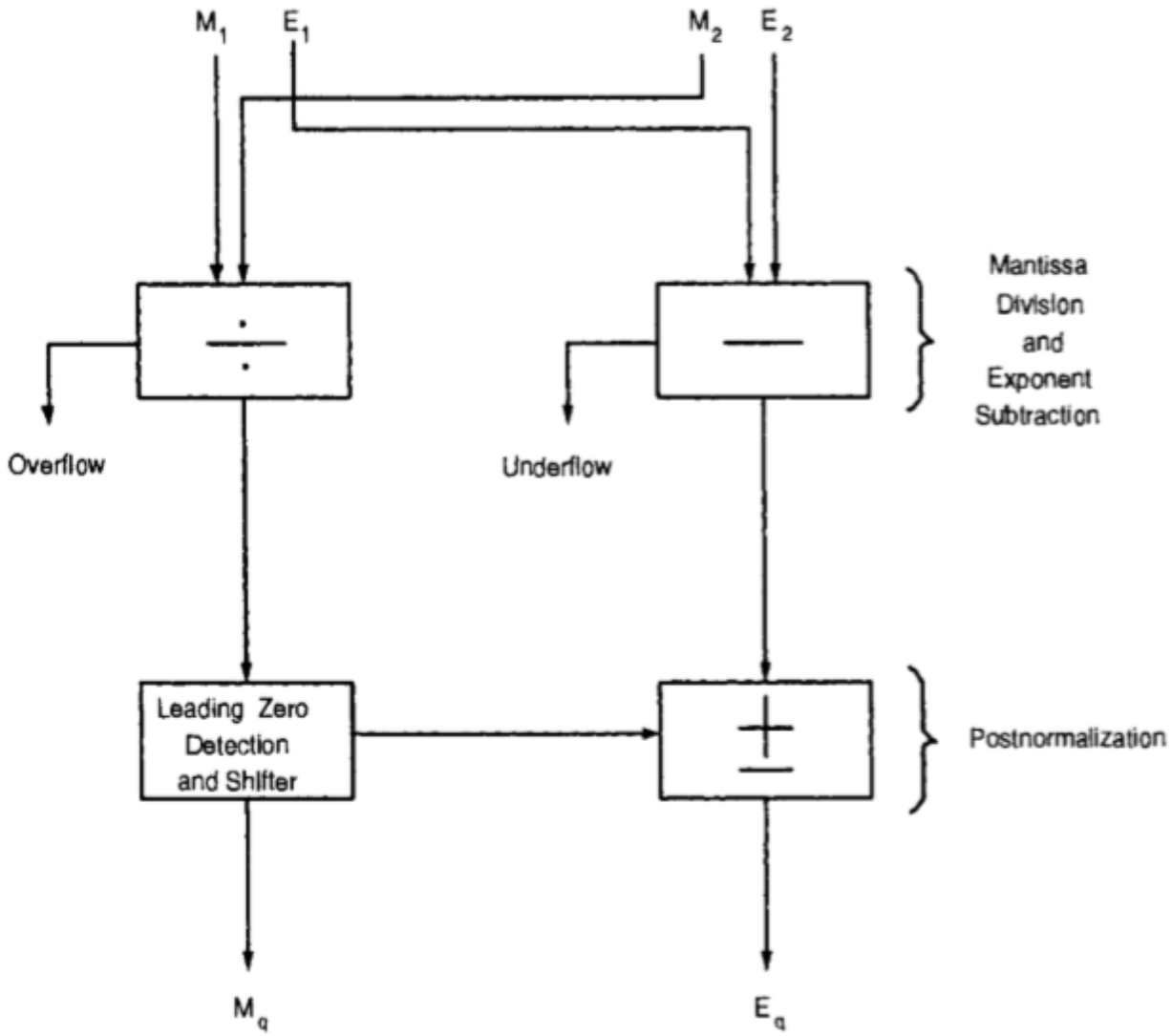


Figure 23 - Floating-point division

The mantissa division is performed through a non-restoring cellular array divider, which performs the fixed-point divisions in one clock cycle (purely combinational). A general scheme of this divider is left.

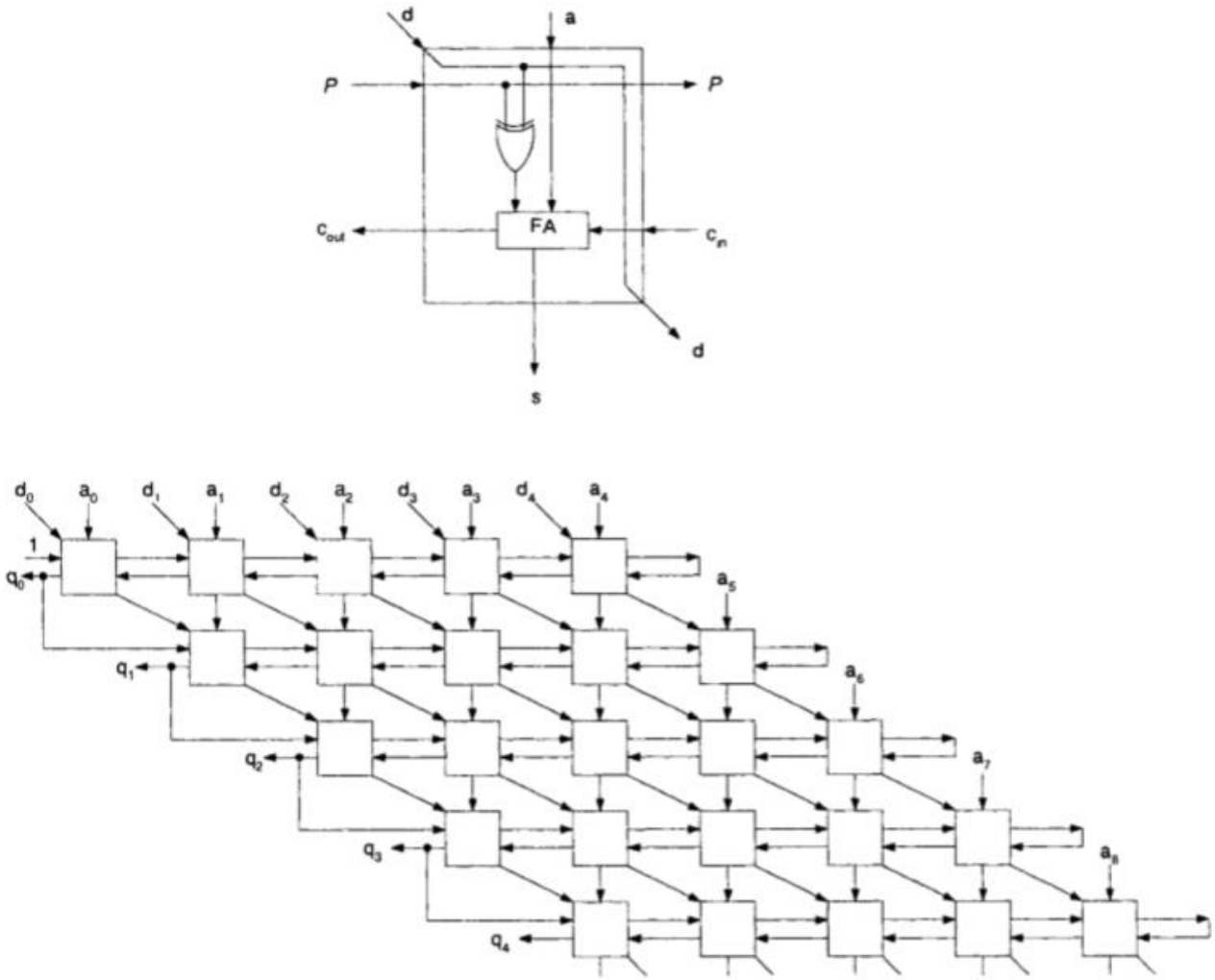


Figure 24 - Non-restoring cellular array divider

The zero-division case is represented by the data itself. The float representation can represent the +inf and –inf values setting the all one pattern, on the exponent and, all zero mantissa.

DATA FORWARDING

In order to avoid the RAW hazards, the data forwarding unit has been implemented (code in appendix F). This unit recognize a RAW at runtime and sets the forwarding control signals to one or more muxes at execution stage, in order to feed back a data steel not stored in register file.

According to the instruction set, 4 possible situations of RAW hazards can arise:

- RAW OPA.
- RAW OPB.
- RAW variable checked by the zero detection.
- RAW variable to be stored in memory.

So, four multiplexers have been instantiated to avoid these stalls.

Each mux has 5 entries, representing the position in pipeline of the operand to be read:

- ALU OUT at EX stage
- ALU OUT at MEM stage
- RAM OUT at MEM stage
- RAM OUT at WB stage
- No stall

The implementation of muxes is shown in figure.

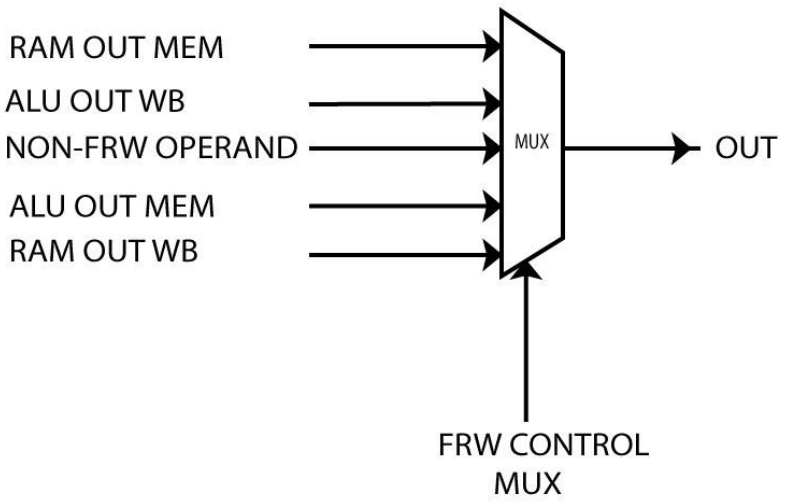


Figure 25- Forwarding multiplexer

The data forwarding acts at execution stage. So, supposing to have a fulfilled pipeline (5 lines) there are only two complementary stages that arise a stall. When the operand at EX stage requires an operand in MEM stage, or when EX stage requires operand is in WB stage. All the next iterations will be solved by the natural flow of the DLX.

An example is here left.

INSTR A	IF	ID	EX	MEM	WB				
B reads from A	---	IF	ID	EX	MEM	WB			
C reads form A	---	---	IF	ID	EX	MEM	WB		
NO STALL ISSUES	---	---	---	IF	ID	EX	MEM	WB	
WITH INSTR A	---	---	---	---	IF	ID	EX	MEM	WB

Figure 26 - Stall example

In order to recognize a stall, the data forwarding need the following signals:

- RF_SEL_1, read from specific RF.
- RF_SEL_2, read from specific RF.
- RS1, source register address 1 delayed at EX stage.
- RS2, source register address 2 delayed at EX stage.
- RD1, read enable 1 delayed at EX stage.
- RD2, read enable 2 delayed at EX stage.
- RD (EX stage), register address write at EX stage.
- RD (MEM stage), register address write at MEM stage.
- WM (EX stage), write memory anticipated at EX stage.
- RM (MEM stage), read memory.
- WR_INT (EX stage), WR enable integer anticipated at EX stage.
- WR_FP (EX stage), WR enable FP anticipated at EX stage.
- WR_DOUBLE (EX stage), WR enable double anticipated at EX stage.
- JMP_CTRL, jump control.

Knowing these signals a behavioural description for each mux-control has been performed in the data-forwarding unit. The stall arises when the instruction at EX stage try to read the value of a variable in MEM stage or in WB stage. Depending on the situation (branch, memory access or ALU operation), the stall is recognized and solved.

Furthermore, in order to forward the correct data each feedback line is pass through a shift and align block (same block as writeback stage). Therefore, the forwarding unit anticipates the ALIGN_CTRL signal at EX stage and sets the correct value to each block.

As mentioned, some input signals of the unit need to be propagated from the decode stage as RS1, RS2, RD1, RD2 in order to verify the effective reading. These signals are then propagated from the ID to EX from the forwarding itself. Same is applied for the signals that needs to be anticipated at EX stage, as WM, WR_INT, WR_FP, WR_DOUBLE. These signals are extracted from the control word propagation registers inside the control unit. The signals that requires a propagation from EX stage to MEM or WB stages are delayed internally.

It's important to underline that inside the forwarding unit all the signals at EX stage refers to the current instruction (the one that could require a data forwarding), call it i-th. The signals at MEM stage refers to the previous instruction, (i-1)-th. The at WB refers to the previous more (i-2)-th. As said, the (i-3)-th instruction cannot produce an hazard with the i-th.

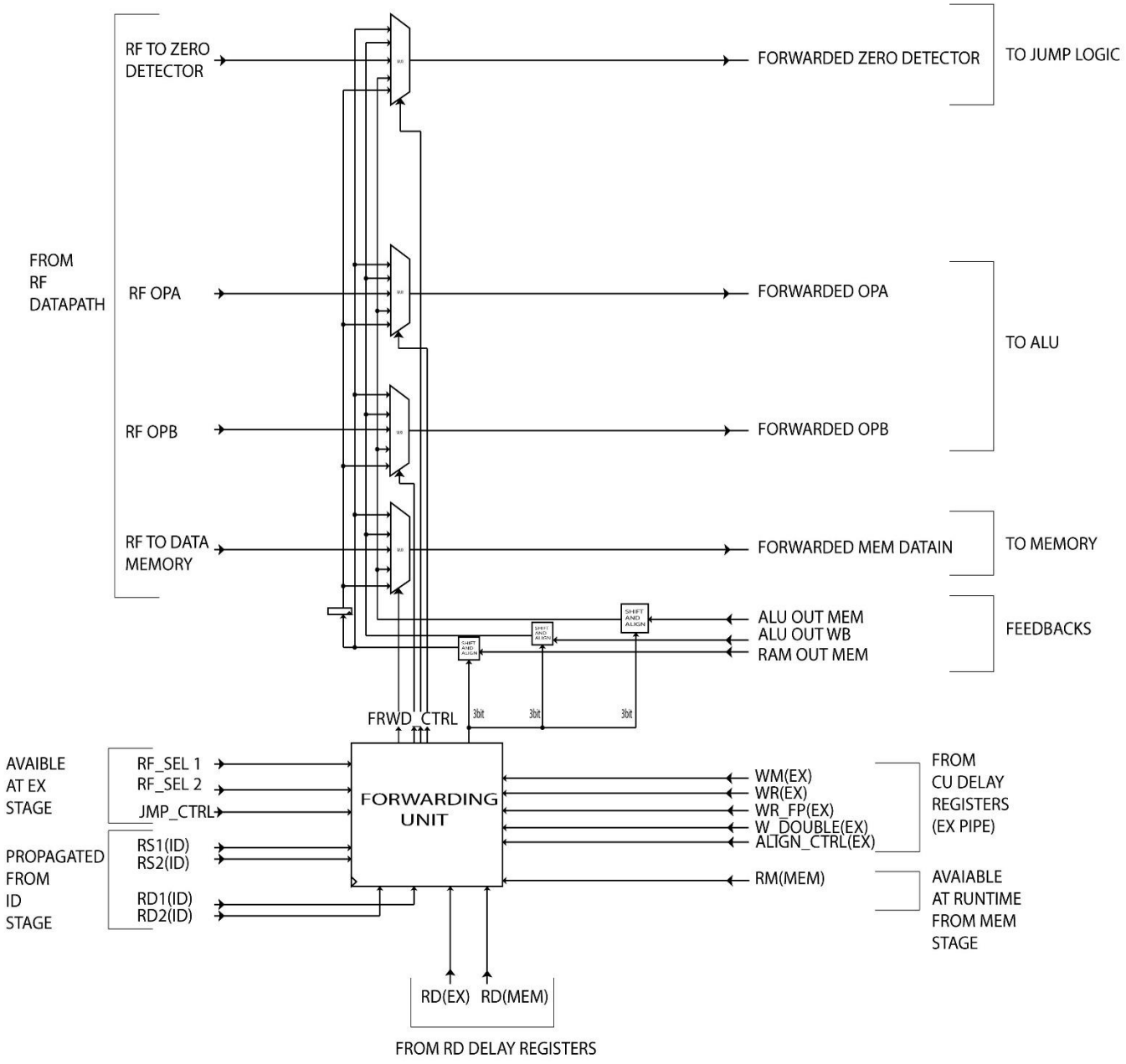


Figure 27 - Data forwarding

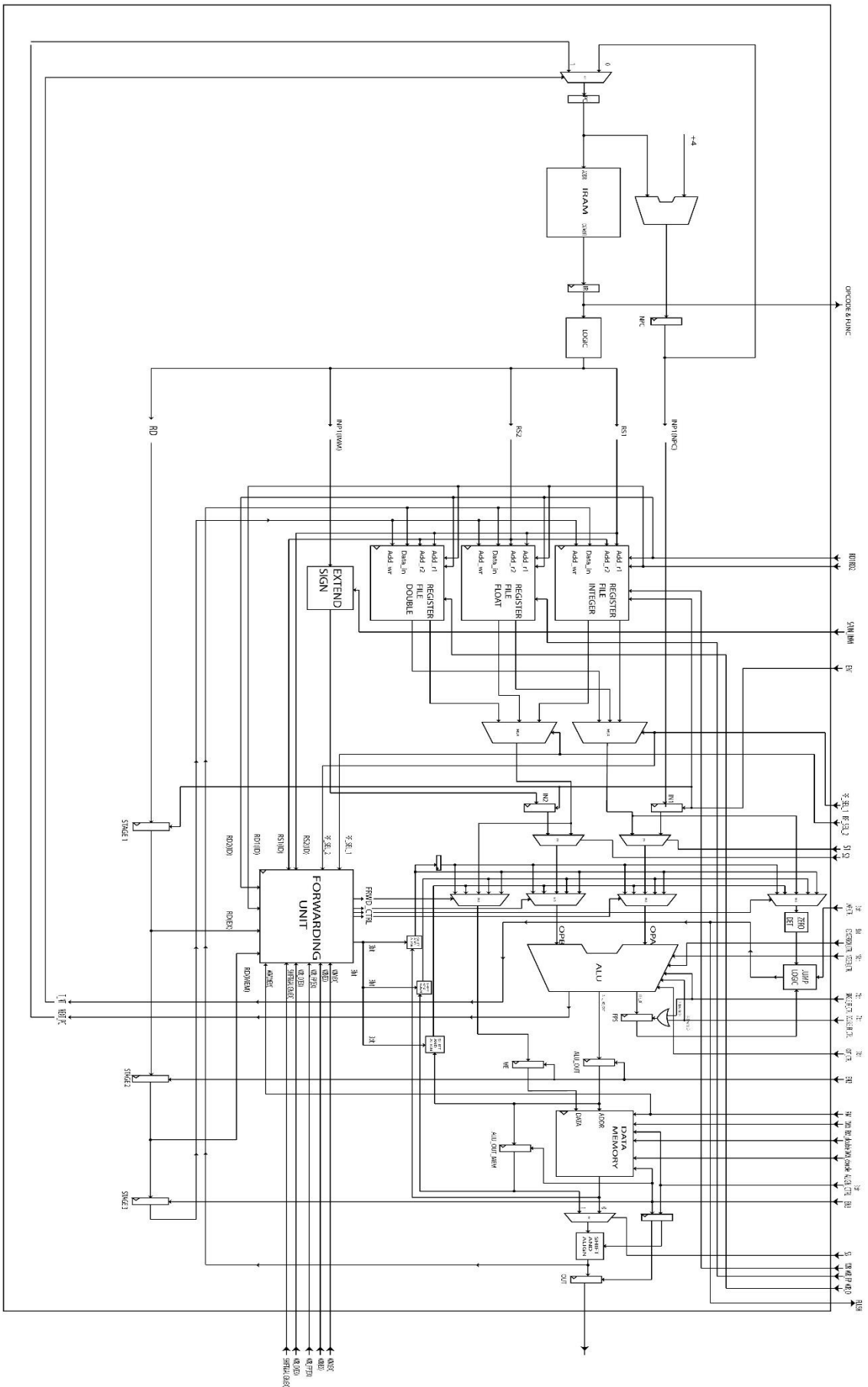


Figure 28 - Datapath

IMPLEMENTATION

Once the DLX netlist has been written, it was tested using many assembly programs. Then the project has been synthetized using Synopsys Design Vision. In this phase an analysis on timing, area and power dissipation was performed. The post synthesis netlist has been extracted. First, a post synthesis simulation was done to check the correct behaviour of the DLX after the synthesis. Using the verified netlist, the physical design phase started. So, the DLX was placed and routed, post routing netlist has been extracted to check again the correct behaviour after the physical design phase.

All the steps will be described in a more detailed manner in the following sections.

Netlist simulation

The goal of this phase was to find and fix the error in the DLX netlist. Some assembly programs were written in order to stimulate as much component as possible. An example of ASM test is leaved here.

```
1   #2.5 = 0x40200000.
2   lhi r1, #0x4020
3
4   #Save 1.
5   addui r2, r0 #1
6
7   #Store.
8   sw 4(r0),r1
9
10  #Load in FP single RF.
11  lf r1, 4(r0)
12
13  #Move operand from integer RF to float single RF.
14  movi2fp r1, r1
15
16  #Convert 1 integer to float single.
17  cvti2f r2, r2
18
19  #Perform floating division.
20  divf r3,r1,r2
21
22  #Loop using FPS.
23  nef r2, r1
24  label:
25  bfpt label
```

Figure 29 - ASM test

The resulting waveform is leaved.

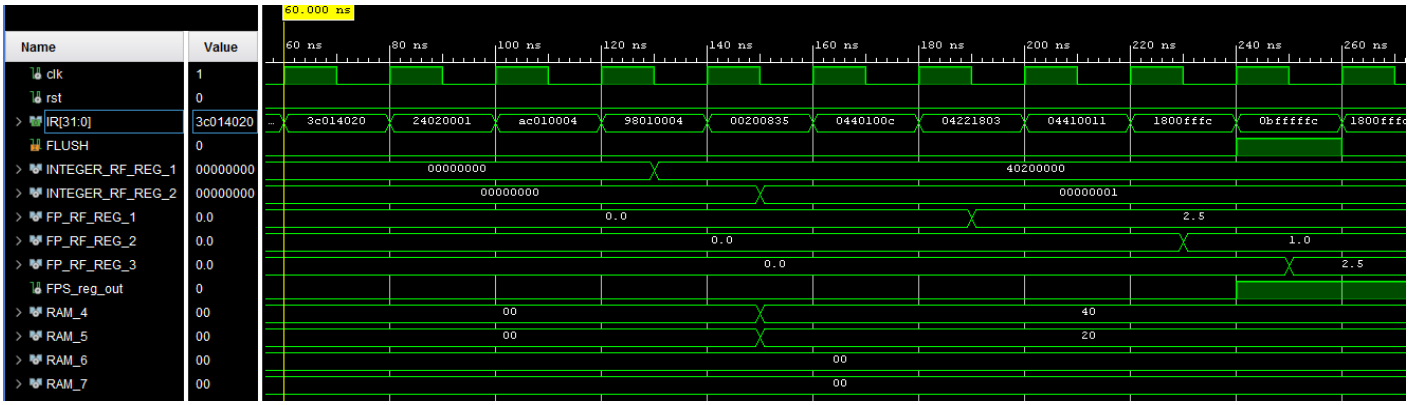


Figure 30 - Waveform

From the figure is possible to see the IR signal, which shows the content of the Instruction register. This signal is updated after the fetch stage.

The test performs the division between floating point 2.5 and 1.0. As is possible to see from the figure the writing of operand in register files and memory cells are performed in the falling edge. Also is possible to notice that no NOP operation are not necessary in the ASM code because the Forwarding unit solves all the hazards between variables. The final loop is performed by the FPS branch, which is settled by the *nef* operation. The following instruction 0x0bfffffc is then flushed by the control unit, and the next program counter is updated to create a loop.

Synthesis

Once the DLX netlist has been written and test, the synthesis phase started. Using Synopsys Design Vision, the netlist was compiled and synthetized. First synthesis was performed without any constraint. In order to understand the unconstrained timing and area the reports were written. In particular the area report, and the timing report. This last was used to understand the delay of the critical path considered on the path from RAM out to ALU out (feedback). This path was considered as critical due to the fact that the ram out is present on the falling edge and the ALU out register reads on the rising edge. So only half of the clock is useful to cover the data propagation.

Then a second synthesis was performed using a constraint on this particular path. So, again all reports were written. The value of the constraint path was fixed to 65 ns.

An improvement from power point of view was seen comparing the unconstrained DLX to constrained one. But the total amount of area was increased. In particular the power was reduced by 59,25% and the area increased by 8,4 %. Due to these improvements the constraint was considered acceptable.

A post synthesis simulation has been extracted to check the correct behaviour of the synthetized design.

The area report was also used to take the decision to delete the divider unit of the double ALU. In particular, a decrement of total area (around 30 %) was noticed without this unit.

Physical design

In this phase, the DLX has been placed and routed using Innovus tool.

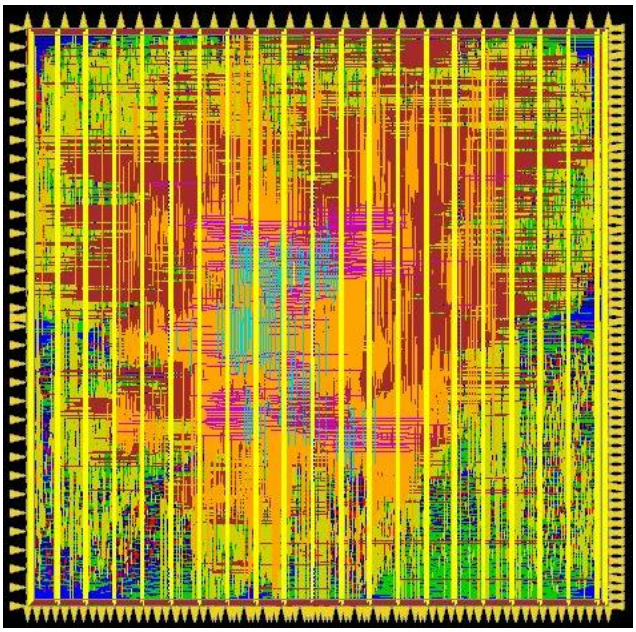


Figure 31 - Routed DLX

The post Clock-Tree-Synthesis was skipped because the post route simulation did not work correctly.

After the placing, the following operations were performed:

- Place filler.
- Routing.
- Post-routing optimization.
- Timing extraction.
- RC extraction.
- Verification of connectivity and geometry.
- Post route netlist.

After physical design, the effective behaviour of the routed DLX was checked.

DISCUSSION AND CONCLUSIONS

From the post-physical timing reports it's possible to check that the value of the constrained critical path was respected. Furthermore, the critical path presents a slack value of 20,976 ns, so the effective timing is 44,024 ns. Because this path has only half clock cycle the effective value is 88,048 ns. Knowing this, is possible to compute the maximum working frequency that is 11,35 Mhz. As is possible to see, the choice to have a non-pipelined ALU negatively affects the maximum clock frequency. On the other hand, all the operations in EX stage are executed exactly in one clock cycle. Therefore, at each clock cycle the DLX commits an instruction. A possible future improvement could be to pipeline some part of the ALU in order to rise the frequency value, keeping in mind that stalls could arise.

The developed DLX has been tested at each stage, using some ASM programs. The behaviour of the device works as expected at each test. In particular the simulation frequency has been raised to the critical one.

A. DLX VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity DLX is
    port (clk:     IN std_logic;
          rst:     IN std_logic;
          DLX_OUT: OUT std_logic_vector(63 downto 0);
          --IRAM SIGNALS
          Dout_IRAM : IN   std_logic_vector(31 downto 0);
          Addr_IRAM : OUT  std_logic_vector(31 downto 0);
          --RAM SIGNALS
          RM:               OUT std_logic;
          WM:               OUT std_logic;
          RM_DOUBLE:        OUT std_logic;
          WM_DOUBLE:        OUT std_logic;
          ALIGN_CTRL:       OUT std_logic_vector(2 downto 0);
          EN3:              OUT std_logic;
          ADDR_RAM:        OUT std_logic_vector(31 downto 0);
          DATAIN_RAM:      OUT std_logic_vector(63 downto 0);
          DATAOUT_RAM:      IN  std_logic_vector(63 downto 0));
end DLX;

architecture Structural of DLX is
component DataPath is
    port (CLK:          IN std_logic;
          RST:          IN std_logic;
          CLK_EN_RF:    IN std_logic_vector(2 downto 0);
          WR_EX_STAGE:  IN std_logic_vector(2 downto 0);
          WM_EX:        IN std_logic;
          ALIGN_CTRL_EX:IN std_logic_vector(2 downto 0);
          --IR INPUTS.
          INP1:         IN std_logic_vector(31 downto 0);
          RS1:          IN std_logic_vector(4 downto 0);
          RS2:          IN std_logic_vector(4 downto 0);
          INP2:         IN std_logic_vector(31 downto 0);
          RD:           IN std_logic_vector(4 downto 0);
          --STAGE 1 CONTROL SIGNALS.
          RD1:          IN std_logic;
          RD2:          IN std_logic;
          S_U_EXT:      IN std_logic_vector(1 downto 0);
          EN1:          IN std_logic;
          --STAGE 2 CONTROL SIGNALS.
          RF_SEL_1:     IN std_logic_vector(1 downto 0);
          RF_SEL_2:     IN std_logic_vector(1 downto 0);
          S1:           IN std_logic;
```

```
          S2:           IN std_logic;
          JMP_CTRL:     IN std_logic_vector(2 downto 0);
          CONV_CTRL:        IN std_logic_vector(4 downto 0);
          INT_CTRL:         IN std_logic_vector(14 downto 0);
          SINGLE_FP_CTRL:   IN std_logic_vector(6 downto 0);
          DOUBLE_FP_CTRL:   IN std_logic_vector(6 downto 0);
          OUT_CTRL:         IN std_logic_vector(2 downto 0);
          EN2:              IN std_logic;
          --STAGE 3 CONTROL SIGNALS.
          RM:               IN std_logic;
          WM:               IN std_logic;
          RM_DOUBLE:        IN std_logic;
          WM_DOUBLE:        IN std_logic;
          ALIGN_CTRL:       IN std_logic_vector(2 downto 0);
          EN3:              IN std_logic;
          --STAGE 4 CONTROL SIGNALS.
          S3:               IN std_logic;
          WR_INT:           IN std_logic;
          WR_FLOAT:         IN std_logic;
          WR_DOUBLE:        IN std_logic;
          --OUTPUTS.
          B_T_NT:           OUT std_logic;
          NPC:              OUT std_logic_vector(31 downto 0);
          DLX_OUT:          OUT std_logic_vector(63 downto 0);
          --RAM SIGNALS
          ADDR_RAM:      OUT std_logic_vector(31 downto 0);
          DATAIN_RAM:    OUT std_logic_vector(63 downto 0);
          DATAOUT_RAM:    IN  std_logic_vector(63 downto 0));
end component;

component DLX_Fetch_Stage is
    port (clk:      IN std_logic;
          rst:      IN std_logic;
          MUX_sel:  IN std_logic;
          NPC_in:   IN std_logic_vector(31 downto 0);
          NPC_out:  OUT std_logic_vector(31 downto 0);
          IR:       OUT std_logic_vector(31 downto 0);
          --IRAM SIGNALS
          Dout_IRAM : IN   std_logic_vector(31 downto 0);
          Addr_IRAM : OUT  std_logic_vector(31 downto 0));
end component;

component CONTROL_UNIT is
    port(CLK:           IN std_logic;
         RST:           IN std_logic;
         OPCODE:        IN std_logic_vector(5 downto 0);
         FUNC:          IN std_logic_vector(10 downto 0);
         FLUSH:         IN std_logic;
         CTRL_WORD:     OUT std_logic_vector(63 downto 0);
         CLK_EN_RF:     OUT std_logic_vector(2 downto 0);
         WR_EX:         OUT std_logic_vector(2 downto 0);
         WM_EX:         OUT std_logic;
         ALIGN_CTRL_EX: OUT std_logic_vector(2 downto 0));
end component;
```

```
signal PC_DP, PC_rst, NPC_s: std_logic_vector(31 downto 0);
signal IR: std_logic_vector(31 downto 0);
signal RS1, RS2, RD: std_logic_vector(4 downto 0);
signal INP2: std_logic_vector(31 downto 0);
signal ext_IMM, ext_offset: std_logic_vector(31 downto 0);
signal OPCODE: std_logic_vector(5 downto 0);
signal FUNC: std_logic_vector(10 downto 0);
signal B_T_NT: std_logic;
type INSTRUCTION_TYPE is (I_TYPE, R_TYPE, J_TYPE);
signal TYPE_CASE: INSTRUCTION_TYPE;
signal CTRL_WORD: std_logic_vector(63 downto 0);
signal CLK_EN_RF: std_logic_vector(2 downto 0);
signal WR_EX_STAGE, ALIGN_CTRL_EX: std_logic_vector(2 downto 0);
signal RM_EX_STAGE, WM_EX: std_logic;
begin
CU:     CONTROL_UNIT port map (CLK       => CLK,
                               RST       => RST,
                               OPCODE    => OPCODE,
                               FUNC      => FUNC,
                               FLUSH     =>  B_T_NT,
                               CTRL_WORD => CTRL_WORD,
                               CLK_EN_RF => CLK_EN_RF,
                               WR_EX     => WR_EX_STAGE,
                               WM_EX     => WM_EX,
                               ALIGN_CTRL_EX => ALIGN_CTRL_EX);

IF_stage: DLX_Fetch_Stage port map (CLK, RST, B_T_NT, PC_DP, NPC_s, IR,
Dout_IRAM, Addr_IRAM);
ID_EX_MEM_WB_stages: DataPath port map (CLK  => CLK,
                                        RST  => RST,
                                        CLK_EN_RF   => CLK_EN_RF,
                                        WR_EX_STAGE => WR_EX_STAGE,
                                        WM_EX       => WM_EX,
                                        ALIGN_CTRL_EX => ALIGN_CTRL_EX,
                                        --IR INPUTS.
                                        INP1 => NPC_s,
                                        RS1  => RS1,
                                        RS2  => RS2,
                                        INP2 => INP2,
                                        RD   => Rd,
                                        --STAGE 1 CONTROL SIGNALS.
                                        RD1         => CTRL_WORD(63),
                                        RD2         => CTRL_WORD(62),
                                        S_U_EXT     => CTRL_WORD(61
downto 60),
                                        EN1         => CTRL_WORD(59),
                                        --STAGE 2 CONTROL SIGNALS.
                                        RF_SEL_1        => CTRL_WORD(58
downto 57),
                                        RF_SEL_2        => CTRL_WORD(56
downto 55),
                                        S1              => CTRL_WORD(54),
                                        S2              => CTRL_WORD(53),
```

```
                                        JMP_CTRL        => CTRL_WORD(52
downto 50),
                                        CONV_CTRL       => CTRL_WORD(49
downto 45),
                                        INT_CTRL        => CTRL_WORD(44
downto 30),
                                        SINGLE_FP_CTRL  => CTRL_WORD(29
downto 23),
                                        DOUBLE_FP_CTRL  => CTRL_WORD(22
downto 16),
                                        OUT_CTRL        => CTRL_WORD(15
downto 13),
                                        EN2             => CTRL_WORD(12),
                                        --STAGE 3 CONTROL SIGNALS.
                                        RM              => CTRL_WORD(11),
                                        WM              => CTRL_WORD(10),
                                        RM_DOUBLE       => CTRL_WORD(9),
                                        WM_DOUBLE       => CTRL_WORD(8),
                                        ALIGN_CTRL      => CTRL_WORD(7
downto 5),
                                        EN3             => CTRL_WORD(4),
                                        --STAGE 4 CONTROL SIGNALS.
                                        S3              => CTRL_WORD(3),
                                        WR_INT          => CTRL_WORD(2),
                                        WR_FLOAT        => CTRL_WORD(1),
                                        WR_DOUBLE       => CTRL_WORD(0),
                                        --OUTPUTS.
                                        B_T_NT          => B_T_NT,
                                        DLX_OUT         => DLX_OUT,
                                        NPC             => PC_DP,
                                        --RAM SIGNALS.
                                        ADDR_RAM    => ADDR_RAM,
                                        DATAIN_RAM  => DATAIN_RAM,
                                        DATAOUT_RAM => DATAOUT_RAM);

RM              <= CTRL_WORD(11);
WM              <= CTRL_WORD(10);
RM_DOUBLE       <= CTRL_WORD(9);
WM_DOUBLE       <= CTRL_WORD(8);
ALIGN_CTRL      <= CTRL_WORD(7 downto 5);
EN3             <= CTRL_WORD(4);

OPCODE <= IR (31 downto 26);
FUNC   <= IR (10 downto 0);

TYPE_CASE <= I_TYPE when (OPCODE(5) or OPCODE(4) or OPCODE(3) or
OPCODE(2)) = '1'                      else --I-TYPE.
             R_TYPE when (OPCODE(5) or OPCODE(4) or OPCODE(3) or
OPCODE(2) or OPCODE(1)) = '0'         else --R-TYPE.
             J_TYPE ;
--J-TYPE.


RS1 <= IR(25 downto 21);
```

```
RS2 <= IR(20 downto 16);

RD <= "11111"          when  (TYPE_CASE = J_TYPE) or (OPCODE = "010011")
else --JUMP AND LINK CASE.
      IR(20 downto 16) when  TYPE_CASE = I_TYPE else
      IR(15 downto 11);

INP2 <= ext_IMM     when TYPE_CASE = I_TYPE else
        ext_offset;

ext_IMM (15 downto 0) <= IR (15 downto 0);
ext_IMM (31 downto 16)<= (others => '0');

ext_offset(25 downto 0) <= IR(25 downto 0);
ext_offset(31 downto 26)<= (others => '0');


end Structural;
```

B. CONTROL UNIT VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use WORK.myTypes.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity CONTROL_UNIT is
    port(CLK:           IN std_logic;
         RST:           IN std_logic;
         OPCODE:        IN std_logic_vector(5 downto 0);
         FUNC:          IN std_logic_vector(10 downto 0);
         FLUSH:         IN std_logic;
         CTRL_WORD:     OUT std_logic_vector(63 downto 0);
         CLK_EN_RF:     OUT std_logic_vector(2 downto 0);
         WR_EX:         OUT std_logic_vector(2 downto 0);
         WM_EX:         OUT std_logic;
         ALIGN_CTRL_EX: OUT std_logic_vector(2 downto 0));
end CONTROL_UNIT;

architecture Behavioral of CONTROL_UNIT is

signal pipe_1_out: std_logic_vector(58 downto 0);
signal pipe_2_out: std_logic_vector(11 downto 0);

signal stage_1_ctrl_word: std_logic_vector(4 downto 0);
signal stage_2_ctrl_word: std_logic_vector(46 downto 0);
signal stage_3_ctrl_word: std_logic_vector(7 downto 0);
signal stage_4_ctrl_word: std_logic_vector(3 downto 0);

signal CONTROL_WORD, FLUSH_CONTROL_WORD: std_logic_vector(63 downto 0); -
-TMP_CTRL_WORD.

constant RD1        : integer:= 63;
constant RF_SEL_1_B0: integer:= 58;
constant RF_SEL_1_B1: integer:= 57;
constant RM         : integer:= 11;
constant WB_INT     : integer:= 2;
constant WB_FP      : integer:= 1;
constant WB_DOUBLE  : integer:= 0;

component Register_Generic_rst is
    generic (N: integer:= 64);
    port (data_in: in std_logic_vector(N-1 downto 0);
          clk:     in std_logic;
          rst:     in std_logic;
          en:      in std_logic;
          data_out: out std_logic_vector(N-1 downto 0));
end component;
```

```
--component register_generic_async_wr is
--    generic (N: integer:= 64);
--    port (data_in: in std_logic_vector(N-1 downto 0);
--          clk:     in std_logic;
--          rst:     in std_logic;
--          en:      in std_logic;
--          data_out: out std_logic_vector(N-1 downto 0));
--end component;
begin
    process(OPCODE, FUNC)
    begin
        case OPCODE is
            when JTYPE_J =>
                CONTROL_WORD <=
"00111000001101000001000000000001000000000000000000001000000010000";
            when JTYPE_JAL =>
                CONTROL_WORD <=
"00111000001101000001000000000001000000000000000001011000000011100";
            when ITYPE_BEQZ =>
                CONTROL_WORD <=
"10011000001011000001000000000001000000000000000000001000000010000";
            when ITYPE_BNEZ =>
                CONTROL_WORD <=
"10011000001100000001000000000001000000000000000000001000000010000";
            when ITYPE_BFPT =>
                CONTROL_WORD <=
"00011000001010000001000000000001000000000000000000001000000010000";
            when ITYPE_BFPF =>
                CONTROL_WORD <=
"00011000001001000001000000000001000000000000000000001000000010000";
            when ITYPE_ADDI =>
                CONTROL_WORD <=
"10011000011000000001000000000001000000000000000000001000000011100";
            when ITYPE_ADDUI =>
                CONTROL_WORD <=
"10001000011000000001000000000001000000000000000000001000000011100";
            when ITYPE_SUBI =>
                CONTROL_WORD <=
"10011000011000000001100000000001000000000000000000001000000011100";
            when ITYPE_SUBUI =>
                CONTROL_WORD <=
"10001000011000000001100000000001000000000000000000001000000011100";
            when ITYPE_ANDI =>
                CONTROL_WORD <=
"10001000011000000001000000000001100000000000000000001000000011100";
            when ITYPE_ORI =>
                CONTROL_WORD <=
"10001000011000000001000010000001100000000000000000001000000011100";
            when ITYPE_XORI =>
                CONTROL_WORD <=
"10001000011000000001000100000001100000000000000000001000000011100";
            when ITYPE_LHI =>
                CONTROL_WORD <=
"00011000011000000000000000000000000000000000000001111000001111100";
```

```
            when ITYPE_JR =>
                CONTROL_WORD <=
"11001000011101000001000000000001000000000000000000000000000000000";
            when ITYPE_JALR =>
                CONTROL_WORD <=
"11001000001101000001000000000000000000000000000001011000000011100";
            when ITYPE_SLLI =>
                CONTROL_WORD <=
"10001000011000000001000000000001000000000000000000001000000011100";
            when ITYPE_NOP =>
                CONTROL_WORD <=
"00000000000000000000000000000000000000000000000000000000000000000";
            when ITYPE_SRLI =>
                CONTROL_WORD <=
"10001000011000000001000000010000100000000000000000001000000011100";
            when ITYPE_SRAI =>
                CONTROL_WORD <=
"10001000011000000001000000010000010000000000000000001000000011100";
            when ITYPE_SEQI =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000001101000000011100";
            when ITYPE_SNEI =>
                CONTROL_WORD <=
"10011000011000000001000000000100000000000000000001101000000011100";
            when ITYPE_SLTI =>
                CONTROL_WORD <=
"10011000011000000001000000001010000000000000000001101000000011100";
            when ITYPE_SGTI =>
                CONTROL_WORD <=
"10011000011000000001000000001100000000000000000001101000000011100";
            when ITYPE_SLEI =>
                CONTROL_WORD <=
"10011000011000000001000000001100000000000000000001101000000011100";
            when ITYPE_SGEI =>
                CONTROL_WORD <=
"10011000011000000001000000001000000000000000000001101000000011100";
            when ITYPE_LB =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000000011000100010100";
            when ITYPE_LH =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000000011000001101000";
            when ITYPE_LW =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000000011000000101000";
            when ITYPE_LBU =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000000011000101101000";
            when ITYPE_LHU =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000000011000010101000";
            when ITYPE_LF =>
                CONTROL_WORD <=
"10011000011000000001000000000010000000000000000000011000000100100";
```

```
            when ITYPE_LD =>
                CONTROL_WORD <=
"10011000011000000001000000000001000000000000000000001101000010001";
            when ITYPE_SB =>
                CONTROL_WORD <=
"11011000011000000001000000000001000000000000000000001010011010000";
            when ITYPE_SH =>
                CONTROL_WORD <=
"11011000011000000001000000000001000000000000000000001010011110000";
            when ITYPE_SW =>
                CONTROL_WORD <=
"11011000011000000001000000000001000000000000000000001010000010000";
            when ITYPE_SF =>
                CONTROL_WORD <=
"11011000111000000001000000000001000000000000000000001010000010000";
            when ITYPE_SD =>
                CONTROL_WORD <=
"11011001011000000001000000000001000000000000000000001010100010000";
            when ITYPE_SLTUI =>
                CONTROL_WORD <=
"10001000011000000001000000001010000000000000000001101000000011100";
            when ITYPE_SGTUI =>
                CONTROL_WORD <=
"10001000011000000001000000001100000000000000000001101000000011100";
            when ITYPE_SLEUI =>
                CONTROL_WORD <=
"10001000011000000001000000001100000000000000000001101000000011100";
            when ITYPE_SGEUI =>
                CONTROL_WORD <=
"10001000011000000001000000001000000000000000000001101000000011100";
            when ITYPE_RSLI =>
                CONTROL_WORD <=
"10001000011000000001000000110000100000000000000000001000000011100";
            when ITYPE_RSRI =>
                CONTROL_WORD <=
"10001000011000000001000001000000100000000000000000001000000011100";
            when ITYPE_NANDI =>
                CONTROL_WORD <=
"10001000011000000001001000000000110000000000000000001000000011100";
            when ITYPE_NORI =>
                CONTROL_WORD <=
"10001000011000000001000110000000110000000000000000001000000011100";
            when ITYPE_XNORI =>
                CONTROL_WORD <=
"10001000011000000001001010000000110000000000000000001000000011100";

            when R_INTEGER_TYPE =>
                case FUNC is
                    when RTYPE_INT_SLL =>
                        CONTROL_WORD <=
"11001000010000000001000000000001000000000000000000001000000011100";
                    when RTYPE_INT_SRL =>
                        CONTROL_WORD <=
"11001000010000000001000000100001000000000000000000001000000011100";
```

```
                    when RTYPE_INT_SRA =>
                        CONTROL_WORD <=
"11001000010000000001000000100000100000000000000000001000000011100";
                    when RTYPE_INT_ADD =>
                        CONTROL_WORD <=
"11001000010000000001000000000001000000000000000000001000000011100";
                    when RTYPE_INT_ADDU =>
                        CONTROL_WORD <=
"11001000010000000001000000000001000000000000000000001000000011100";
                    when RTYPE_INT_SUB =>
                        CONTROL_WORD <=
"11001000010000000001100000000001000000000000000000001000000011100";
                    when RTYPE_INT_SUBU =>
                        CONTROL_WORD <=
"11001000010000000001100000000001000000000000000000001000000011100";
                    when RTYPE_INT_AND =>
                        CONTROL_WORD <=
"11001000010000000001000000000000110000000000000000001000000011100";
                    when RTYPE_INT_OR =>
                        CONTROL_WORD <=
"11001000010000000001000010000000110000000000000000001000000011100";
                    when RTYPE_INT_XOR =>
                        CONTROL_WORD <=
"11001000010000000001000100000000110000000000000000001000000011100";
                    when RTYPE_INT_SEQ =>
                        CONTROL_WORD <=
"11001000010000000001000000000001000000000000000000001101000000011100";
                    when RTYPE_INT_SNE =>
                        CONTROL_WORD <=
"11001000010000000001000000000010000000000000000000001101000000011100";
                    when RTYPE_INT_SLT =>
                        CONTROL_WORD <=
"11001000010000000001000000001010000000000000000000001101000000011100";
                    when RTYPE_INT_SGT =>
                        CONTROL_WORD <=
"11001000010000000001000000001100000000000000000000001101000000011100";
                    when RTYPE_INT_SLE =>
                        CONTROL_WORD <=
"11001000010000000001000000001100000000000000000000001101000000011100";
                    when RTYPE_INT_SGE =>
                        CONTROL_WORD <=
"11001000010000000001000000001000000000000000000000001101000000011100";
                    when RTYPE_INT_MOVF =>
                        CONTROL_WORD <=
"10001010010000000000000000000000000000000000000000001001000000011010";
                    when RTYPE_INT_MOVD =>
                        CONTROL_WORD <=
"10001100010000000000000000000000000000000000000000001001001000011001";
                    when RTYPE_INT_MOVFP2I =>
                        CONTROL_WORD <=
"10001010010000000000000000000000000000000000000000001001000000011100";
                    when RTYPE_INT_MOVI2FP =>
                        CONTROL_WORD <=
"10001000010000000000000000000000000000000000000000001001000000011010";
```

```
                    when RTYPE_INT_SLTU =>
                        CONTROL_WORD <=
"11001000010000000001000000001010000000000000000001101000000011100";
                    when RTYPE_INT_SGTU =>
                        CONTROL_WORD <=
"11001000010000000001000000001100000000000000000001101000000011100";
                    when RTYPE_INT_SLEU =>
                        CONTROL_WORD <=
"11001000010000000001000000001100000000000000000001101000000011100";
                    when RTYPE_INT_SGEU =>
                        CONTROL_WORD <=
"11001000010000000001000000001000000000000000000001101000000011100";
                    when RTYPE_INT_RSL =>
                        CONTROL_WORD
<="11001000010000000001000000110000100000000000000000001000000011100";
                    when RTYPE_INT_RSR =>
                        CONTROL_WORD <=
"11001000010000000001000001000000100000000000000000001000000011100";
                    when RTYPE_INT_NAND =>
                        CONTROL_WORD <=
"11001000010000000001001000000000110000000000000000001000000011100";
                    when RTYPE_INT_NOR =>
                        CONTROL_WORD <=
"11001000010000000001000110000000110000000000000000001000000011100";
                    when RTYPE_INT_XNOR =>
                        CONTROL_WORD <=
"11001000010000000001001010000000110000000000000000001000000011100";
                    when others =>
                        CONTROL_WORD <= (others => '0');
                end case;

                --FLOATING POINT
            when R_FLOAT_TYPE =>
                case FUNC is
                    when RTYPE_FLOAT_ADDF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000001000011000000001010000000011010";
                    when RTYPE_FLOAT_SUBF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000001100011000000001010000000011010";
                    when RTYPE_FLOAT_MULTF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000001000001000000001010000000011010";
                    when RTYPE_FLOAT_DIVF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000001000010000000001010000000011010";
                    when RTYPE_FLOAT_ADDD =>
                        CONTROL_WORD <=
"11001101010000000000000000000000000000001000001011100000000011001";
                    when RTYPE_FLOAT_SUBD =>
                        CONTROL_WORD <=
"11001101010000000000000000000000000000001100001011100000000011001";
                    when RTYPE_FLOAT_MULTD =>
```

```
                        CONTROL_WORD <=
"11001101010000000000000000000000000000000010000100111000000011001";
                    when RTYPE_FLOAT_DIVD =>
                        CONTROL_WORD <=
"11001101010000000000000000000000000000000010000110111000000011001";
                    when RTYPE_FLOAT_CVTF2D =>
                        CONTROL_WORD <=
"10001010010000001010000000000000000000000000000001001000000011001";
                    when RTYPE_FLOAT_CVTF2I =>
                        CONTROL_WORD <=
"10001010010000001011000000000000000000000000000001001000000011100";
                    when RTYPE_FLOAT_CVTD2F =>
                        CONTROL_WORD <=
"10001100010000001100000000000000000000000000000001001000000011010";
                    when RTYPE_FLOAT_CVTD2I =>
                        CONTROL_WORD <=
"10001100010000011000000000000000000000000000000001001000000011100";
                    when RTYPE_FLOAT_CVTI2F =>
                        CONTROL_WORD <=
"10001000010000010010000000000000000000000000000001001000000011010";
                    when RTYPE_FLOAT_CVTI2D =>
                        CONTROL_WORD <=
"10001000010000010100000000000000000000000000000001001000000011001";
                    when RTYPE_FLOAT_MULT =>
                        CONTROL_WORD <=
"11001000010000000001010000000000010000000000000000001000000011100";
                    when RTYPE_FLOAT_DIV =>
                        CONTROL_WORD <=
"11001000010000110010000000000000010000100000000000110000000011100";
                    when RTYPE_FLOAT_EQF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000011001000000000001000000000000";
                    when RTYPE_FLOAT_NEF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000011010000000000001000000000000";
                    when RTYPE_FLOAT_LTF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000011101000000000001000000000000";
                    when RTYPE_FLOAT_GTF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000011011000000000001000000000000";
                    when RTYPE_FLOAT_LEF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000011110000000000001000000000000";
                    when RTYPE_FLOAT_GEF =>
                        CONTROL_WORD <=
"11001010110000000000000000000000011100000000000001000000000000";
                    when RTYPE_FLOAT_MULTU =>
                        CONTROL_WORD <=
"11001000010000000001000000000000010000000000000000001000000011100";
                    when RTYPE_FLOAT_DIVU =>
                        CONTROL_WORD <=
"11001000010000010010000000000000010000100000000000110000000011100";
```

```
                    when others =>
                        CONTROL_WORD <= (others => '0');
                end case;

            when others =>
                CONTROL_WORD <= (others => '0');
        end case;
    end process;

FLUSH_CONTROL_WORD <= CONTROL_WORD when FLUSH = '0' else
                      (others => '0');



PIPE_1: Register_Generic_rst generic map(59)
                             port map(FLUSH_CONTROL_WORD(58 downto 0),
clk, rst, '1', pipe_1_out);
PIPE_2: Register_Generic_rst generic map(12)
                             port map(pipe_1_out(11 downto 0), clk, rst,
'1', pipe_2_out);
PIPE_3: Register_Generic_rst generic map(4)
                             port map(pipe_2_out(3 downto 0), clk, rst,
'1', stage_4_ctrl_word);

stage_1_ctrl_word <= FLUSH_CONTROL_WORD(63 downto 59);     --ID STAGE.
stage_2_ctrl_word <= pipe_1_out(58 downto 12);             --EX STAGE.
stage_3_ctrl_word <= pipe_2_out(11 downto 4);              --MEM STAGE.

CTRL_WORD <=
stage_1_ctrl_word&stage_2_ctrl_word&stage_3_ctrl_word&stage_4_ctrl_word;

--OUT WR SIGNALS.
ALIGN_CTRL_EX    <= pipe_1_out(7 downto 5);
WM_EX            <= pipe_1_out(10);
WR_EX(WB_INT)    <= pipe_1_out(WB_INT);
WR_EX(WB_FP)     <= pipe_1_out(WB_FP);
WR_EX(WB_DOUBLE) <= pipe_1_out(WB_DOUBLE);
end Behavioral;
```

C. FETCH STAGE VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity DLX_Fetch_Stage is
    port (clk:      IN std_logic;
          rst:      IN std_logic;
          MUX_sel:  IN std_logic;
          NPC_in:   IN std_logic_vector(31 downto 0);
          NPC_out:  OUT std_logic_vector(31 downto 0);
          IR:       OUT std_logic_vector(31 downto 0);
          --IRAM SIGNALS
          Dout_IRAM : IN   std_logic_vector(31 downto 0);
          Addr_IRAM : OUT  std_logic_vector(31 downto 0));
end DLX_Fetch_Stage;

architecture Structural of DLX_Fetch_Stage is
component Register_Generic_rst is
    generic (N: integer:= 64);
    port (data_in: in std_logic_vector(N-1 downto 0);
          clk:     in std_logic;
          rst:     in std_logic;
          en:      in std_logic;
          data_out: out std_logic_vector(N-1 downto 0));
end component;

signal PC: std_logic_vector(31 downto 0):= (others => '0');
signal NPC, NPC_out_reg: std_logic_vector(31 downto 0):= (others => '0');
signal NIR: std_logic_vector(31 downto 0):= (others => '0');
signal rst_sync : std_logic;
signal sel: std_logic_vector(1 downto 0);
begin
--CONNECT IRAM.
Addr_IRAM <= PC;
NIR       <= Dout_IRAM;

sel <= MUX_sel & rst_sync;

PC <= (others => '0')   when sel="01" else
      (others => '0')   when sel="11" else
      NPC_in            when sel="10" else
      NPC_out_reg       when sel="00" ;

NEXT_PC: Register_Generic_rst generic map (32)
                    port map (NPC, clk, rst, '1', NPC_out_reg);--WHEN
FLUSH RESET THE NPC.
```

```
IR_REG: Register_Generic_rst generic map (32)
                    port map (NIR, clk, rst,'1', IR);

NPC <= std_logic_vector(unsigned(PC) + 4);

NPC_out <= NPC_out_reg;

    process(clk)
    begin
        if (clk='1'and clk'event) then
            if (rst = '1') then
                rst_sync <= '1';
            else
                rst_sync <= '0';
            end if;
        end if;
    end process;
end Structural;
```

D. DATAPATH VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity DataPath is
    port (CLK:          IN std_logic;
          RST:          IN std_logic;
          CLK_EN_RF:    IN std_logic_vector(2 downto 0);
          WR_EX_STAGE:  IN std_logic_vector(2 downto 0);
          WM_EX:        IN std_logic;
          ALIGN_CTRL_EX:IN std_logic_vector(2 downto 0);
          --IR INPUTS.
          INP1:         IN std_logic_vector(31 downto 0);
          RS1:          IN std_logic_vector(4 downto 0);
          RS2:          IN std_logic_vector(4 downto 0);
          INP2:         IN std_logic_vector(31 downto 0);
          RD:           IN std_logic_vector(4 downto 0);
          --STAGE 1 CONTROL SIGNALS.
          RD1:          IN std_logic;
          RD2:          IN std_logic;
          S_U_EXT:      IN std_logic_vector(1 downto 0);
          EN1:          IN std_logic;
          --STAGE 2 CONTROL SIGNALS.
          RF_SEL_1:     IN std_logic_vector(1 downto 0);
          RF_SEL_2:     IN std_logic_vector(1 downto 0);
          S1:           IN std_logic;
          S2:           IN std_logic;
          JMP_CTRL:     IN std_logic_vector(2 downto 0);
          CONV_CTRL:        IN std_logic_vector(4 downto 0);
          INT_CTRL:         IN std_logic_vector(14 downto 0);
          SINGLE_FP_CTRL:   IN std_logic_vector(6 downto 0);
          DOUBLE_FP_CTRL:   IN std_logic_vector(6 downto 0);
          OUT_CTRL:         IN std_logic_vector(2 downto 0);
          EN2:              IN std_logic;
          --STAGE 3 CONTROL SIGNALS.
          RM:               IN std_logic;
          WM:               IN std_logic;
          RM_DOUBLE:        IN std_logic;
          WM_DOUBLE:        IN std_logic;
          ALIGN_CTRL:       IN std_logic_vector(2 downto 0);
          EN3:              IN std_logic;
          --STAGE 4 CONTROL SIGNALS.
          S3:               IN std_logic;
          WR_INT:           IN std_logic;
          WR_FLOAT:         IN std_logic;
          WR_DOUBLE:        IN std_logic;
```

```
          --OUTPUTS.
          B_T_NT:           OUT std_logic;
          NPC:              OUT std_logic_vector(31 downto 0);
          DLX_OUT:          OUT std_logic_vector(63 downto 0);
          --RAM SIGNALS
          ADDR_RAM:      OUT std_logic_vector(31 downto 0);
          DATAIN_RAM:    OUT std_logic_vector(63 downto 0);
          DATAOUT_RAM:    IN  std_logic_vector(63 downto 0));
end DataPath;

architecture Structural of DataPath is

component ALU is
    Port (
        OPA, OPB     : in std_logic_vector(63 downto 0);  -- 2 inputs N-
bit
        integer_ctrl : in std_logic_vector(14 downto 0);
        single_precision_fp_ctrl : in std_logic_vector(6 downto 0);
        double_precision_fp_ctrl : in std_logic_vector(6 downto 0);
        conversion_ctrl : in std_logic_vector(4 downto 0);
        out_ctrl : in std_logic_vector(2 downto 0);
        ALU_Out   : out std_logic_vector(63 downto 0);
        PC_ALU: out std_logic_vector(63 downto 0);
        FPS : out std_logic);
end component;

component DLX_RF is
    port ( CLK:    IN std_logic;
           RST:       IN std_logic;
           ENABLE:       IN std_logic;
           RD1:    IN std_logic;
           RD2:    IN std_logic;
           RD_DOUBLE: IN std_logic;
           WR_DOUBLE: IN std_logic;
           WR:           IN std_logic;
           ADD_WR:       IN std_logic_vector(4 downto 0);
           ADD_RD1:   IN std_logic_vector(4 downto 0);
           ADD_RD2:   IN std_logic_vector(4 downto 0);
           DATAIN:       IN std_logic_vector(63 downto 0);
           OUT1:   OUT std_logic_vector(63 downto 0);
           OUT2:   OUT std_logic_vector(63 downto 0));
end component;

component Register_Generic is
    generic (N: integer:= 64);
    port (data_in: in std_logic_vector(N-1 downto 0);
          clk:     in std_logic;
          en:      in std_logic;
          data_out: out std_logic_vector(N-1 downto 0));
end component;

component Zero_detector is
    Generic (N: integer:= 32);
    Port (A: in std_logic_vector (N-1 downto 0);
```

```
          A_eq_zero: out std_logic);
end component;

component FFD is
    port (D: in std_logic;
          clk:     in std_logic;
          en_rd:      in std_logic;
          rst:     in std_logic;
          Q:       out std_logic);
end component;

component LOGIC_JMP is
    port (zero_det: in std_logic;
          FPS:      in std_logic;
          JMP_CTRL: in std_logic_vector(2 downto 0);
          B_T_NT:   OUT std_logic);
end component;

component EXT_IMM is
    port (S_U_EXT: in std_logic_vector(1 downto 0);
          IMM:     in std_logic_vector(31 downto 0);
          EXT_IMM: out std_logic_vector(63 downto 0));
end component;

component SHIFT_AND_ALIGN is
    port (INPUT:  IN std_logic_vector(63 downto 0);
          CTRL:   IN std_logic_vector(2 downto 0);
          OUTPUT: OUT std_logic_vector(63 downto 0));
end component;

component MASK_AND_ALIGN is
    port (INPUT:  IN std_logic_vector(63 downto 0);
          CTRL:   IN std_logic_vector(2 downto 0);
          OUTPUT: OUT std_logic_vector(63 downto 0));
end component;

component FORWARDING_UNIT is
    port (CLK:              IN std_logic;
          EN:               IN std_logic;
          RST:              IN std_logic;
          RF_SEL_1:         IN std_logic_vector(1 downto 0);
          RF_SEL_2:         IN std_logic_vector(1 downto 0);
          RS1_ID:           IN std_logic_vector(4 downto 0);
          RS2_ID:           IN std_logic_vector(4 downto 0);
          RD_MEM:           IN std_logic_vector(4 downto 0);
          RD_WB:            IN std_logic_vector(4 downto 0);
          RD1_ID:           IN std_logic;
          RD2_ID:           IN std_logic;
          JMP_CTRL:         IN std_logic_vector(2 downto 0);
          RM_MEM:           IN std_logic;
          WM_EX:            IN std_logic;
          WM_MEM:           IN std_logic;
          WR_INT_EX:        IN std_logic;
          WR_FP_EX:         IN std_logic;
```

```
          WR_DOUBLE_EX:     IN std_logic;
          ALIGN_CTRL_EX:    IN std_logic_vector(2 downto 0);
          WR_INT_WB:        IN std_logic;
          WR_FP_WB:         IN std_logic;
          WR_DOUBLE_WB:     IN std_logic;
          FEED_ALIGN_CTRL_EX: OUT std_logic_vector(2 downto 0);
          FEED_ALIGN_CTRL_MEM:OUT std_logic_vector(2 downto 0);
          FEED_ALIGN_CTRL_WB: OUT std_logic_vector(2 downto 0);
          MUX_FRW_A_CTRL:   OUT std_logic_vector(2 downto 0);
          MUX_FRW_B_CTRL:   OUT std_logic_vector(2 downto 0);
          MUX_FRW_C_CTRL:   OUT std_logic_vector(2 downto 0);
          MUX_FRW_D_CTRL:   OUT std_logic_vector(2 downto 0));
end component;
----------------------STAGE 1 SIGNALS-------------------------------------
--
signal RD_stage_1_reg_out, RD_stage_2_reg_out, RF_addr_wr:
std_logic_vector(4 downto 0);
signal write_back, aligned_write_back: std_logic_vector(63 downto 0);
signal A_reg_out_int, B_reg_out_int, A_reg_out_float, B_reg_out_float,
A_reg_out_double, B_reg_out_double: std_logic_vector(63 downto 0);
signal A_SEL_RF, B_SEL_RF: std_logic_vector(63 downto 0);
signal IMM_1, IMM_2, IMM_1_reg_out, IMM_2_reg_out: std_logic_vector(63
downto 0);
signal OP_A, OP_B, RF_OP_A, RF_OP_B: std_logic_vector(63 downto 0);
signal zero_det_out: std_logic;
signal CLK_EN_INT, CLK_EN_FP, CLK_EN_DOUBLE: std_logic;
-------------------------------------------------------------------------
--
----------------------STAGE 2 SIGNALS-------------------------------------
--
signal alu_out: std_logic_vector(63 downto 0);
signal alu_reg_out, PC_ALU: std_logic_vector(63 downto 0);
signal data_memory, aligned_data_memory, B_SEL_FRW: std_logic_vector(63
downto 0);
signal FPS, FPS_reg_out : std_logic:='0';
signal FPS_EN: std_logic;
signal MUX_A_FRW_CTRL, MUX_B_FRW_CTRL, MUX_C_FRW_CTRL, MUX_D_FRW_CTRL:
std_logic_vector(2 downto 0);
signal ZERO_DET_IN_FRW: std_logic_vector(63 downto 0);
-------------------------------------------------------------------------
--
----------------------STAGE 3 SIGNALS-------------------------------------
--
signal ram_out: std_logic_vector (63 downto 0);
signal alu_write_back_reg_out: std_logic_vector(63 downto 0);
signal ALIGN_AND_ENABLE: std_logic_vector(3 downto 0);
signal CLK_EN_MEM: std_logic;
-------------------------------------------------------------------------
--
----------------------STAGE 4 SIGNALS-------------------------------------
--
signal  ALIGN_AND_ENABLE_reg_out: std_logic_vector(3 downto 0);
-------------------------------------------------------------------------
--
```

```
----------------------FEEDBACK FORWARDING SIGNALS-----------------------
--
signal FEEDBACK_ALIGNED_MEM_OUT, FEEDBACK_ALIGNED_ALU_MEM:
std_logic_vector(63 downto 0);
signal FEEDBACK_ALIGNED_ALU_OUT, FEEDBACK_ALIGNED_MEM_OUT_REG_OUT:
std_logic_vector(63 downto 0);
signal FEED_ALIGN_CTRL_EX, FEED_ALIGN_CTRL_MEM, FEED_ALIGN_CTRL_WB:
std_logic_vector(2 downto 0);
-------------------------------------------------------------------------
--

begin
---------------------------------------STAGE 1 (ID)----------------------
-----------------------------------------------------------------

INTEGER_REGISTER_FILE: DLX_RF port map (CLK => CLK,
                                        RST => RST,
                                        ENABLE => EN1,
                                        RD1 => RD1,
                                        RD2 => RD2,
                                        RD_DOUBLE => '0',
                                        WR_DOUBLE => '0',
                                        WR => WR_INT,
                                        ADD_WR => RF_addr_wr,
                                        ADD_RD1 => RS1,
                                        ADD_RD2 => RS2,
                                        DATAIN => aligned_write_back,
                                        OUT1 => A_reg_out_int,
                                        OUT2 => B_reg_out_int);

FLOATING_REGISTER_FILE: DLX_RF port map (CLK => CLK,
                                         RST => RST,
                                         ENABLE => EN1,
                                         RD1 => RD1,
                                         RD2 => RD2,
                                         RD_DOUBLE => '0',
                                         WR_DOUBLE => '0',
                                         WR => WR_FLOAT,
                                         ADD_WR => RF_addr_wr,
                                         ADD_RD1 => RS1,
                                         ADD_RD2 => RS2,
                                         DATAIN => aligned_write_back,
                                         OUT1 => A_reg_out_float,
                                         OUT2 => B_reg_out_float);

DOUBLE_REGISTER_FILE:   DLX_RF port map (CLK => CLK,
                                         RST => RST,
                                         ENABLE => EN1,
                                         RD1 => RD1,
                                         RD2 => RD2,
                                         RD_DOUBLE => '1',
                                         WR_DOUBLE => '1',
                                         WR => WR_DOUBLE,
                                         ADD_WR => RF_addr_wr,
```

```
                                        ADD_RD1 => RS1,
                                        ADD_RD2 => RS2,
                                        DATAIN => aligned_write_back,
                                        OUT1 => A_reg_out_double,
                                        OUT2 => B_reg_out_double);
--CLOCK GATING ENABLES.
CLK_EN_INT    <= CLK  and CLK_EN_RF(0);
CLK_EN_FP     <= CLK  and CLK_EN_RF(1);
CLK_EN_DOUBLE <= CLK  and CLK_EN_RF(2);

IN_1: Register_Generic generic map (64)
                       port map (IMM_1 ,CLK, EN1, IMM_1_reg_out);
IN_2: Register_Generic generic map (64)
                       port map (IMM_2 ,CLK, EN1, IMM_2_reg_out);
ZERO_DET: Zero_detector generic map (64)
                        port map (ZERO_DET_IN_FRW, zero_det_out);

IMM_1(31 downto 0) <= INP1;
IMM_1(63 downto 32)<= (others => '0');

--S_U_EXT flags the immediate sign, in case of J-TYPE instructions left
S_U_EXT to '0'.
EXT_IMMEDIATE: EXT_IMM port map (S_U_EXT, INP2, IMM_2);

-------------------------------------------------------------------------
-------------------------------------------------------------
---------------------------------------STAGE 2 (EX)-----------------------
-------------------------------------------------------------
EX_ALU: ALU port map (OPA => OP_A,
                      OPB => OP_B,
                      integer_ctrl => INT_CTRL,
                      single_precision_fp_ctrl => SINGLE_FP_CTRL,
                      double_precision_fp_ctrl => DOUBLE_FP_CTRL,
                      conversion_ctrl => CONV_CTRL,
                      out_ctrl => OUT_CTRL,
                      ALU_Out => alu_out,
                      PC_ALU => PC_ALU,
                      FPS => FPS);

DATA_FORWORDING_UNIT: FORWARDING_UNIT port map (CLK => CLK,
                                                EN  => EN2,
                                                RST => RST,
                                                RF_SEL_1 => RF_SEL_1,
                                                RF_SEL_2 => RF_SEL_2,
                                                RS1_ID   => RS1,
                                                RS2_ID   => RS2,
                                                JMP_CTRL => JMP_CTRL,
                                                RD_MEM   =>
RD_stage_2_reg_out,
                                                RD_WB    => RF_addr_wr,
                                                RD1_ID   => RD1,
                                                RD2_ID   => RD2,
                                                RM_MEM   => RM,
                                                WM_EX    => WM_EX,
```

```
                                                    WM_MEM   => WM,
                                                    WR_INT_EX =>
WR_EX_STAGE(2),
                                                    WR_FP_EX  =>
WR_EX_STAGE(1),
                                                    WR_DOUBLE_EX =>
WR_EX_STAGE(0),
                                                    ALIGN_CTRL_EX =>
ALIGN_CTRL_EX,
                                                    WR_INT_WB      => WR_INT,
                                                    WR_FP_WB       =>
WR_FLOAT,
                                                    WR_DOUBLE_WB   =>
WR_DOUBLE,
                                                    FEED_ALIGN_CTRL_EX  =>
FEED_ALIGN_CTRL_EX,
                                                    FEED_ALIGN_CTRL_MEM =>
FEED_ALIGN_CTRL_MEM,
                                                    FEED_ALIGN_CTRL_WB  =>
FEED_ALIGN_CTRL_WB,
                                                    MUX_FRW_A_CTRL =>
MUX_A_FRW_CTRL,
                                                    MUX_FRW_B_CTRL =>
MUX_B_FRW_CTRL,
                                                    MUX_FRW_C_CTRL =>
MUX_C_FRW_CTRL,
                                                    MUX_FRW_D_CTRL =>
MUX_D_FRW_CTRL);

ALU_REG: Register_Generic generic map (64)
                          port map (alu_out ,CLK, EN2, alu_reg_out);
MEM_REG: Register_Generic generic map (64)
                          port map (B_SEL_FRW ,CLK, EN2, data_memory);

FPS_REG: FFD              port map (FPS ,CLK, FPS_EN, RST ,FPS_reg_out);

FPS_EN <= SINGLE_FP_CTRL(4) or SINGLE_FP_CTRL(3) or SINGLE_FP_CTRL(2) or
DOUBLE_FP_CTRL(4) or DOUBLE_FP_CTRL(3) or DOUBLE_FP_CTRL(2); --IF CMP FP
IS USED.

JMP_LOGIC: LOGIC_JMP port map (zero_det_out, FPS_reg_out, JMP_CTRL,
B_T_NT);

NPC <= PC_ALU(31 downto 0);

--A_SEL_RF MUX
A_SEL_RF <=  A_reg_out_int    when RF_SEL_1 = "00" else
             A_reg_out_float  when RF_SEL_1 = "01" else
             A_reg_out_double when RF_SEL_1 = "10" else
             A_reg_out_int;
--B_SEL_RF MUX
B_SEL_RF <=  B_reg_out_int    when RF_SEL_2 = "00" else
             B_reg_out_float  when RF_SEL_2 = "01" else
             B_reg_out_double when RF_SEL_2 = "10" else
```

```
            B_reg_out_int;


--MUX A
RF_OP_A <= IMM_1_reg_out when S1 = '0' else
           A_SEL_RF;

--MUX B
RF_OP_B <= B_SEL_RF when S2 = '0' else
           IMM_2_reg_out;

--FORWARD MUX A
OP_A <= RF_OP_A                              when MUX_A_FRW_CTRL = "000"
else
        FEEDBACK_ALIGNED_ALU_MEM             when MUX_A_FRW_CTRL = "001"
else
        FEEDBACK_ALIGNED_ALU_OUT             when MUX_A_FRW_CTRL = "010"
else
        FEEDBACK_ALIGNED_MEM_OUT             when MUX_A_FRW_CTRL = "011"
else
        FEEDBACK_ALIGNED_MEM_OUT_REG_OUT     when MUX_A_FRW_CTRL = "100"
else
        RF_OP_A;

--FORWARD MUX B
OP_B <= RF_OP_B                              when MUX_B_FRW_CTRL = "000"
else
        FEEDBACK_ALIGNED_ALU_MEM             when MUX_B_FRW_CTRL = "001"
else
        FEEDBACK_ALIGNED_ALU_OUT             when MUX_B_FRW_CTRL = "010"
else
        FEEDBACK_ALIGNED_MEM_OUT             when MUX_B_FRW_CTRL = "011"
else
        FEEDBACK_ALIGNED_MEM_OUT_REG_OUT     when MUX_B_FRW_CTRL = "100"
else
        RF_OP_B;

--FORWARD MUX B FOR DATA MEMORY.
B_SEL_FRW <= B_SEL_RF                                when MUX_C_FRW_CTRL =
"000" else
             FEEDBACK_ALIGNED_ALU_MEM                when MUX_C_FRW_CTRL =
"001" else
             FEEDBACK_ALIGNED_ALU_OUT                when MUX_C_FRW_CTRL =
"010" else
             FEEDBACK_ALIGNED_MEM_OUT                when MUX_C_FRW_CTRL =
"011" else
             FEEDBACK_ALIGNED_MEM_OUT_REG_OUT        when MUX_C_FRW_CTRL =
"100" else
             B_SEL_RF;

--FORWARD MUX D FOR DATA MEMORY.
ZERO_DET_IN_FRW <= A_reg_out_int                          when
MUX_D_FRW_CTRL = "000" else
```

```
                  FEEDBACK_ALIGNED_ALU_MEM            when
MUX_D_FRW_CTRL = "001" else
                  FEEDBACK_ALIGNED_ALU_OUT            when
MUX_D_FRW_CTRL = "010" else
                  FEEDBACK_ALIGNED_MEM_OUT            when
MUX_D_FRW_CTRL = "011" else
                  FEEDBACK_ALIGNED_MEM_OUT_REG_OUT    when
MUX_D_FRW_CTRL = "100" else
                  A_reg_out_int;
-------------------------------------------------------------------------
-------------------------------------------------------------
-------------------------------------STAGE 3 (MEM)----------------------
-------------------------------------------------------------
--RAM: DLX_RAM generic map(5) --1024 byte = 1kB.
--             port map (CLK => CLK,
--                       RST => RST,
--                       RD_M => RM,
--                       WR_M => WM,
--                       RD_DOUBLE => RM_DOUBLE,
--                       WR_DOUBLE => WM_DOUBLE,
--                       ALIGN_CTRL => ALIGN_CTRL,
--                       EN => EN3,
--                       ADDR => alu_reg_out(4 DOWNTO 0),
--                       DATAIN => data_memory,--CAMBIATO
--                       DATAOUT => ram_out);
--RAM SIGNALS.
ram_out    <= DATAOUT_RAM;
DATAIN_RAM <= data_memory;
ADDR_RAM   <= alu_reg_out(31 downto 0);

M_A_A: MASK_AND_ALIGN  port map (data_memory, ALIGN_CTRL,
aligned_data_memory);
MEM_REG_ALU: Register_Generic generic map(64)
                              port map(alu_reg_out, CLK, EN3,
alu_write_back_reg_out);
ALIGN_AND_ENABLE <= ALIGN_CTRL&EN3;
-------------------------------------------------------------------------
-------------------------------------------------------------
ALIGN_EN_STAGE3_TO_4: Register_Generic generic map(4)
                                       port map (ALIGN_AND_ENABLE, CLK,
'1', ALIGN_AND_ENABLE_reg_out);

-------------------------------------STAGE 4 (WB)-----------------------
-------------------------------------------------------------
--MUX C
write_back <= ram_out when S3 = '0' else
              alu_write_back_reg_out;


S_A_A: SHIFT_AND_ALIGN port map (write_back, ALIGN_AND_ENABLE_reg_out(3
downto 1), aligned_write_back);
-------------------------------------------------------------------------
-------------------------------------------------------------
```

```
REGISTER_OUT: Register_Generic generic map (64)
                      port map (aligned_write_back ,CLK,
ALIGN_AND_ENABLE_reg_out(0), DLX_OUT);

-----------------------------------DELAY RD 2 CC-----------------------
-------------------------------------------------------------
STAGE_1_RD: Register_Generic generic map (5)
                      port map (RD ,CLK, EN1, RD_stage_1_reg_out);
STAGE_2_RD: Register_Generic generic map (5)
                      port map (RD_stage_1_reg_out ,CLK, EN2,
RD_stage_2_reg_out);
STAGE_3_RD: Register_Generic generic map (5)
                      port map (RD_stage_2_reg_out ,CLK, EN3,
RF_addr_wr);
-------------------------------------------------------------------------
-------------------------------------------------------------

-----------------------------------DATA FORWARDING FEEDBACKS------------
-------------------------------------------------------------------------
S_A_A_FEEDBACK_ALU_EX:  SHIFT_AND_ALIGN port map (alu_reg_out,
FEED_ALIGN_CTRL_MEM, FEEDBACK_ALIGNED_ALU_OUT);
S_A_A_FEEDBACK_ALU_MEM: SHIFT_AND_ALIGN port map (alu_write_back_reg_out,
FEED_ALIGN_CTRL_WB, FEEDBACK_ALIGNED_ALU_MEM);
S_A_A_FEEDBACK_MEM:     SHIFT_AND_ALIGN port map (ram_out,
FEED_ALIGN_CTRL_MEM, FEEDBACK_ALIGNED_MEM_OUT);

FEEDBACK_MEM_WB_REG: Register_Generic generic map(64)
                                       port map(FEEDBACK_ALIGNED_MEM_OUT,
CLK, '1', FEEDBACK_ALIGNED_MEM_OUT_REG_OUT);
-------------------------------------------------------------------------
-------------------------------------------------------------------------

end Structural;
```

E. ALU VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ALU is
    Port (
        OPA, OPB     : in std_logic_vector(63 downto 0);  -- 2 inputs N-bit
        integer_ctrl : in std_logic_vector(14 downto 0);
        single_precision_fp_ctrl : in std_logic_vector(6 downto 0);
        double_precision_fp_ctrl : in std_logic_vector(6 downto 0);
        conversion_ctrl : in std_logic_vector(4 downto 0);
        out_ctrl : in std_logic_vector(2 downto 0);
        ALU_Out   : out std_logic_vector(63 downto 0);
        PC_ALU: out std_logic_vector(63 downto 0);
        FPS : out std_logic);
end ALU;

architecture Structural of ALU is


component integer_ALU is
    generic (constant N: natural := 64);
    Port (
        OPA, OPB     : in std_logic_vector(N-1 downto 0);  -- 2 inputs N-bit
        ADD_SUB, sign_usign_n, ENABLE : in std_logic; -- operation selectors
        logic_control, shifter_control, comparator_control, out_control : in
std_logic_vector(2 downto 0); -- operation and otput selectors
        ALU_Out   : out std_logic_vector(N-1 downto 0);
        cmp_out : out std_logic); -- 1 output N-bit
end component;

component Floating_Point_Single_Precision_unit is
    port(FP_a: in std_logic_vector(63 downto 0);
         FP_b: in std_logic_vector(63 downto 0);
         EN:   in std_logic;
         ADD_SUB: in std_logic;
         CMP_CTRL: in std_logic_vector(2 downto 0);
         OP_CTRL:  in std_logic_vector(1 downto 0);
         CMP_OUT:  out std_logic;
         FP_out:   out std_logic_vector(63 downto 0));
end component;

component Floating_Point_Double_Precision_Unit is
    port(FP_a: in std_logic_vector(63 downto 0);
         FP_b: in std_logic_vector(63 downto 0);
         EN:   in std_logic;
         ADD_SUB: in std_logic;
         CMP_CTRL: in std_logic_vector(2 downto 0);
         OP_CTRL:  in std_logic_vector(1 downto 0);
         CMP_OUT:  out std_logic;
         FP_out:   out std_logic_vector(63 downto 0));
end component;

component Converter_unit is
    port (x: in std_logic_vector(63 downto 0);
```

```
          conversion_ctrl: in std_logic_vector (3 downto 0);
          y: out std_logic_vector(63 downto 0));
end component;

component Floating_point_single_precision_to_integer is
    Port (FP_in : in std_logic_vector (31 downto 0);
          s_u: in std_logic;    --Specify if output format is signed or
unsigned.
          integer_out: out std_logic_vector (31 downto 0));
end component;


signal converted_OPA, converted_OPB, out_integer, out_float, out_double,
float_to_int_out, integer_cmp_out: std_logic_vector(63 downto 0);
signal int_cmp, float_cmp, double_cmp: std_logic;
begin

conv_A: Converter_unit port map(OPA, conversion_ctrl(3 downto 0),
converted_OPA);

conv_B: Converter_unit port map(OPB, conversion_ctrl(3 downto 0),
converted_OPB);

int: integer_ALU port map (OPA => converted_OPA,
                           OPB => converted_OPB,
                           ENABLE => integer_ctrl(14),
                           ADD_SUB => integer_ctrl(13),
                           sign_usign_n => integer_ctrl(12),
                           logic_control => integer_ctrl(11 downto 9),
                           shifter_control => integer_ctrl(8 downto 6),
                           comparator_control => integer_ctrl(5 downto 3),
                           out_control => integer_ctrl(2 downto 0),
                           ALU_Out => out_integer,
                           cmp_out => int_cmp);

float_single: Floating_Point_Single_Precision_unit port map (FP_a =>
converted_OPA,
                                                        FP_b => converted_OPB,
                                                        EN =>
single_precision_fp_ctrl(6),
                                                        ADD_SUB =>
single_precision_fp_ctrl(5),
                                                        CMP_CTRL =>
single_precision_fp_ctrl(4 downto 2),
                                                        OP_CTRL =>
single_precision_fp_ctrl(1 downto 0),
                                                        CMP_OUT => float_cmp,
                                                        FP_out => out_float);

float_double: Floating_Point_Double_Precision_unit port map (FP_a =>
converted_OPA,
                                                               FP_b =>
converted_OPB,
                                                               EN =>
double_precision_fp_ctrl(6),
                                                               ADD_SUB =>
double_precision_fp_ctrl(5),
                                                               CMP_CTRL =>
double_precision_fp_ctrl(4 downto 2),
                                                               OP_CTRL =>
double_precision_fp_ctrl(1 downto 0),
                                                               CMP_OUT =>
double_cmp,
                                                               FP_out =>
```

```
out_double);

fp_to_int: Floating_point_single_precision_to_integer port map (out_float(31
downto 0), conversion_ctrl(4), float_to_int_out(31 downto 0));

float_to_int_out(63 downto 32) <= (others => '0');

integer_cmp_out (0) <= int_cmp;
integer_cmp_out(63 downto 1) <= (others => '0');

ALU_out <= out_integer                                  when out_ctrl="000" else
           float_to_int_out                             when out_ctrl="001" else
           out_float                                    when out_ctrl="010" else
           out_double                                   when out_ctrl="011" else
           converted_OPA                                when out_ctrl="100" else
           std_logic_vector(unsigned(converted_OPA)+4)  when out_ctrl="101" else
           integer_cmp_out                              when out_ctrl="110" else
           converted_OPB                                when out_ctrl="111";

FPS <= float_cmp or double_cmp;

PC_ALU <=  out_integer;
--FPS <= int_cmp when out_ctrl="101" else
--           float_cmp when out_ctrl="110" else
--           double_cmp when out_ctrl="111" else
--           '0';

end Structural;
```

F. FORWARDING UNIT VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity FORWARDING_UNIT is
   port (CLK:            IN std_logic;
         EN:             IN std_logic;
         RST:            IN std_logic;
         RF_SEL_1:       IN std_logic_vector(1 downto 0);
         RF_SEL_2:       IN std_logic_vector(1 downto 0);
         RS1_ID:         IN std_logic_vector(4 downto 0);
         RS2_ID:         IN std_logic_vector(4 downto 0);
         RD_MEM:         IN std_logic_vector(4 downto 0);
         RD_WB:          IN std_logic_vector(4 downto 0);
         RD1_ID:         IN std_logic;
         RD2_ID:         IN std_logic;
         JMP_CTRL:       IN std_logic_vector(2 downto 0);
         RM_MEM:         IN std_logic;
         WM_EX:          IN std_logic;
         WM_MEM:         IN std_logic;
         WR_INT_EX:      IN std_logic;
         WR_FP_EX:       IN std_logic;
         WR_DOUBLE_EX:   IN std_logic;
         ALIGN_CTRL_EX:  IN std_logic_vector(2 downto 0);
         WR_INT_WB:      IN std_logic;
         WR_FP_WB:       IN std_logic;
         WR_DOUBLE_WB:   IN std_logic;
         FEED_ALIGN_CTRL_EX: OUT std_logic_vector(2 downto 0);
         FEED_ALIGN_CTRL_MEM:OUT std_logic_vector(2 downto 0);
         FEED_ALIGN_CTRL_WB: OUT std_logic_vector(2 downto 0);
         MUX_FRW_A_CTRL:  OUT std_logic_vector(2 downto 0);
         MUX_FRW_B_CTRL:  OUT std_logic_vector(2 downto 0);
         MUX_FRW_C_CTRL:  OUT std_logic_vector(2 downto 0);
         MUX_FRW_D_CTRL:  OUT std_logic_vector(2 downto 0));
end FORWARDING_UNIT;

architecture Behavioral of FORWARDING_UNIT is

component Register_Generic_rst is
   generic (N: integer:= 64);
   port (data_in: in std_logic_vector(N-1 downto 0);
         clk:    in std_logic;
         rst:    in std_logic;
         en:     in std_logic;
```

```
        data_out: out std_logic_vector(N-1 downto 0));
end component;

signal RS_RD_ID, RS_RD_EX_reg_out: std_logic_vector(11 downto 0);
signal WR_EX, WR_MEM: std_logic_vector(4 downto 0);
signal RS1_EX, RS2_EX: std_logic_vector(4 downto 0);
signal RD1_EX, RD2_EX: std_logic;
signal RM_WB, WM_WB: std_logic;
signal WR_INT_MEM, WR_FP_MEM, WR_DOUBLE_MEM: std_logic;
signal tmp_MUX_FRW_B_CTRL, tmp_MUX_FRW_A_CTRL, tmp_MUX_FRW_C_CTRL, tmp_MUX_FRW_D
_CTRL:  std_logic_vector(2 downto 0);
signal ALIGN_CTRL_MEM, ALIGN_CTRL_WB: std_logic_vector(2 downto 0);
signal JMP_FRW: std_logic;
begin

RS_RD_EX_STAGE: Register_Generic_rst generic map (12)
                        port map (RS_RD_ID, CLK, RST ,'1', RS_RD_EX_reg_out);
WR_MEM_STAGE:   Register_Generic_rst generic map (5)
                        port map (WR_EX, CLK, RST ,'1', WR_MEM);
ALIGN_CTRL_MEM_STAGE: Register_Generic_rst generic map (3)
                        port map (ALIGN_CTRL_EX, CLK, RST ,'1', ALIGN_CTRL_MEM);
ALIGN_CTRL_WB_STAGE: Register_Generic_rst generic map (3)
                        port map (ALIGN_CTRL_MEM, CLK, RST ,'1', ALIGN_CTRL_WB);

RS_RD_ID <= RS1_ID&RS2_ID&RD1_ID&RD2_ID;
WR_EX <= WR_INT_EX&WR_FP_EX&WR_DOUBLE_EX&RM_MEM&WM_MEM;


RS1_EX <= RS_RD_EX_reg_out(11 downto 7);
RS2_EX <= RS_RD_EX_reg_out(6 downto 2);
RD1_EX <= RS_RD_EX_reg_out(1);
RD2_EX <= RS_RD_EX_reg_out(0);

WR_INT_MEM      <= WR_MEM(4);
WR_FP_MEM       <= WR_MEM(3);
WR_DOUBLE_MEM   <= WR_MEM(2);
RM_WB           <= WR_MEM(1);
WM_WB           <= WR_MEM(0);

--CASE 011 or 100 => BEQZ or BNEZ.
JMP_FRW <=  ((not JMP_CTRL(2)) and JMP_CTRL(1) and JMP_CTRL(0)) OR ((JMP_CTRL(2)) and (not JMP_
CTRL(1)) and (not JMP_CTRL(0)));

   process(RF_SEL_1, RF_SEL_2, RS1_EX, RS2_EX, RD_WB , RD1_EX, RD2_EX ,RD_MEM, WR_INT_MEM
, WR_FP_MEM, WR_DOUBLE_MEM, WR_INT_WB, WR_FP_WB, WR_DOUBLE_WB, RM_MEM, WM_ME
M, WM_EX ,RM_WB, WM_WB, JMP_FRW)
   begin
-----------------------------------------------------------------OPA MUX CTRL-----------------------------------------------------
-----------------------------------
--INTEGERS.
      --INTEGER FROM ALU(MEM).
      if ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD1_EX = '1') AND RF_
SEL_1 = "00" AND RM_MEM = '0'  AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "010";
```

```
      --INTEGER FROM ALU (WB).
      elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD1_EX = '1') AND RF_S
EL_1 = "00" AND RM_WB = '0'  AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "001";
      --INTEGER FROM MEMORY (MEM).
      elsif ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD1_EX = '1') AND R
F_SEL_1 = "00" AND RM_MEM = '1'  AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "011";
      --INTEGER FROM MEMORY (WB).
      elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD1_EX = '1') AND RF_S
EL_1 = "00" AND RM_WB = '1'  AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "100";
--FLOAT SINGLE.
      --SINGLE FROM ALU(MEM).
      elsif ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_FP_MEM = '1') AND (RD1_EX = '1') AND RF
_SEL_1 = "01" AND RM_MEM = '0'  AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "010";
      --SINGLE FROM ALU (WB).
      elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_FP_WB = '1') AND (RD1_EX = '1') AND RF_SE
L_1 = "01" AND RM_WB = '0'  AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "001";
      --SINGLE FROM MEMORY (MEM).
      elsif ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_FP_MEM = '1') AND (RD1_EX = '1') AND RF
_SEL_1 = "01" AND RM_MEM = '1' AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "011";
      --SINGLE FROM MEMORY (WB).
      elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_FP_WB = '1') AND (RD1_EX = '1') AND RF_SE
L_1 = "01" AND RM_WB = '1' AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "100";
--FLOAT DOUBLE.
      --DOUBLE FROM ALU(MEM).
      elsif ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_DOUBLE_MEM = '1') AND (RD1_EX = '1') A
ND RF_SEL_1 = "10" AND RM_MEM = '0' AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "010";
      --DOUBLE FROM ALU (WB).
      elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_DOUBLE_WB = '1') AND (RD1_EX = '1') AND
 RF_SEL_1 = "10" AND RM_WB = '0' AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "001";
      --DOUBLE FROM MEMORY (MEM).
      elsif ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_DOUBLE_MEM = '1') AND (RD1_EX = '1') A
ND RF_SEL_1 = "10" AND RM_MEM = '1' AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "011";
      --DOUBLE FROM MEMORY (WB).
      elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_DOUBLE_WB = '1') AND (RD1_EX = '1') AND
 RF_SEL_1 = "10" AND RM_WB = '1' AND JMP_FRW = '0') then
         tmp_MUX_FRW_A_CTRL <= "100";
      else
      --NO RAW.
         tmp_MUX_FRW_A_CTRL <= "000";
      end if;
-------------------------------------------------------------------------------------------------------------------------------
-------------------------
-----------------------------------------------------------OPB MUX CTRL-----------------------------------------------------
-----------------------------------
```

```
--INTEGERS.
	--INTEGER FROM ALU(MEM).
	if ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD2_EX = '1') AND RF_
SEL_2 = "00" AND RM_MEM = '0' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "010";
	--INTEGER FROM ALU (WB).
	elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD2_EX = '1') AND RF_S
EL_2 = "00" AND RM_WB = '0' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "001";
	--INTEGER FROM MEMORY (MEM).
	elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD2_EX = '1') AND R
F_SEL_2 = "00" AND RM_MEM = '1' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "011";
	--INTEGER FROM MEMORY (WB).
	elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD2_EX = '1') AND RF_S
EL_2 = "00" AND RM_WB = '1' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "100";
--FLOAT SINGLE.
	--SINGLE FROM ALU(MEM).
	elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_FP_MEM = '1') AND (RD2_EX = '1') AND RF
_SEL_2 = "01" AND RM_MEM = '0' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "010";
	--SINGLE FROM ALU (WB).
	elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_FP_WB = '1') AND (RD2_EX = '1') AND RF_SE
L_2 = "01" AND RM_WB = '0' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "001";
	--SINGLE FROM MEMORY (MEM).
	elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_FP_MEM = '1') AND (RD2_EX = '1') AND RF
_SEL_2 = "01" AND RM_MEM = '1' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "011";
	--SINGLE FROM MEMORY (WB).
	elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_FP_WB = '1') AND (RD2_EX = '1') AND RF_SE
L_2 = "01" AND RM_WB = '1' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "100";
--FLOAT DOUBLE.
	--DOUBLE FROM ALU(MEM).
	elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_DOUBLE_MEM = '1') AND (RD2_EX = '1') A
ND RF_SEL_2 = "10" AND RM_MEM = '0' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "010";
	--DOUBLE FROM ALU (WB).
	elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_DOUBLE_WB = '1') AND (RD2_EX = '1') AND
 RF_SEL_2 = "10" AND RM_WB = '0' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "001";
	--DOUBLE FROM MEMORY (MEM).
	elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_DOUBLE_MEM = '1') AND (RD2_EX = '1') A
ND RF_SEL_2 = "10" AND RM_MEM = '1' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "011";
	--DOUBLE FROM MEMORY (WB).
	elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_DOUBLE_WB = '1') AND (RD2_EX = '1') AND
 RF_SEL_2 = "10" AND RM_WB = '1' AND WM_EX = '0' AND JMP_FRW = '0') then
		tmp_MUX_FRW_B_CTRL <= "100";
	else
	--NO RAW.
		tmp_MUX_FRW_B_CTRL <= "000";
```

```
        end if;
-------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------OPC MUX CTRL-----------------------------------------------------------------------------
--INTEGERS.
        --INTEGER FROM ALU(MEM).
        if ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD2_EX = '1') AND RF_
SEL_2 = "00" AND RM_MEM = '0' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "010";
        --INTEGER FROM ALU (WB).
        elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD2_EX = '1') AND RF_S
EL_2 = "00" AND RM_WB = '0' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "001";
        --INTEGER FROM MEMORY (MEM).
        elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD2_EX = '1') AND R
F_SEL_2 = "00" AND RM_MEM = '1' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "011";
        --INTEGER FROM MEMORY (WB).
        elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD2_EX = '1') AND RF_S
EL_2 = "00" AND RM_WB = '1' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "100";
--FLOAT SINGLE.
        --SINGLE FROM ALU(MEM).
        elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_FP_MEM = '1') AND (RD2_EX = '1') AND RF
_SEL_2 = "01" AND RM_MEM = '0' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "010";
        --SINGLE FROM ALU (WB).
        elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_FP_WB = '1') AND (RD2_EX = '1') AND RF_SE
L_2 = "01" AND RM_WB = '0' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "001";
        --SINGLE FROM MEMORY (MEM).
        elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_FP_MEM = '1') AND (RD2_EX = '1') AND RF
_SEL_2 = "01" AND RM_MEM = '1' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "011";
        --SINGLE FROM MEMORY (WB).
        elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_FP_WB = '1') AND (RD2_EX = '1') AND RF_SE
L_2 = "01" AND RM_WB = '1' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "100";
--FLOAT DOUBLE.
        --DOUBLE FROM ALU(MEM).
        elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_DOUBLE_MEM = '1') AND (RD2_EX = '1') A
ND RF_SEL_2 = "10" AND RM_MEM = '0' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "010";
        --DOUBLE FROM ALU (WB).
        elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_DOUBLE_WB = '1') AND (RD2_EX = '1') AND
 RF_SEL_2 = "10" AND RM_WB = '0' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "001";
        --DOUBLE FROM MEMORY (MEM).
        elsif ((unsigned(RS2_EX) = unsigned(RD_MEM)) AND (WR_DOUBLE_MEM = '1') AND (RD2_EX = '1') A
ND RF_SEL_2 = "10" AND RM_MEM = '1' AND WM_EX = '1' AND JMP_FRW = '0') then
            tmp_MUX_FRW_C_CTRL <= "011";
        --DOUBLE FROM MEMORY (WB).
        elsif ((unsigned(RS2_EX) = unsigned(RD_WB)) AND (WR_DOUBLE_WB = '1') AND (RD2_EX = '1') AND
```

```
 RF_SEL_2 = "10" AND RM_WB = '1' AND WM_EX = '1' AND JMP_FRW = '0') then
          tmp_MUX_FRW_C_CTRL <= "100";
       else
       --NO RAW.
          tmp_MUX_FRW_C_CTRL <= "000";
       end if;
-------------------------------------------------------------------------------------------------------------------------------------------
--------------------------
-----------------------------------------------------------------OPD MUX CTRL-----------------------------------------------------------
-------------------------------------
--INTEGERS.
       --INTEGER FROM ALU(MEM).
       if ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD1_EX = '1') AND RF_
SEL_1 = "00" AND RM_MEM = '0' AND WM_EX = '0' AND JMP_FRW = '1') then
          tmp_MUX_FRW_D_CTRL <= "010";
       --INTEGER FROM ALU (WB).
       elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD1_EX = '1') AND RF_S
EL_1 = "00" AND RM_WB = '0' AND WM_EX = '0' AND JMP_FRW = '1') then
          tmp_MUX_FRW_D_CTRL <= "001";
       --INTEGER FROM MEMORY (MEM).
       elsif ((unsigned(RS1_EX) = unsigned(RD_MEM)) AND (WR_INT_MEM = '1') AND (RD1_EX = '1') AND R
F_SEL_1 = "00" AND RM_MEM = '1' AND WM_EX = '0' AND JMP_FRW = '1') then
          tmp_MUX_FRW_D_CTRL <= "011";
       --INTEGER FROM MEMORY (WB).
       elsif ((unsigned(RS1_EX) = unsigned(RD_WB)) AND (WR_INT_WB = '1') AND (RD1_EX = '1') AND RF_S
EL_1 = "00" AND RM_WB = '1' AND WM_EX = '0' AND JMP_FRW = '1') then
          tmp_MUX_FRW_D_CTRL <= "100";
       else
       --NO RAW.
          tmp_MUX_FRW_D_CTRL <= "000";
       end if;
-------------------------------------------------------------------------------------------------------------------------------------------
--------------------------

   end process;

MUX_FRW_A_CTRL <= tmp_MUX_FRW_A_CTRL when EN = '1' else
            (others => '0');
MUX_FRW_B_CTRL <= tmp_MUX_FRW_B_CTRL when EN = '1' else
            (others => '0');
MUX_FRW_C_CTRL <= tmp_MUX_FRW_C_CTRL when EN = '1' else
            (others => '0');
MUX_FRW_D_CTRL <= tmp_MUX_FRW_D_CTRL when EN = '1' else
            (others => '0');
FEED_ALIGN_CTRL_EX  <= ALIGN_CTRL_EX;
FEED_ALIGN_CTRL_MEM <= ALIGN_CTRL_MEM;
FEED_ALIGN_CTRL_WB  <= ALIGN_CTRL_WB;
end Behavioral;
```

G. JUMP LOGIC VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity LOGIC_JMP is
    port (zero_det: in std_logic;
          FPS:      in std_logic;
          JMP_CTRL: in std_logic_vector(2 downto 0);
          B_T_NT:   OUT std_logic);
end LOGIC_JMP;

architecture Behevioral of LOGIC_JMP is
signal MUX_CTRL: std_logic;
begin

B_T_NT  <=  '0'             when JMP_CTRL = "000" else --DEFAULT NPC =
PC+1.
            FPS             when JMP_CTRL = "010" else --CHECK FPS = '1'.
            not FPS         when JMP_CTRL = "001" else --CHECK FPS = '0'.
            zero_det        when JMP_CTRL = "011" else --CHECK EQZERO.
            not zero_det    when JMP_CTRL = "100" else --CHECK NEQZ.
            '1'             when JMP_CTRL = "101" else --JUMP
UNCONDITION.
            '0';

end Behevioral;
```