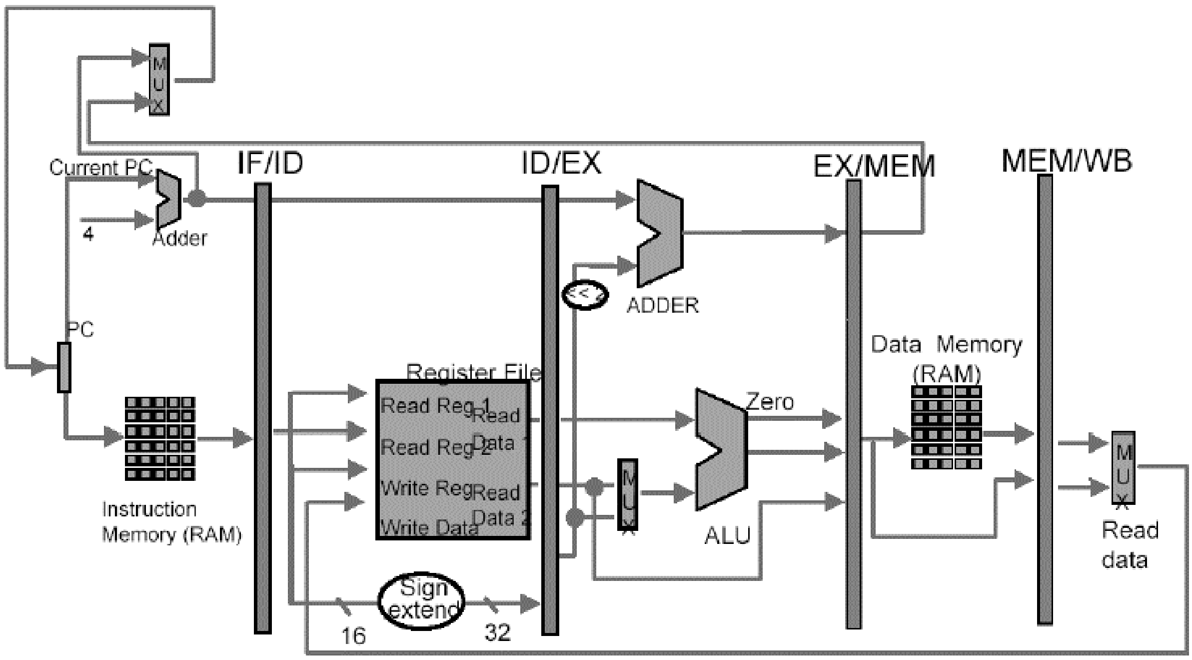
Lab 3

Architecture of CPU

Designing a pipelined MIPS on FPGA

VHDL, QUARTUS & FPGA



Maor Assayag 318550746
Refael Shetrit 204654891
Facilitators: Boris Braginsky & Prof. Hugo Guterman

Table of contents

General Description

1.1 Aims of the Laboratory

The aim of this laboratory is to design a simple MIPS compatible CPU. The CPU will use a PIPELINED architecture and must be capable of performing instructions from MIPS instruction set. The design will be executed on the Altera Board. The MIPS architecture is Harvard architecture in order to increase throughput and simplify the logic. There is need to implement floating point instructions (ADD, SUB and MUL from previous work) and floating-point register file.

1.2 Assignment definition

You must design a pipelined MIPS compatible CPU (at least 4 stages). All the possible hazards must be solved in hardware!
The architecture must include a MIPS ISA compatible CPU with data and program memory for hosting data and code. The block diagram of the architecture is given in The system design section. The CPU will have a standard MIPS register file. The top level and the MIPS core must be structural. The design must be compiled and loaded to the Altera board for testing. A single clock (CLK) should be used in the design.

1.3 Workspace & language

- ModelSim ALTERA STARTER EDITION 10.1b
- VHDL (2008's syntax)
- ATOM editor version 1.25.1
- Quartus II 12.1 Web Edition (32-Bit) & Altera DE1 FPGA

System Design

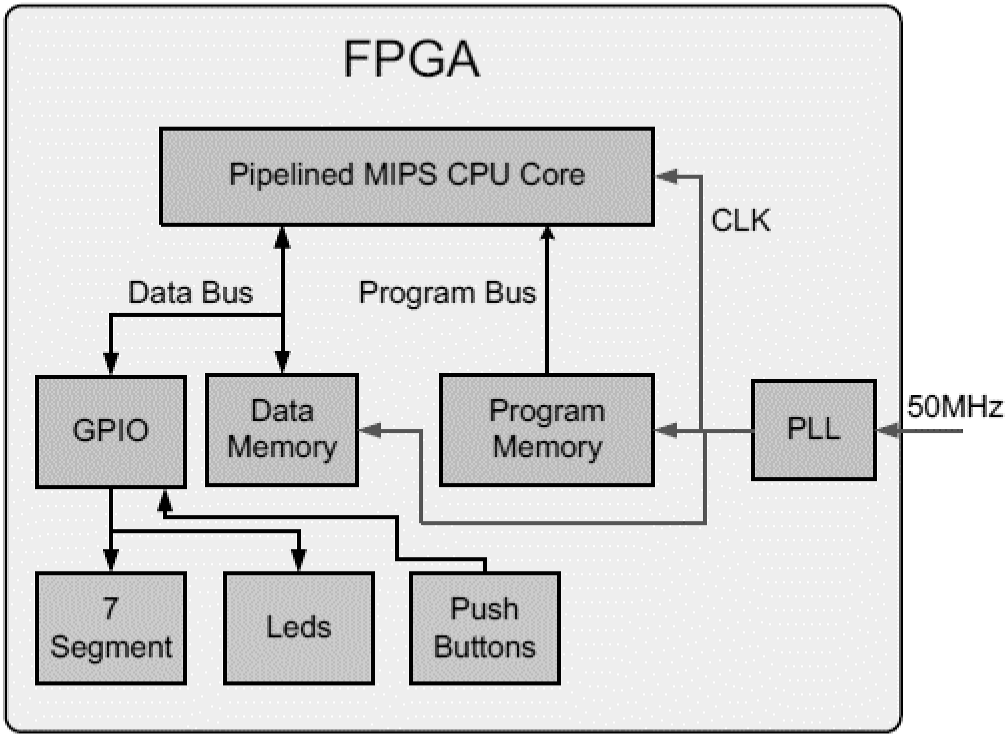


Figure 1.1: Overall system design

Mnemonic	Format	Opcode Field	Function Field	Instruction
Add	R	0	32	Add
Addi	I	8	-	Add Immediate
Addu	R	0	33	Add Unsigned
Sub	R	0	34	Subtract
Subu	R	0	35	Subtract Unsigned
And	R	0	36	Bitwise And
Or	R	0	37	Bitwise OR
Sll	R	0	0	Shift Left Logical
Srl	R	0	2	Shift Right Logical
Slt	R	0	42	Set if Less Than
Lui	I	15	-	Load Upper Immediate
Lw	I	35	-	Load Word
Sw	I	43	-	Store Word
Beq	I	4	-	Branch on Equal
Bne	I	5	-	Branch on Not Equal
J	J	2	-	Jump
Jal	J	3	-	Jump and Link (used for Call)
Jr	R	0	8	Jump Register (used for Return)

AddF	R	0	50	Add in FPU
SubF	R	0	45	Sub in FPU
MulF	R	0	47	Mul in FPU
SltF	R	0	61	Slt in FPU

Table 1.1: MIPS Op Codes

The PLL is used to make a higher frequency from the 50MHz clock and is used in FPGA compilation only.
The GPIO (General Purpose I/O) is a simple buffer registers mapped to some data address (Higher than data memory) that enables the CPU to output data to LEDS and 7-Segment and to read the Push-Buttons state.

The CPU will be based on standard 32bit MIPS ISA and the Instructions will be 32-bit wide.

The following table shows the MIPS instruction format. For more information see MIPS technical documents :

Field	Description
opcode	6-bit primary operation code
rd	5-bit specifier for the destination register
rs	5-bit specifier for the source register
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM
immediate	16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
sa	5-bit shift amount
function	6-bit function field used to specify functions within the primary *opcode* SPECIAL

Table 1.1: CPU Instruction Format Fields

Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Table 1.2: CPU Instruction Format

Memory	Maximal Size	Write Latency	Read Latency
Program Memory	1KByte	1 clk	1-2 clk
Data Memory	1KByte	1 clk	1-2 clk

Table 1.3: Memory sizes and latency

The FPU module from Work 2 must be connected to CPU. Additionally, interrupts from buttons must be created, using memory mapped registers and interrupt vector. Also 7 segments must be used with memory mapped registers. If necessary other IO devices (switches, led) can be used with memory mapped registers too.

As a test bench you need to write code which first of all have initialize necessary interrupts from buttons/switches. Secondary sort floating point vector (pre-stored in data memory, vector size is 8) and show the sorting vector on 7 segments display with delay of 1s.

The Top-Level design must be structural and contain the following entities:

- Floating point UNIT Entity (For MUL, ADD, SUB, SLT operations)
- Pipelined MIPS

The synchronous parts of the system will be constructed using Flip-Flops (DFF). Other entities can be designed behaviorally, structurally or mixed.

Pipeline

One of the most effective ways to speed up a digital design is to use pipelining. The processor can be divided into subparts, where each part may execute in one clock cycle. This implies that it is possible to increase the clock frequency compared to a non-pipelined design. It will also be easier to optimize each stage than trying to optimize the whole design.
While the instruction throughput increases, instruction latency is added. The architecture is using a pipeline with **5 stages** :

1. **Instruction Fetch**, instructions are fetched from the instruction memory.
2. **Instruction Decode**, instructions are decoded, and control signals are generated.
3. **Execute,** arithmetic and logic instructions are executed.
4. **Memory access**, memory is accessed on load and store instructions.
5. **Write back**, the result is written back to the appropriate register.

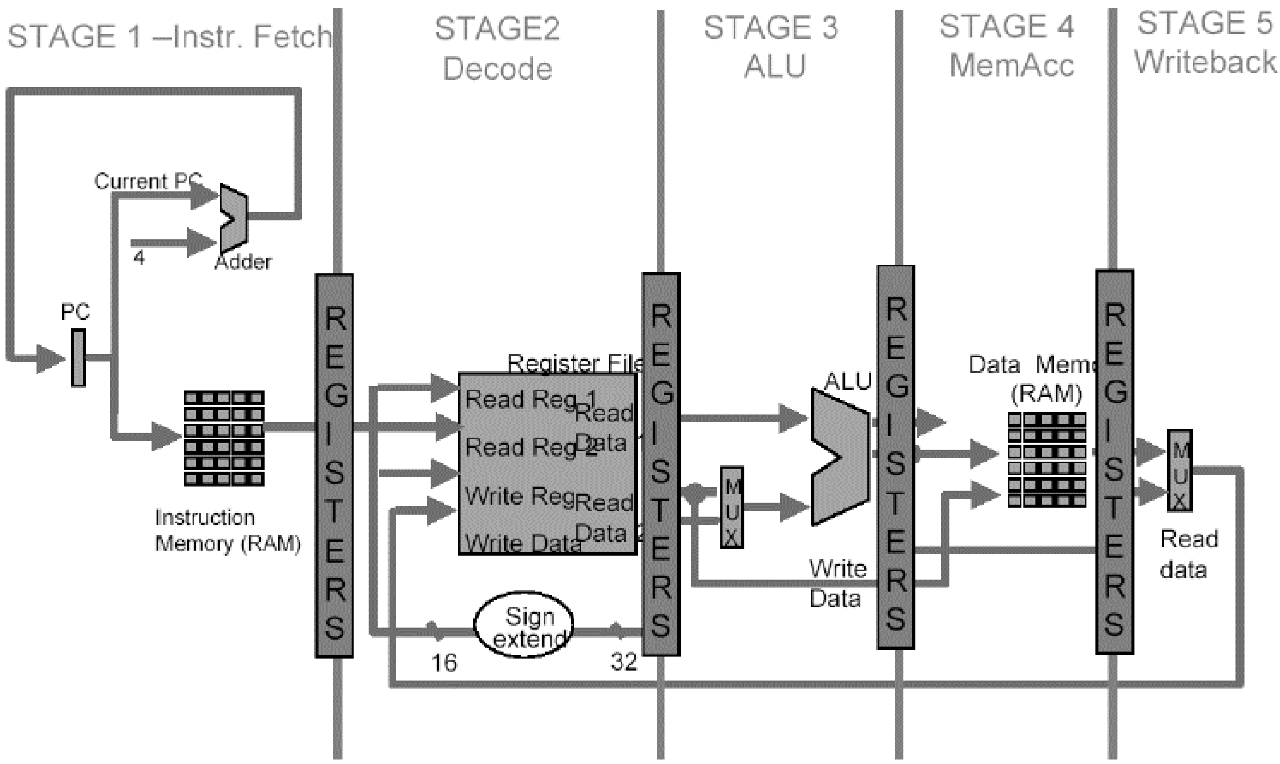


Figure 1.2: Pipeline stages

Pipeline hazards

In some cases, the next instruction cannot execute in the following clock cycle. These events are called hazards . In this design there are three types of hazards.

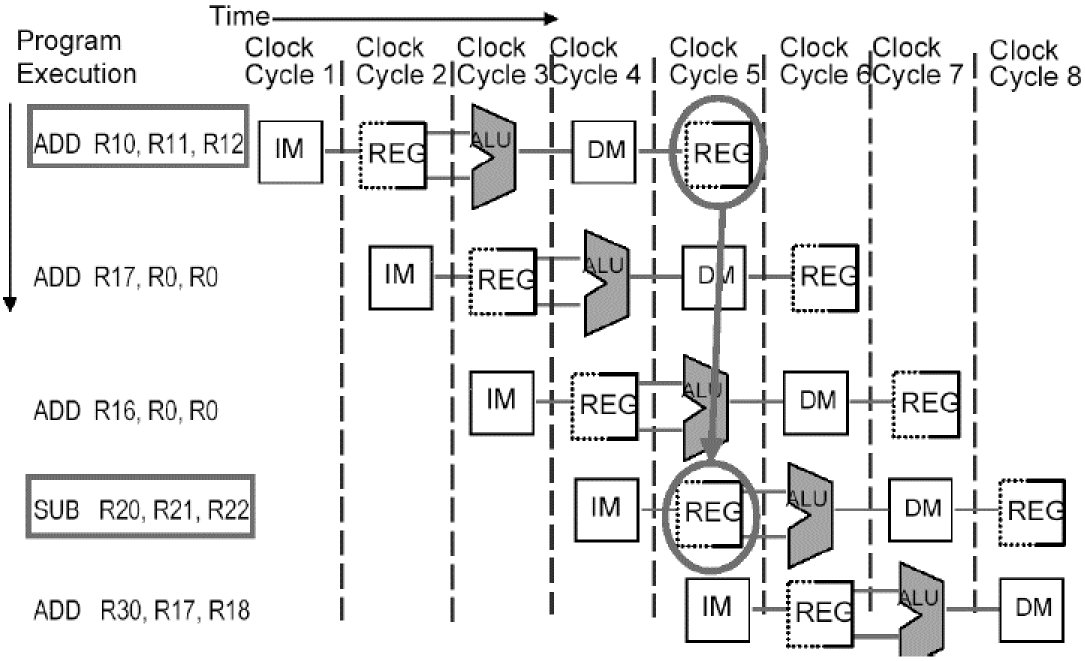


Figure 1.3: hazard example

1. **Structural hazards**
   Though the MIPS instruction set was designed to be pipelined, it does not solve the structural limitation of the design. If only one memory is used it will be impossible to solve a store or load instruction without stalling the pipeline. This is because a new instruction is fetched from the memory every clock cycle, and it is not possible to access the memory twice during a clock cycle.

2. **Control hazards**
   Control hazards arise from the need to decide based on the results of one instruction while others are executing. This applies to the branch instruction. If it is not possible to solve the branch in the second stage, we will need to stall the pipeline. One solution to this problem is branch prediction, where one guesses, based on statistics, if a branch is to be taken or not. In the MIPS architecture delayed decision was

used . A delayed branch always executes the next sequential instruction following the branch instruction. This is normally solved by the assembler, which will rearrange the code and insert an instruction that is not affected by the branch. The assembler made for this project does not support code reordering, it must be done manually.

3. **Data Hazards**
   If an instruction depends on the result of a previous instruction still in the pipeline, we will have a data hazard. These dependencies are too common to expect the compilers to be able avoid this problem. A solution is to get the result from the pipeline before it reaches the write back stage. This solution is called forwarding or bypassing.

Dealing with the hazards

1. Using two memories solves the structural hazard. One for instructions and one for data.
   Normally only one memory is used in a system. In that case separate instruction and data caches can be used to solve the structural hazard. In this project only one memory was available and because no caches were implemented, the processor is stalled for each load and store instruction.

2. Using delayed decision solves the control hazards.

3. Forwarding solves the data hazards. Still it will not be possible to combine a load instruction and an instruction that reads its result. This is due to the pipeline design and a hazard detection unit will stall the pipeline one cycle.

Modules Description

3.1 Full Adder

File name: full_adder.vhd

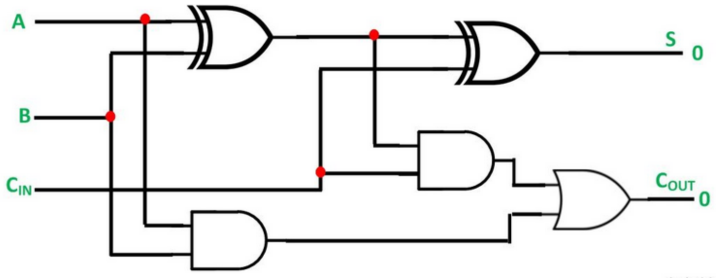


Figure 3.1 : Graphical description for : Full Adder

Port name	direction	type & size	functionality
a	*in*	*std_logic*	*bit A*
b	*in*	*std_logic*	*bit B*
Cin	*in*	*std_logic*	*bit Cin*
s	*out*	*std_logic*	*bit S*
Cout	*out*	*std_logic*	*bit Cout*

Table 3.1.1: Port Table for : Full Adder

Description: 2-bit full adder with carry in\out. Designed as a component for the Adder **structural** architecture entity..

Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.1.2 : Logic Table for : Full Adder

3.2 Adder

File name: add.vhd

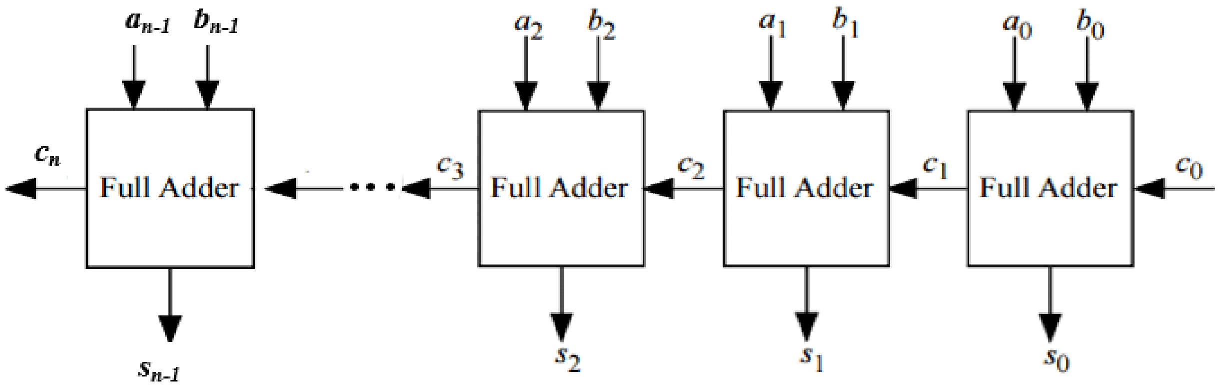


Figure 3.2 : Graphical description for : Adder n-bit

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
Cin	*in*	*std_logic (1 bit)*	*Carry bit in*
A	*in*	*signed (N bits)*	*Number A*
B	*in*	*signed (N bits)*	*Number B*
Sum	*out*	*signed (N bits)*	*Sum = A+B*
Cout	*out*	*std_logic (1 bit)*	*Carry bit out*

Table 3.2: Port Table for: Adder

Description: n-bit Adder designed using 2-bit full-adders with carry in\out (for-generate). The design is with structural architecture as an aid component for ADD/SUB entities.

3.3 XOR GATE

File name: xor.vhd

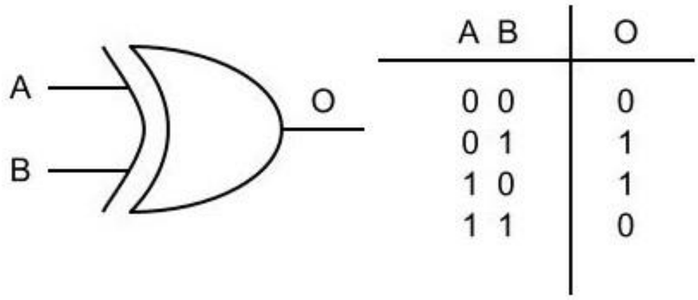


Figure 3.3: Graphical description for: XOR gate

Port name	direction	type & size	functionality
A	*in*	*Std_logic, 1 bit*	*Bit A*
B	*in*	*Std_logic 1 bit*	*Bit B*
C	*out*	*Std_logic 1 bit*	*C = A XOR B*

Table 3.3: Port Table for: XOR gate

Description: xor gate of 2 inputs (1 bit each) that generate 1-bit output. Design with behavioral architecture as an aid component for ADD/SUB entities.

3.4 ADD/SUB operations

File name: ADD_SUB.vhd

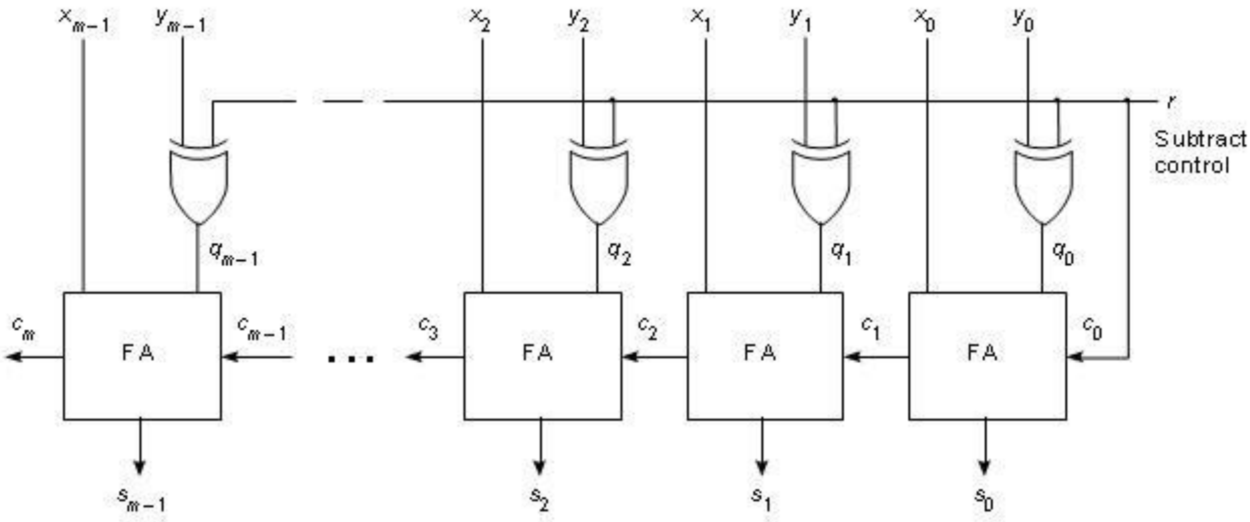


Figure 3.4: Graphical description for: Adder/Subtractor n-bit

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
addORsub	*In*	*std_logic (1 bit)*	*'1' for sub, '0' for add*
A	*in*	*signed (N bits)*	*Number A*
B	*in*	*signed (N bits)*	*Number B*
Sum	*out*	*signed (N bits)*	*Sum = A+B*

Table 3.4: Port Table for: Adder/Subtractor n-bit

Description: n-bit Adder/Subtractor designed using 2-bit full-adders with carry in\out (ADD component). The design is with structural architecture.

3.5 MAX/MIN operation

File name: MAX_MIN.vhd

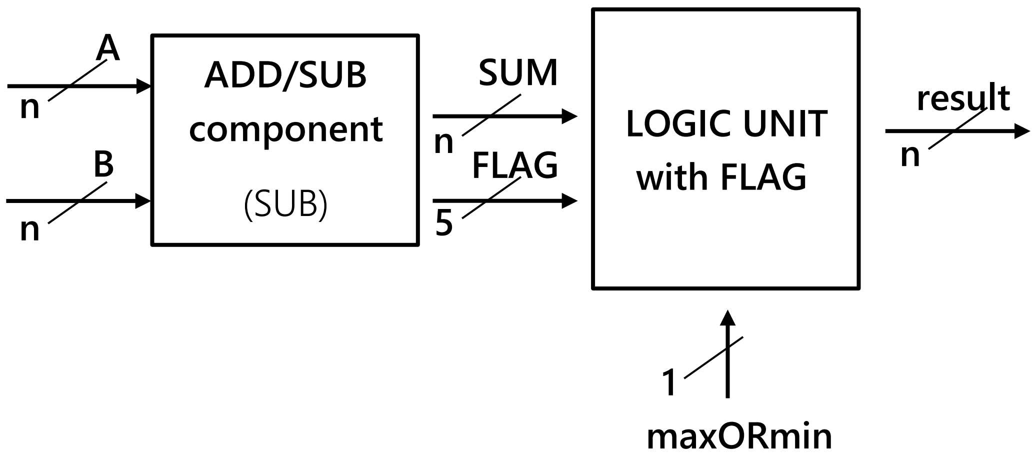


Figure 3.5: Graphical description for: MAX/MIN operation

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
maxORmin	*in*	*std_logic (1 bit)*	*Operation selector bit*
A	*in*	*signed (N bits)*	*Number A*
B	*in*	*signed (N bits)*	*Number B*
result	*out*	*signed (N bits)*	*C = MAX/MIN(A,B)*

Table 3.5: Port Table for: MAX/MIN operation

Description: max/min operation. Input 2 Numbers (N bits each) and 1-bit operation selector for max or min operation. The design is with behavioral architecture with the aid component ADD_SUB. After using the SUB operation, we can know the order between A and B from calculate the FLAGS.

3.6 Shift Left/Right (1-bit shifter)

File name: shift_Nbits.vhd

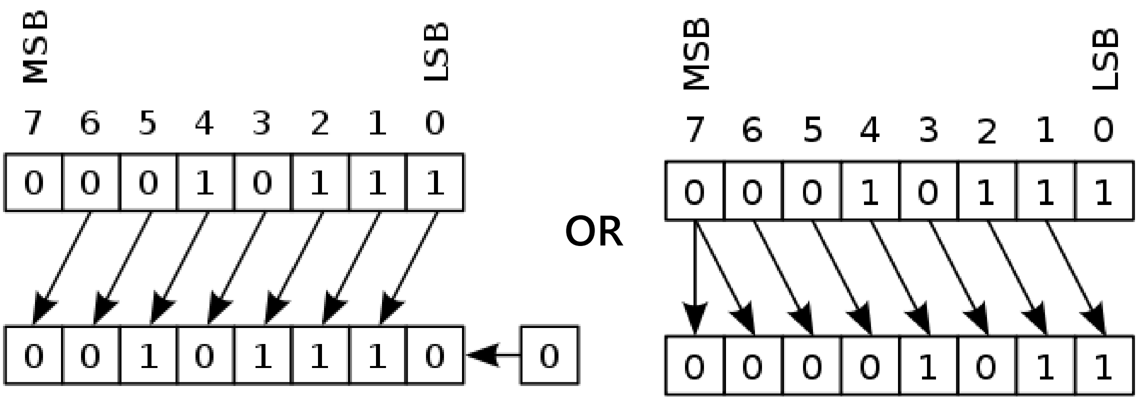


Figure 3.6: Graphical description for: Shift Left/Right (1-bit shifter)

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
dir	*In*	*std_logic (1 bit)*	*'0' left '1' right*
enable	*In*	*std_logic (1 bit)*	*'0' – Aout=A, '1' – Aout is the shifted number*
A	*in*	*signed (N bits)*	*Number A*
Aout	*out*	*signed (N bits)*	*Shifted Number A*

Table 3.6: Port Table for: Shift Left/Right (1-bit shifter)

Description: 1-bit shifter (1 bit to the left/right) that generate N bit output. The design is with **structural architecture** as an aid component for the TOP design shift unit. If enable = '0' then the output will be the input A. The shift unit will generate 64 shifters and will passing enables according to the required number B.

3.7 Shift Unit (TOP design)

File name: shift_unit.vhd

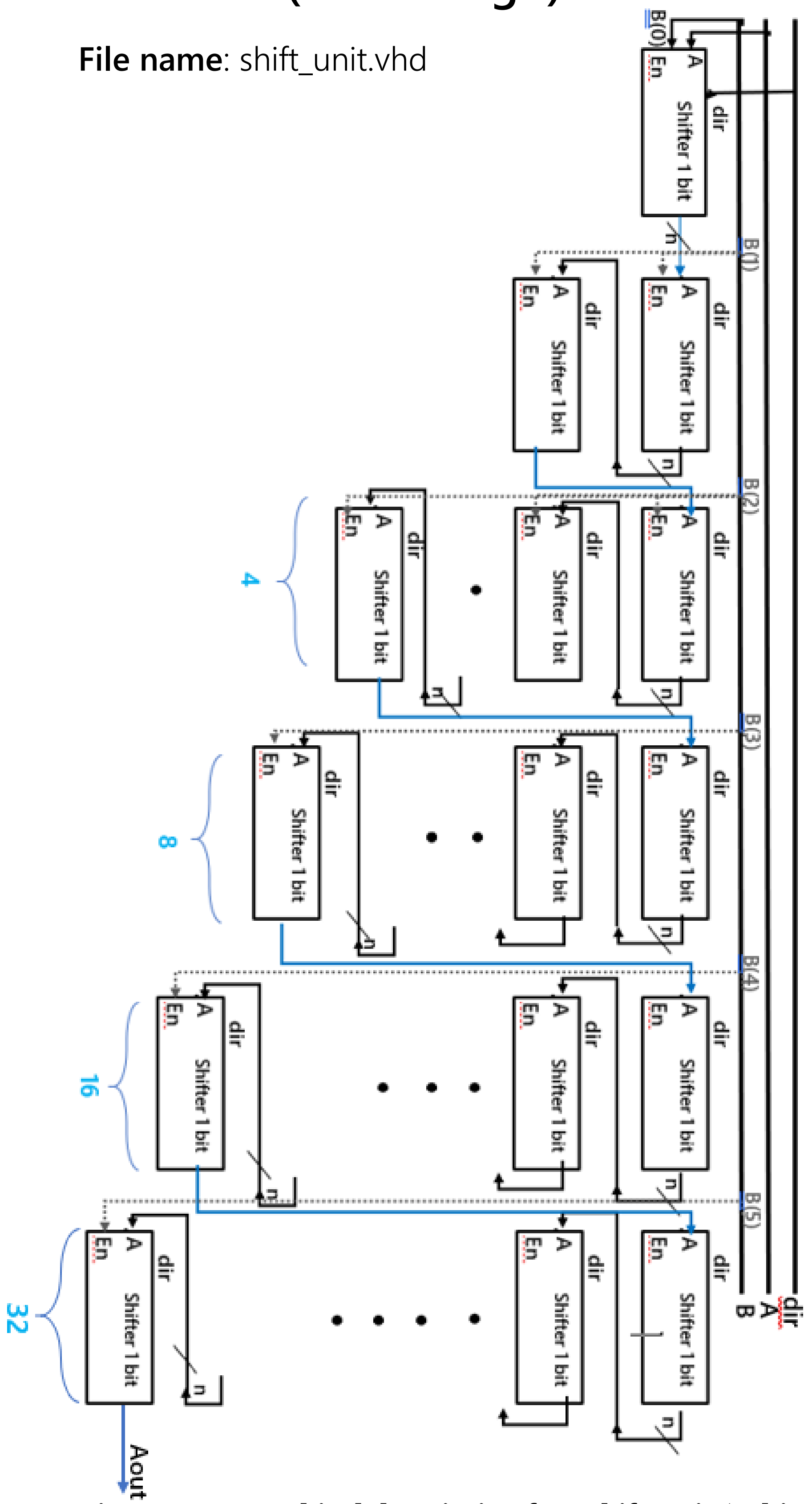


Figure 3.7: Graphical description for: Shift Unit (B-bits shifter)

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
dir	*In*	*std_logic (1 bit)*	*'0' left '1' right*
A	*in*	*signed (N bits)*	*Number A*
B	*in*	*signed (N bits)*	*Number B*
result	*out*	*signed (N bits)*	*Result = A >> B*

Table 3.7: Port Table for: Shift Unit (B-bits shifter)

Description: |B|-bits shifter (to the right/left) that generate N bit output with **Barrel** logic. The design is with **structural architecture** with the aid of the structural component shift_Nbits as required.

3.8 Mux 2N-N bit

File name: MUX_Nbits.vhd

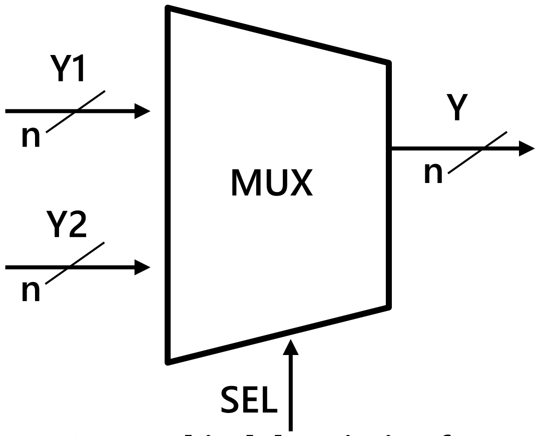


Figure 3.8: Graphical description for: Mux 2N-N bit

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
SEL	*In*	*std_logic(1 bit)*	*Selection bit*
Y1	*in*	*signed (N bit)*	
Y2	*In*	*signed (N bit)*	
Y	*out*	*signed (N bit)*	*Y1 \ Y2, according to SEL*

Table 3.8: Port Table for: Mux 2N-N bit

Description: 2N-N mux with behavioral architecture. Designed as an aid component for general use.

3.9 MUL operation

File name: MUL.vhd

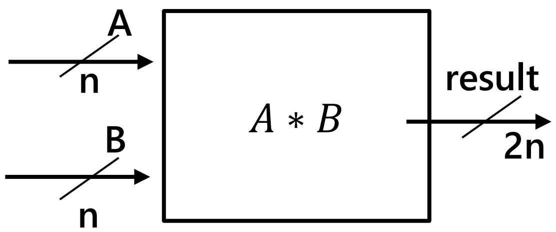


Figure 3.9: Graphical description for: MUL operation

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
A	*in*	*signed (N bits)*	*Number A*
B	*in*	*signed (N bits)*	*Number B*
result	*out*	*signed (2*N bits)*	*A*B*

Table 3.9: Port Table for: MUL operation

Description: MUL operation. Input 2 Numbers (N bits each). The design is with **behavioral architecture**. Support Signed numbers.

3.10 basic d-flip-flop (dff)

File name: dff_1bit.vhd

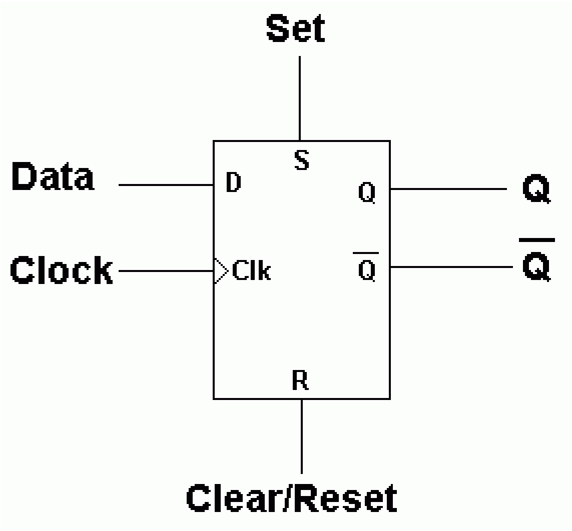


Figure 3.10: Graphical description for: 1-bit dff entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
en	*In*	*std_logic (1 bit)*	*Enable bit*
clk	*In*	*std_logic (1 bit)*	*clock bit*
rst	*In*	*std_logic (1 bit)*	*reset bit*
d	*in*	*std_logic (1 bit)*	*bit d*
q	*in*	*std_logic (1 bit)*	*bit q*

Table 3.10: Port Table for: 1-bit dff entity

Description: 1-bit dff. Designed with **structural architecture** as an aid component for N-bits dff.

3.11 N dff's

File name: N_dff.vhd

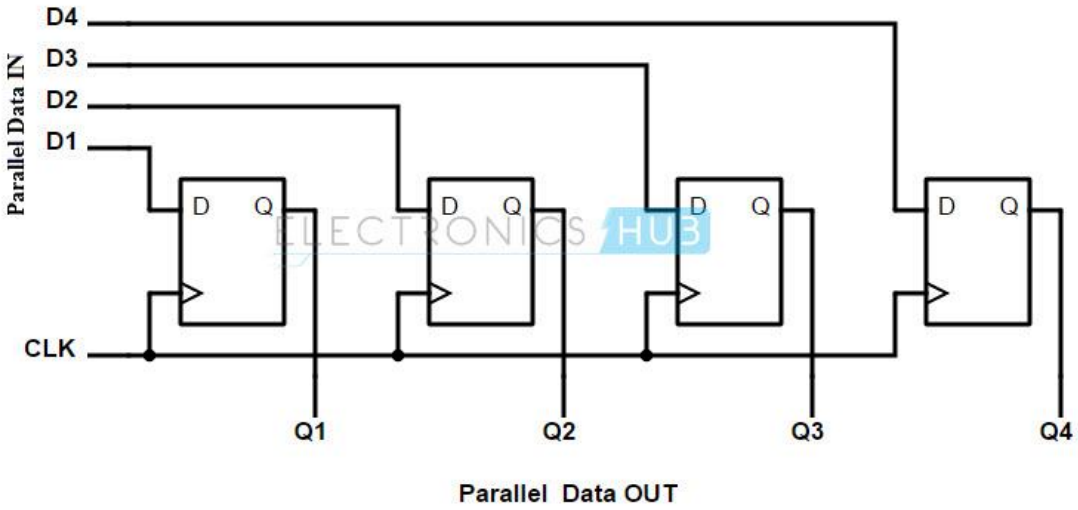


Figure 3.11: Graphical description for:(example N=4) N-bit dff entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
clk	*In*	*std_logic (1 bit)*	*clock bit*
enable	*In*	*std_logic (1 bit)*	*Enable bit*
rst	*In*	*std_logic (1 bit)*	*reset bit*
d	*in*	*std_logic (N bits)*	*Number D*
q	*in*	*std_logic (N bits)*	*Number Q*

Table 3.11: Port Table for: N-bit dff entity

Description: N-bit dff (which is really N 1-bit dffs). Designed with **structural architecture** as an register. Component: 1bit_dff.

3.12 Swap Nbits numbers

File name: Swap.vhd

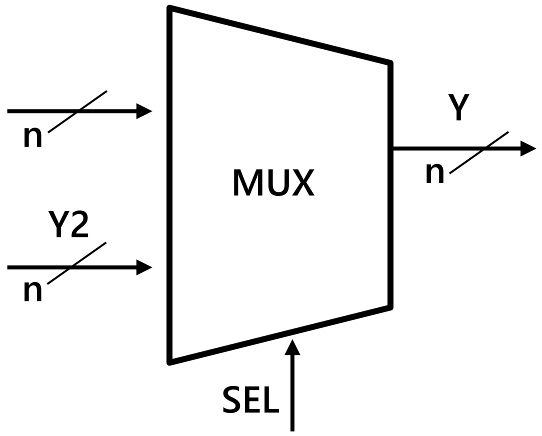


Figure 3.12: Graphical description for Swap Nbits numbers entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
SEL	*In*	*std_logic(1 bit)*	*Selection bit*
Y1	*in*	*signed (N bit)*	
Y2	*In*	*signed (N bit)*	
Y	*out*	*signed (N bit)*	*Y1 \ Y2, according to SEL*

Table 3.12: Port Table for: Swap Nbits numbers entity.

Description: Swap between 2 Nbits numbers. Designed with **structural architecture** as an aid component for FPU unit.

3.13 ADD\SUB Floating Point numbers

File name: ADD_SUB_FPU.vhd

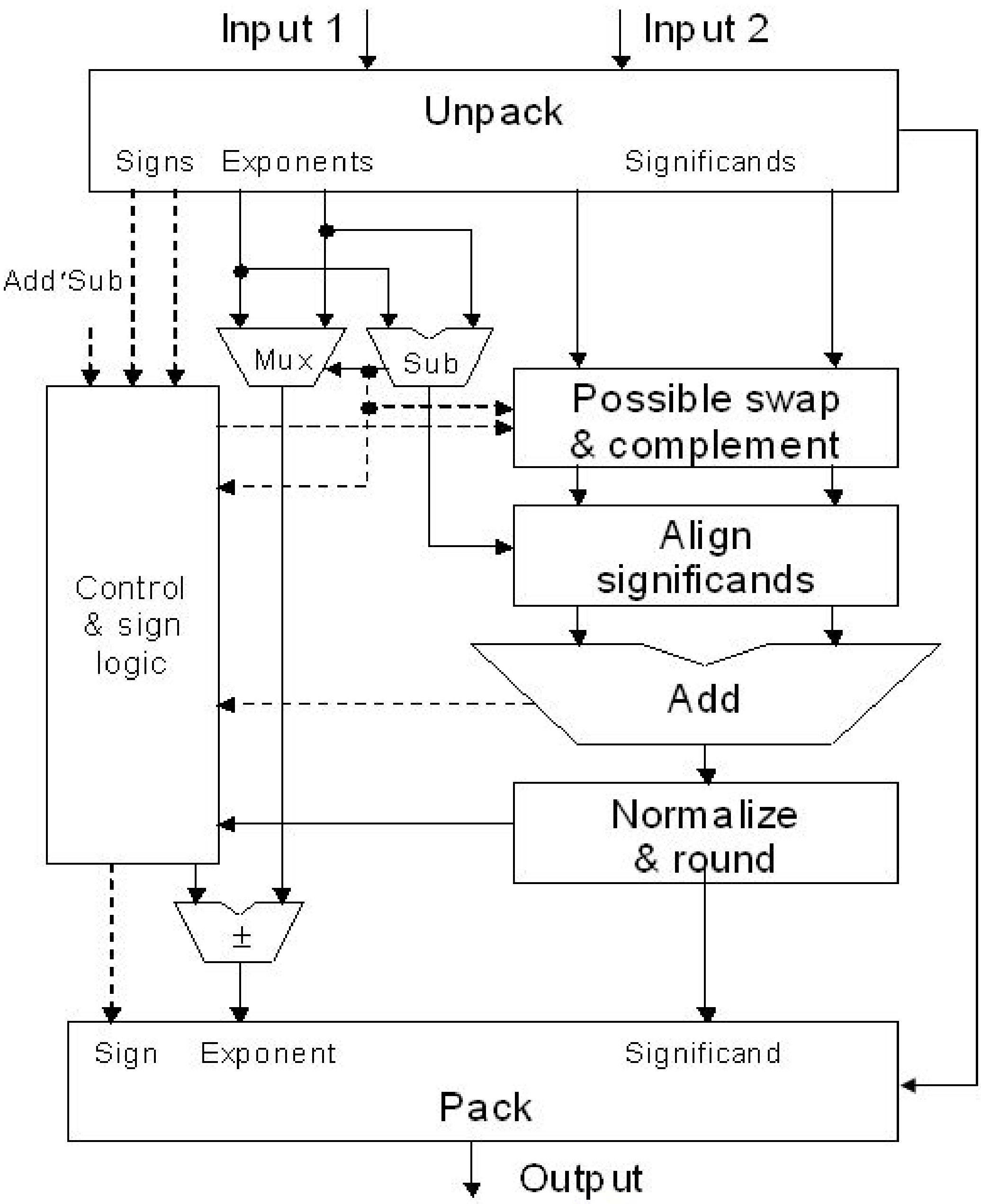


Figure 3.13: Graphical description for ADD\SUB Floating Point numbers entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
OPP	*In*	*std_logic (1 bit)*	*'1' for sub, '0' for add*
A	*in*	*signed (N bits)*	*ieee A*
B	*in*	*signed (N bits)*	*ieee B*
Sum	*out*	*signed (N bits)*	*Ieee C = A+B*

Table 3.13: Port Table for: N-bit dff entity

Description: Add\Sub floating point numbers. Designed with **structural architecture** as an aid component for FPU unit.
Components: ADD_SUB, Swap, shift_unit, MUX_Nbits, MAX_MIN.

3.14 MUL Floating Point numbers

File name: MUL_FPUvhd

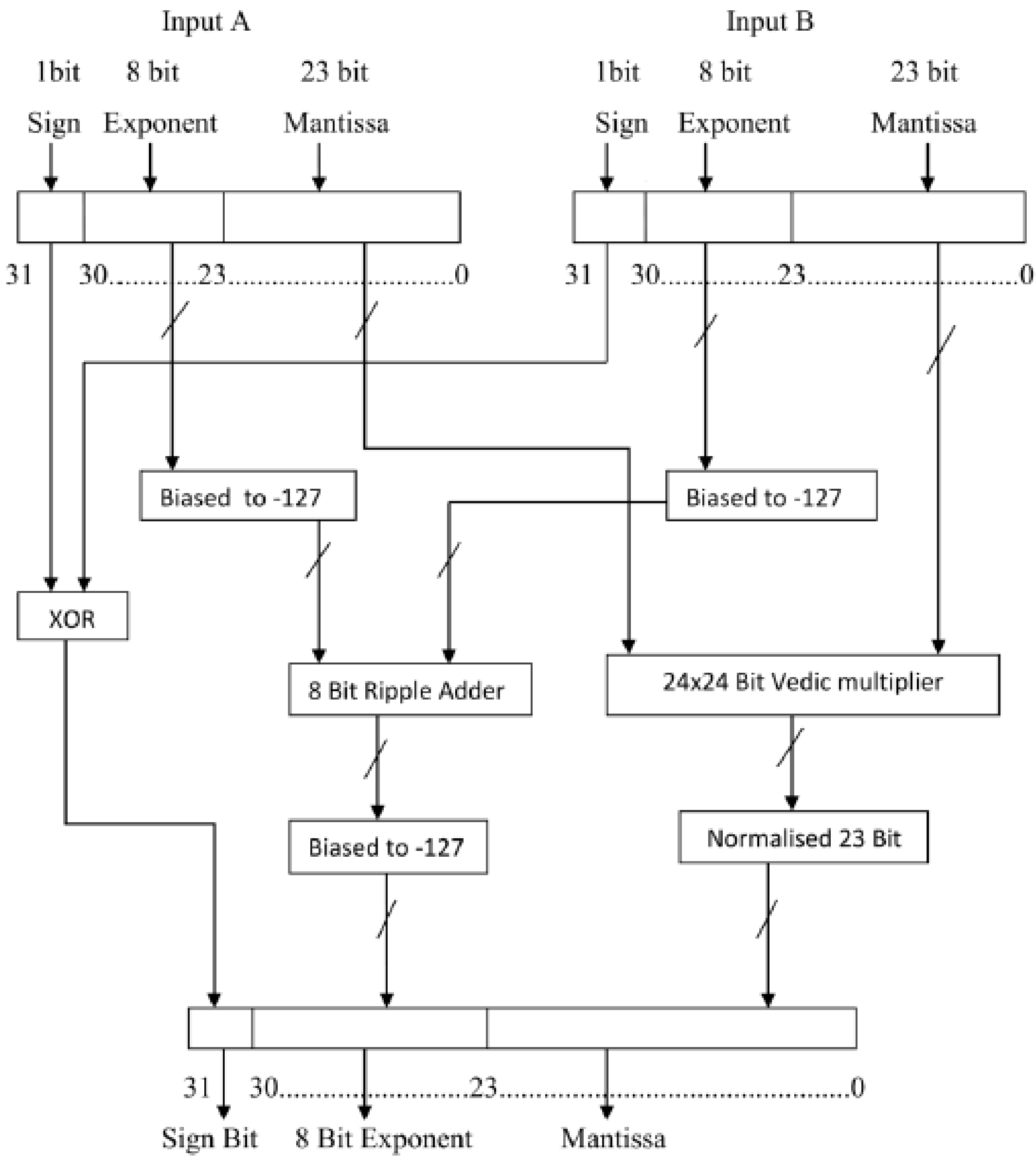


Figure 3.14: Graphical description for: MUL Floating Point numbers entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
A	*in*	*signed (N bits)*	*ieee A*
B	*in*	*signed (N bits)*	*ieee B*
Sum	*out*	*signed (N bits)*	*Ieee C = A*B*

Table 3.14: Port Table for: MUL Floating Point numbers entity.

Description: Multiply floating-point numbers. Designed with **structural architecture** as an aid component for FPU unit.
Components: ADD_SUB, MUL, Swap, LeadingZeroes_counter.

3.15 FPU output selector

File name: FPU_selector.vhd

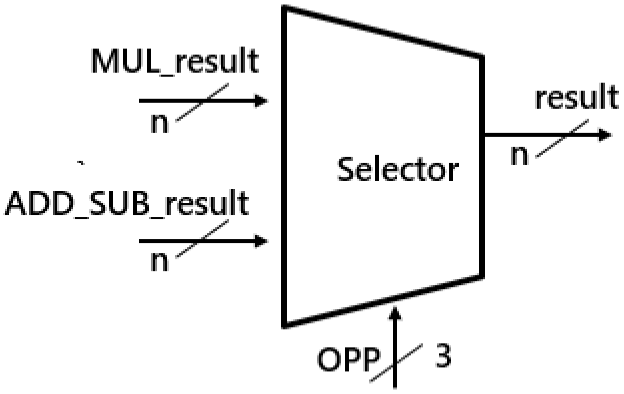


Figure 3.15: Graphical description for: FPU output selector entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
OPP	*in*	*Std_logic_vector (3 bits)*	*OPP code LSBS*
MUL_result	*in*	*signed (N bits)*	*ieee MUL*
ADD_SUB_result	*in*	*signed (N bits)*	*Ieee ADD\SUB*
result	*in*	*signed (N bits)*	*ieee select*

Table 3.15: Port Table for: N-bit dff entity

Description: FPU output selector. Designed with **structural architecture** as an aid component for FPU top design.

3.16 FPU top design unit

File name: FPU_Unit.vhd

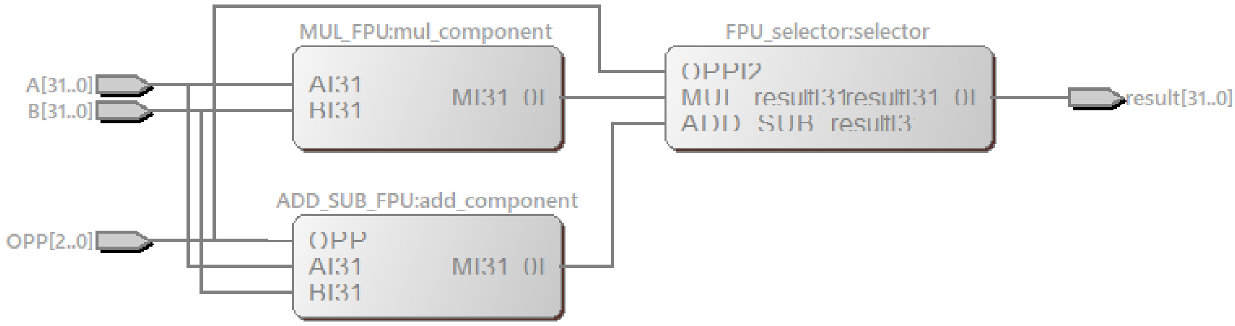


Figure 3.16: Graphical description for FPU top design unit entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
OPP	*in*	*Std_logic_vector (3 bits)*	*OPP code LSBS*
A	*in*	*signed (N bits)*	*ieee A*
B	*in*	*signed (N bits)*	*Ieee B*
result	*in*	*signed (N bits)*	*ieee result*

Table 3.16: Port Table for: FPU top design unit entity.

Description: FPU top design unit Designed with **structural architecture**. **Components**: ADD_SUB_FPU, MUL_FPU, FPU_Selector.

3.17 LeadingZeros counter

File name: LeadingZeros_counter.vhd

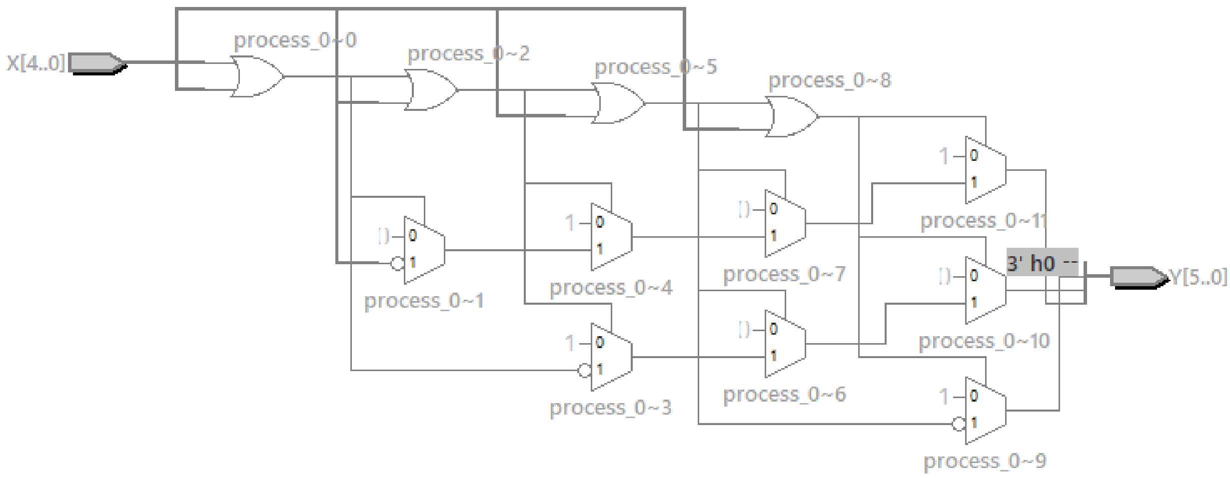


Figure 3.17: Graphical description for LeadingZeros counter entity.

Port name	direction	type & size	functionality
N	*in*	*generic integer*	*How many bits*
X	*in*	*signed (N bits)*	*Number X*
Y	*out*	*std_logic (6 bits)*	*Y'leading zeros*

Table 3.17: Port Table for: LeadingZeros counter entity.

Description: Leading Zeros counter. Designed with **structural architecture** as an aid component for FPU commands.

3.18 MIPS TOP design

File name: MIPS.vhd

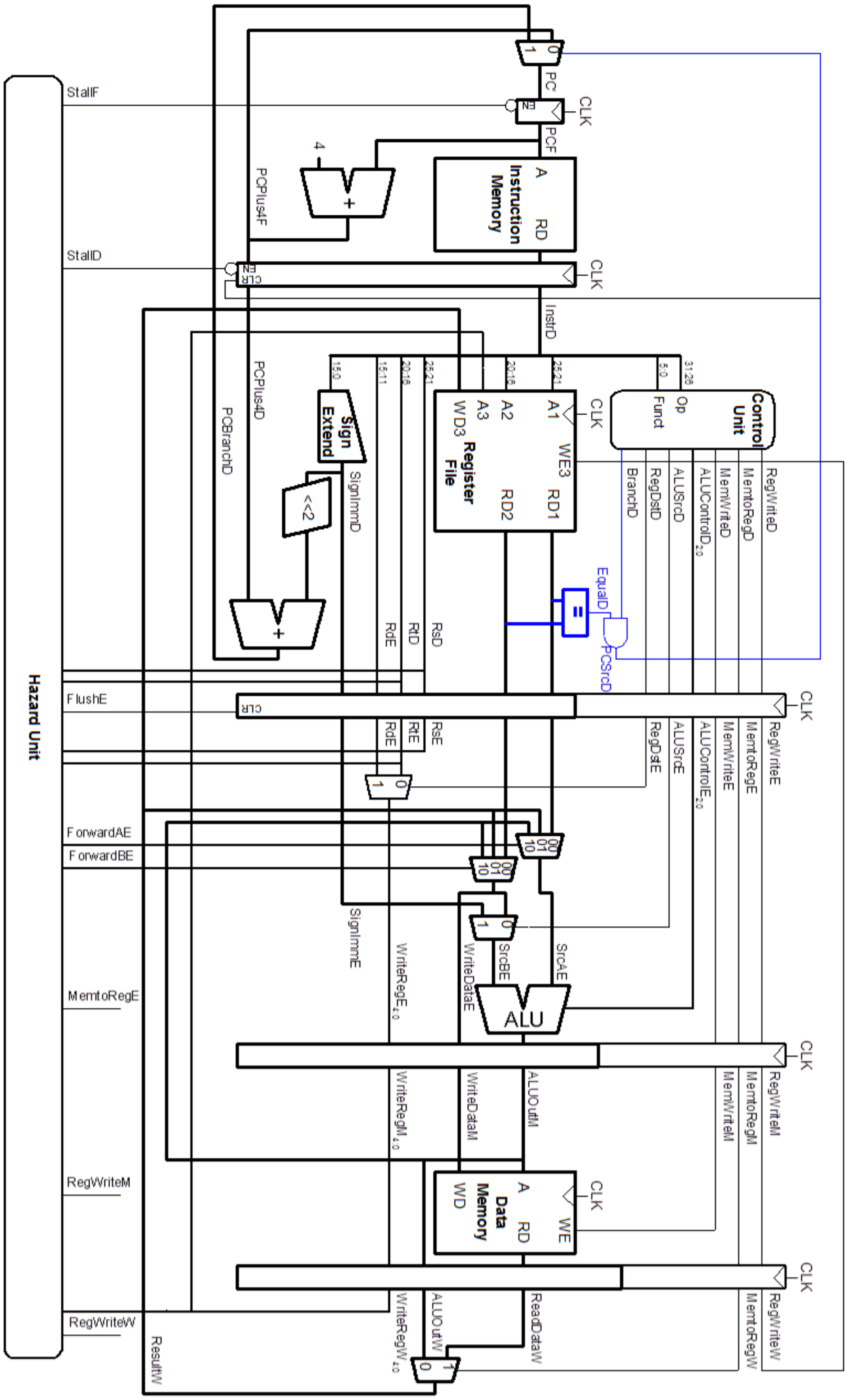


Figure 3.18: Graphical description for pipelined MIPS entity.

Port name	direction	type & size	functionality
Reset	*in*	*Std_logic*	*Reset bit*
Clock	*in*	*Std_logic*	*Clock bit*
PC	*Out*	*std_logic (10 bits)*	*Program counter*
ALU_result_out	*out*	*std_logic (32 bits)*	*ALU result*
Read_data_1_out	*out*	*std_logic (32 bits)*	*Instruction operand 1*
Read_data_2_out	*out*	*std_logic (32 bits)*	*Instruction operand 1*
Write_data_out	*out*	*std_logic (32 bits)*	*Write data to memory*
Instruction_out	*out*	*std_logic (32 bits)*	*Current instruction*
Branch_out	*out*	*Std_logic*	*Branch indicator*
Zero_out	*out*	*Std_logic*	*Zero indicator*
Memwrite_out	*out*	*Std_logic*	*Memory write indicator*
Reqwrite_out	*out*	*Std_logic*	*Register write indicator*

Table 3.18: Port Table for: pipelined MIPS entity

Description: The CPU is using a PIPELINED architecture and capable of performing instructions from MIPS instruction set. Designed with **structural architecture** as top design.

3.19 Hex to 7-Segment

File name: 7-Segment_8_bit.vhd

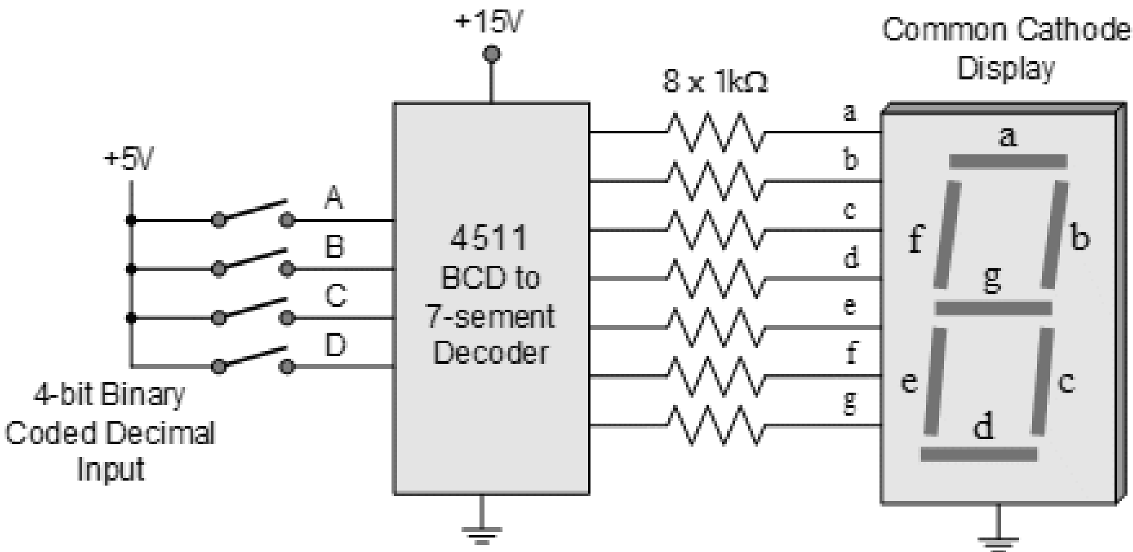


Figure 3.19: Graphical description for: Hex to 7-Segment entity.

Port name	direction	type & size	functionality
q	*in*	*std_logic (8 bits)*	*Number Q*
Segment1	*Out*	*std_logic (7 bits)*	*Segment1*
Segment2	*out*	*std_logic (7 bits)*	*Segment2*

Table 3.19: Port Table for: Hex to 7-Segment entity.

Description: 8 bit (2 hex number) to 7-segment display on the FPGA. Designed with **behavioral architecture** as an aid component for FPGA top design entity register.

Analyzing

4.1 MIPS– overall system

File name: MIPS.vhd

Full Image in DOC/rtl_mips.png

Flow Summary	
Flow Status	Successful - Wed May 30 10:40:25 2018
Quartus II 32-bit Version	12.1 Build 177 11/07/2012 SJ Web Edition
Revision Name	MIPS
Top-level Entity Name	MIPS
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	3,028 / 18,752 (16 %)
Total combinational functions	3,028 / 18,752 (16 %)
Dedicated logic registers	1,289 / 18,752 (7 %)
Total registers	1289
Total pins	176 / 315 (56 %)
Total virtual pins	0
Total memory bits	16,384 / 239,616 (7 %)
Embedded Multiplier 9-bit elements	7 / 52 (13 %)
Total PLLs	0 / 4 (0 %)

Figures 4.1.2: Flow Summary for: overall system entity.

	Analysis & Synthesis Resource Usage Summary	
	Resource	Usage
1	Estimated Total logic elements	3,988
2		
3	Total combinational functions	3028
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	2104
2	-- 3 input functions	723
3	-- <=2 input functions	201
5		
6	Logic elements by mode	
1	-- normal mode	2859
2	-- arithmetic mode	169
7		
8	Total registers	1289
1	-- Dedicated logic registers	1289
2	-- I/O registers	0
9		
10	I/O pins	176
11	Total memory bits	16384
12	Embedded Multiplier 9-bit elements	7
13	Maximum fan-out	1355
14	Total fan-out	15454

Figures 4.1.2: Logic usage for: overall system entity.

Entity	Logic Cells	Dedicated Logic Registers	DSP Elements	DSP 9x9	DSP 18x18	Pins	LUT-Only LCs	LUT-
Cyclone II: EP2C20F484C7								
MIPS	3028 (35)	1289 (0)	7	1	3	176	1739 (35)	1289
N_dff:ALU_result_C	223 (0)	32 (0)	0	0	0	0	191 (0)	32 ((
N_dff:ALU_result_D	32 (0)	32 (0)	0	0	0	0	0 (0)	32 ((
N_dff:ALUop_control_B	2 (0)	2 (0)	0	0	0	0	0 (0)	2 (0)
dff_1bit:ALUSrc_control_B	1 (1)	1 (1)	0	0	0	0	0 (0)	1 (1)
control:CTL	12 (12)	0 (0)	0	0	0	0	10 (10)	2 (2)
Execute:EXE	794 (124)	0 (0)	7	1	3	0	791 (121)	3 (3)
Execute_branch:EXE_brn	100 (100)	0 (0)	0	0	0	0	92 (92)	8 (8)
HAZARD:HAZ	58 (58)	0 (0)	0	0	0	0	58 (58)	0 (0)
Idecode:ID	1546 (1546)	992 (992)	0	0	0	0	554 (554)	992
Ifetch:IFE	16 (16)	8 (8)	0	0	0	0	8 (8)	8 (8)
N_dff:Instruction_A	32 (0)	32 (0)	0	0	0	0	0 (0)	32 ((
N_dff:Instruction_B	16 (0)	16 (0)	0	0	0	0	0 (0)	16 (0
N_dff:Instruction_C	10 (0)	10 (0)	0	0	0	0	0 (0)	10 (0
N_dff:Instruction_D	10 (0)	10 (0)	0	0	0	0	0 (0)	10 (0
dmemory:MEM	0 (0)	0 (0)	0	0	0	0	0 (0)	0 (0)

Figures 4.1.2: Logic usage for: each main entity.

Analyze: We zoom in on the main components. We can see that the MUL_FPU required more logic units than the ADD_SUB_FPU, because the MUL_FPU required the MUL hardware. The FPU_UNIT using most of the components of the ALU, such as shift_unit, MUL, ADD\SUB – therefor it required more logic units than the Arithmetic_Unit.

Figure 4.1.3: Critical path for: overall system entity, full-png in DOC.

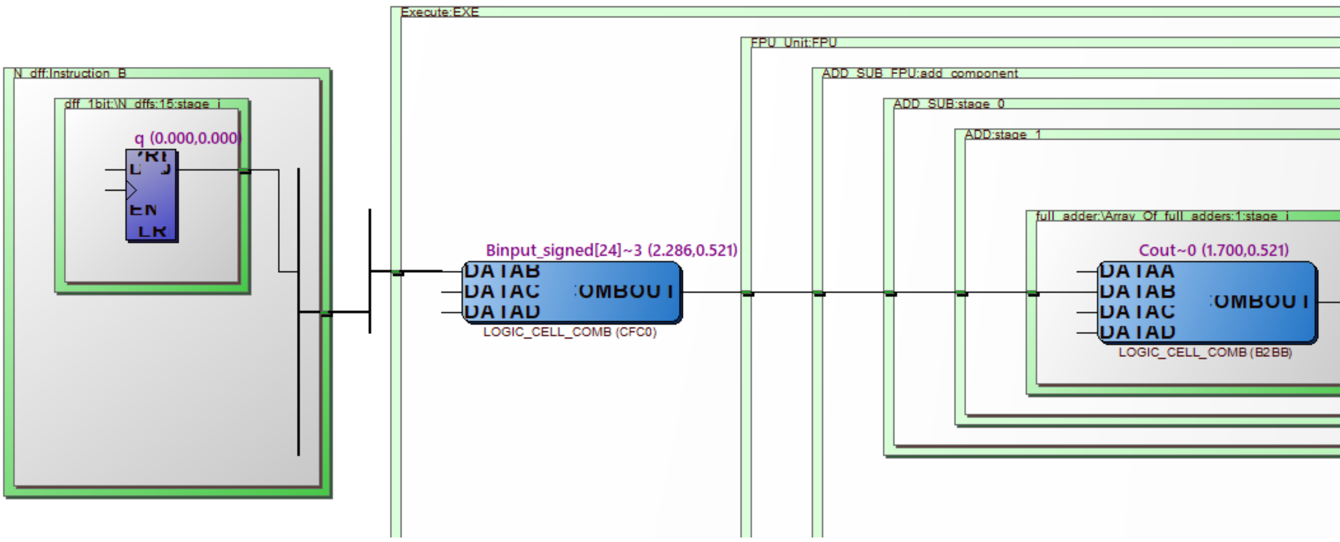


Figure 4.1.5: Critical path for: overall system entity, full-png in DOC.

Analyze: $Registers \rightarrow FPGA\ TOP \dashrightarrow Insturction\ Register\ B \rightarrow Execute \rightarrow FPU\ Unit \rightarrow ADD\ SUB\ FPU\ component \rightarrow ADD\ SUB\ component \rightarrow FullAdders \rightarrow LeadingZeroes\ component \rightarrow Shift\ Unit \rightarrow Register\ result(Write\ Back)$

Frequency limiting operation: FPU commands - using the **Shift Unit, FPU UNIT.** The shift unit using Ndff to shift the number, which require 1 clk per shifter (1 register).

Propose solution for CPU frequency improvements in two cases (current ALU):

1. The problematic operation is commonly used in software :
   ⇨ Design shift unit with dedicated hardware (with separate clock), and refactor the output selector not to wait for the shift result.

2. The problematic operation is almost unused:
   ⇨ Then we can refactor the system with enable to this specific hardware. The hardware will not be in use until the uncommon operation will be required.

Maximal operating clock:

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	17.52 MHz	17.52 MHz	clock	

Figure 4.1.4: Fmax Summary

Cases checked : see pages 6-9.

Conclusions and future work : By using VHDL in digital design it is possible to use a high level of abstraction in the design. This lets you put more effort on the functionality of the circuit. Another aspect of VHDL is that the design will be self-documented. we have for a long time been interested in low-level programming of microprocessors. During the design of the MIPS I learned a lot on processor design that will be useful in the future when optimizing time-critical programs.

4.2 FPU unit

File name: FPU_unit.vhd

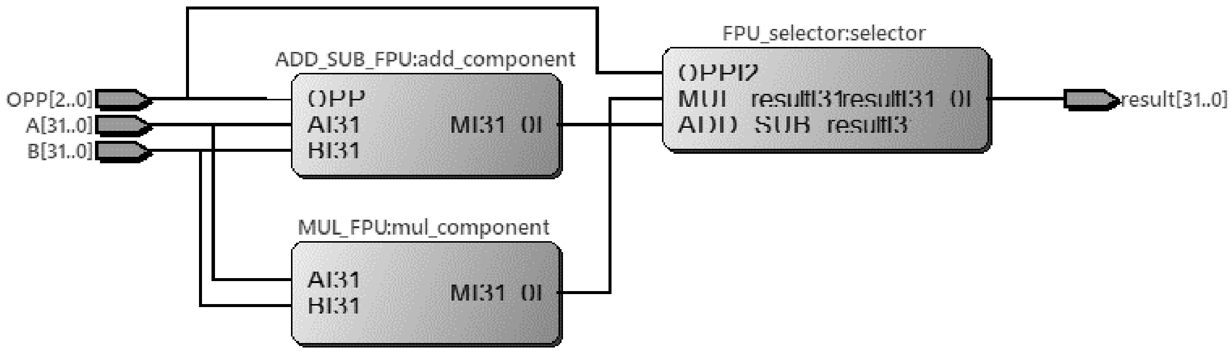


Figure 4.2.1: RTL Viewer for: FPU top design entity.

4.4 Floating Point Adder

File name: ADD_SUB_FPU.vhd

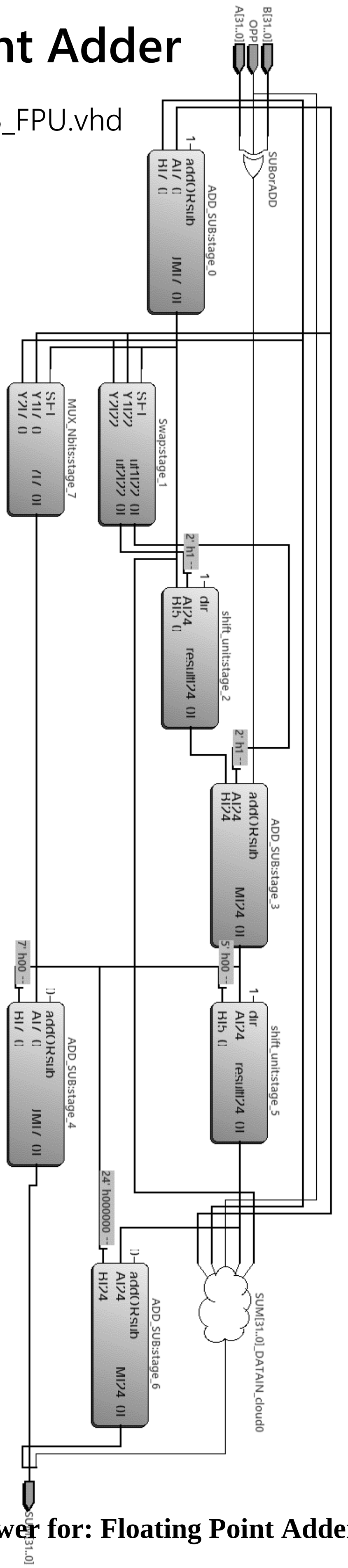


Figure 4.4.1: RTL Viewer for: Floating Point Adder entity.

4.5 Floating Point Multiplier

File name: MUL_FPU.vhd

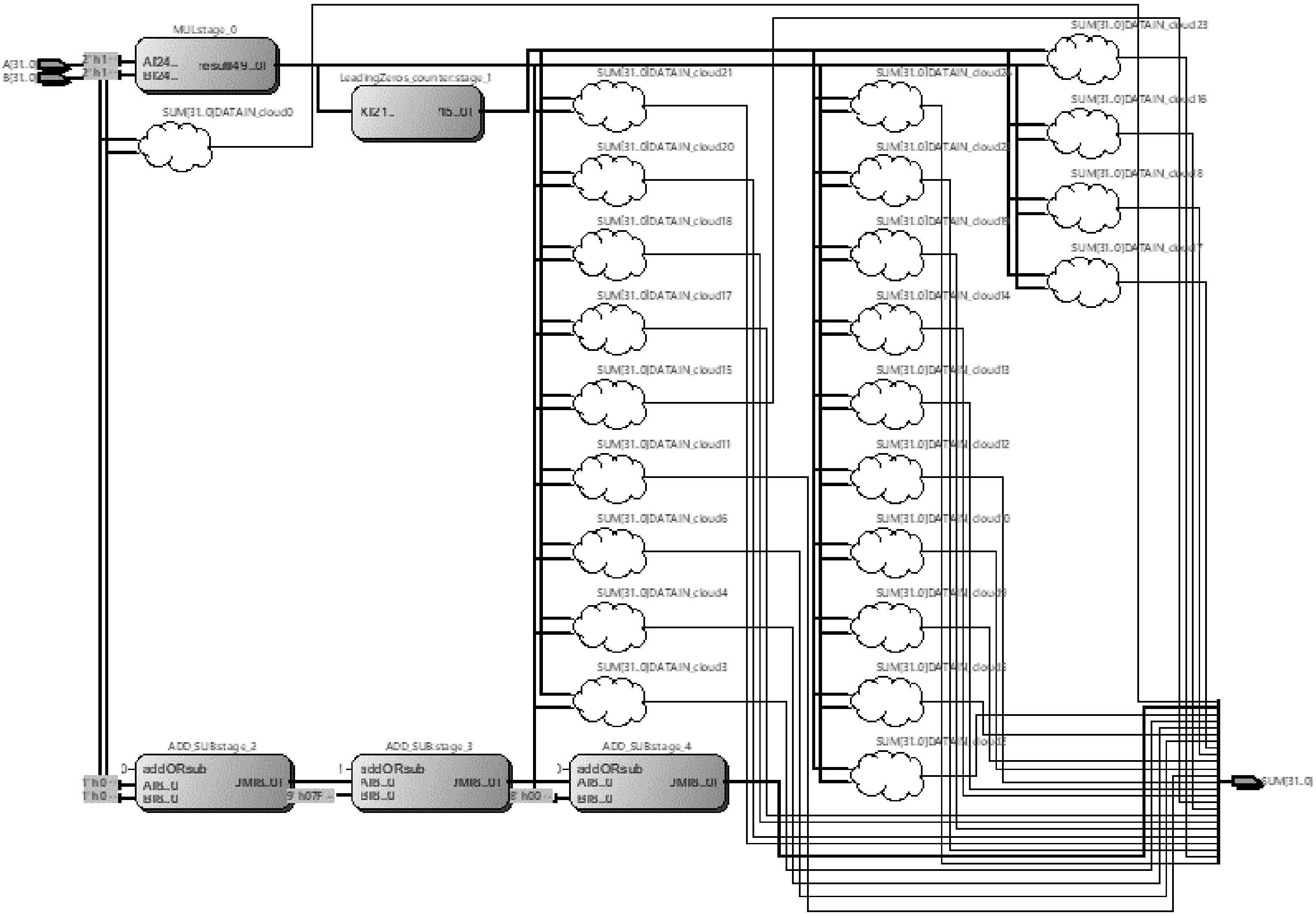


Figure 4.5.1: RTL Viewer for: Floating Point Multiplier entity.

4.6 Fetch

File name: IFETCH.vhd

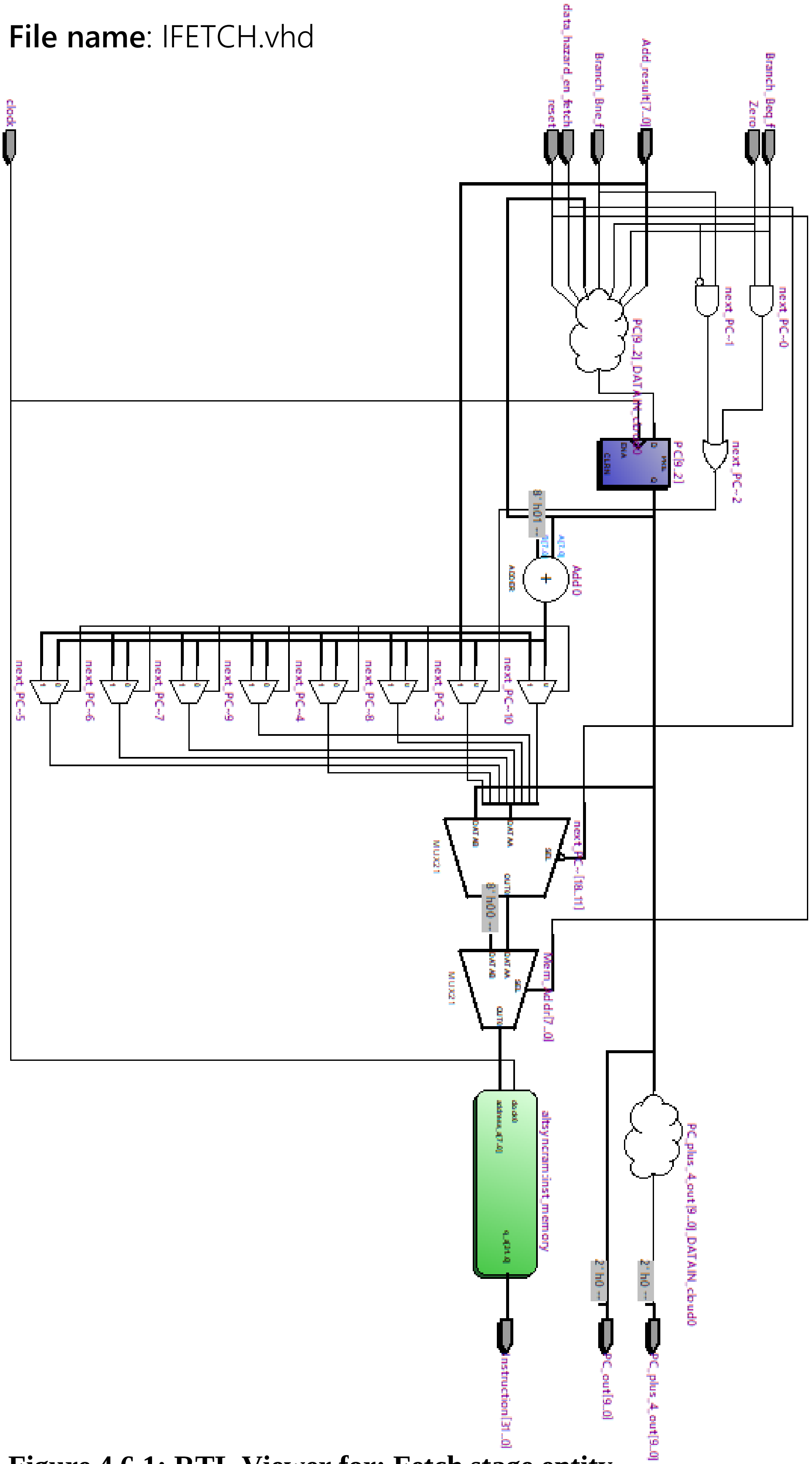


Figure 4.6.1: RTL Viewer for: Fetch stage entity.

Description: The first stage in the pipeline is the Instruction Fetch. Instructions will be fetched from the memory and the Instruction Pointer (IP) will be updated.

4.7 Decode

File name: IDECODE.vhd

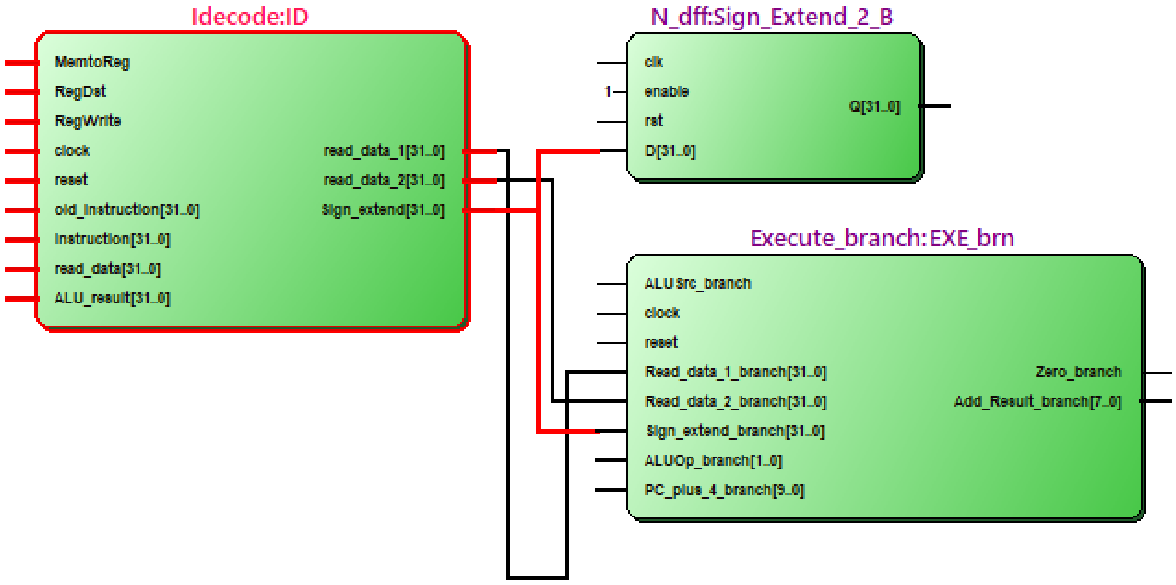


Figure 4.7.1: RTL Viewer for: Decode stage entity.

Description: The Instruction Decode stage is the second in the pipeline. Branch targets will be calculated here and the Register File, the dual-port memory containing the register values, resides in this stage. The forwarding units, solving the data hazards in the pipeline, reside here. Their function is to detect if the register to be fetched in this stage is written to in a later stage. In that case the data is forward to this stage and the data hazard is solved.

4.8 Execute

File name: EXECUTE.vhd

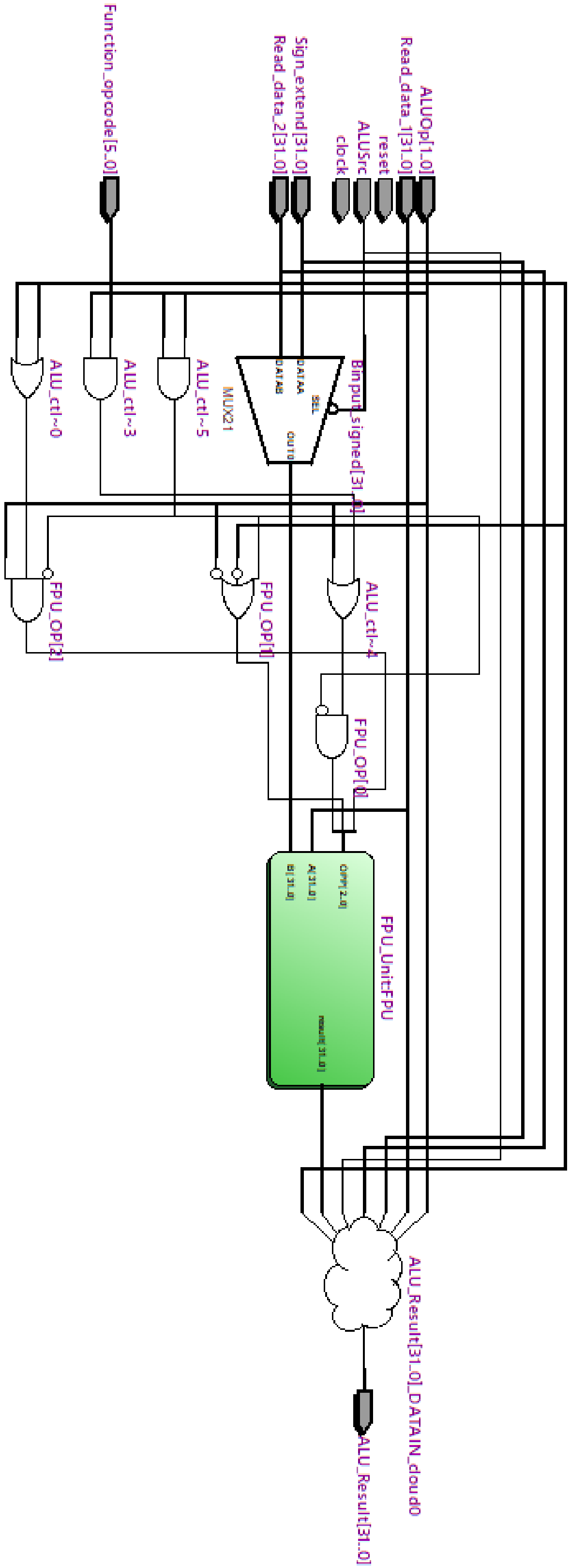


Figure 4.8.1: RTL Viewer for: Execute stage entity.

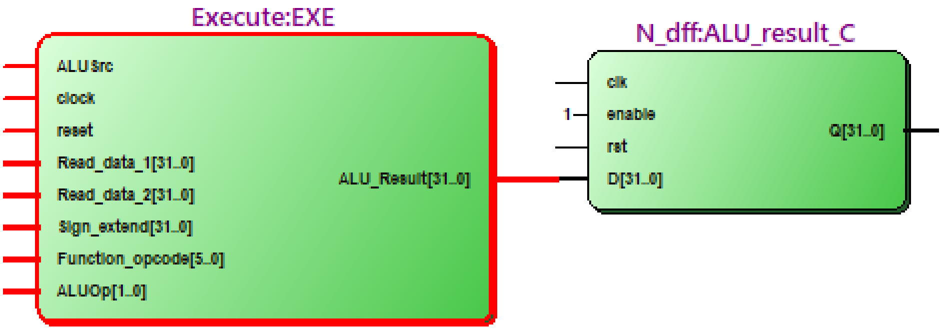


Figure 4.8.2: RTL Viewer for: Execute stage entity.

Description: The third stage in the pipeline is where the arithmetic- and logic-instructions will be executed. All instructions are executed with 32-bit operands and the result is a 32-bit word. An overflow event handler was not included in this project.

4.9 Control

File name: CONTROL.vhd

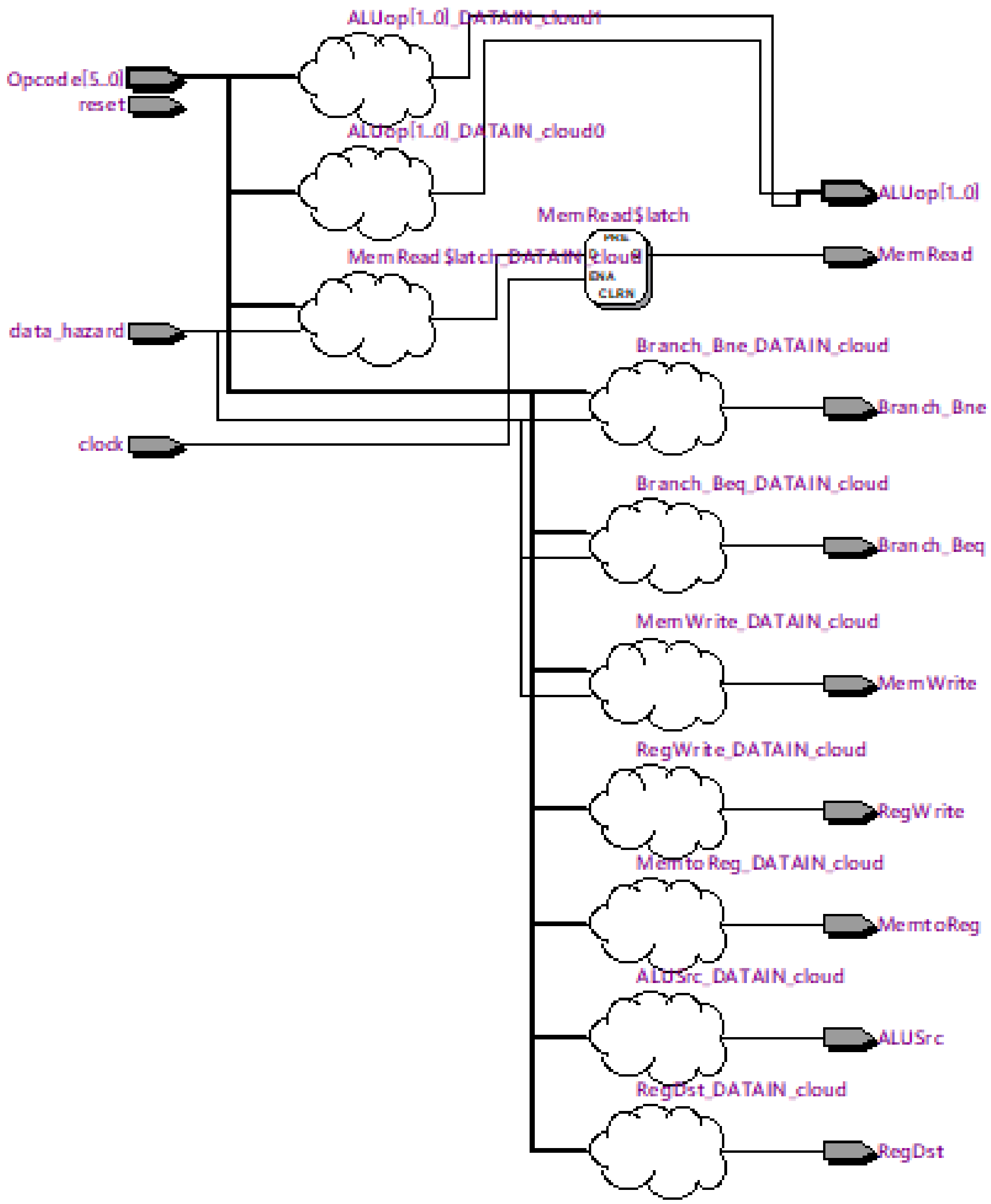


Figure 4.9.1: RTL Viewer for: Control stage entity.

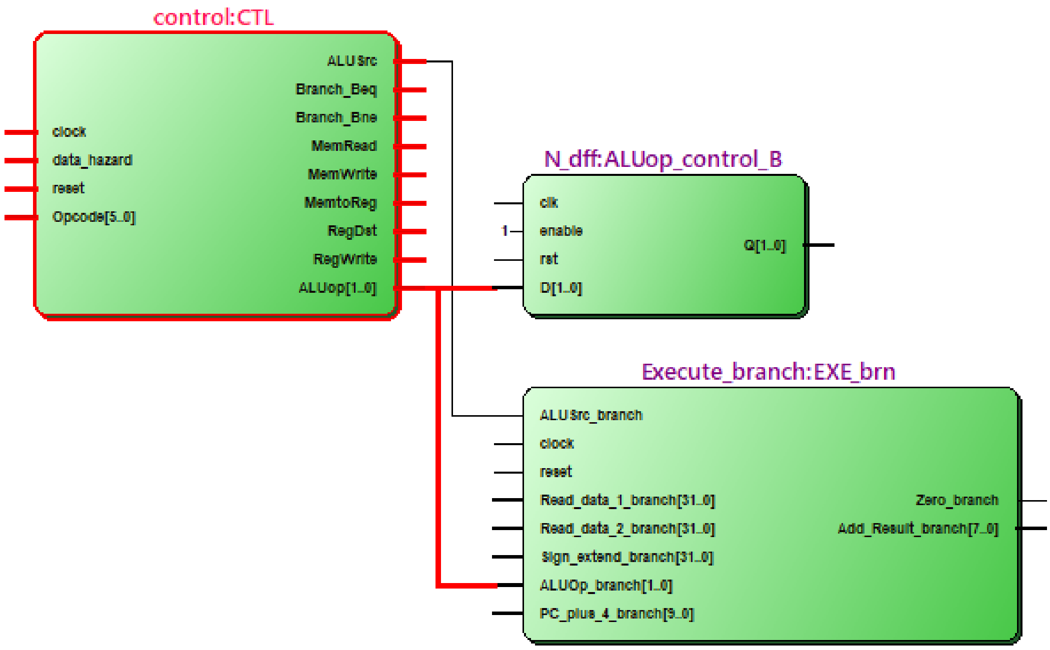


Figure 4.9.2: RTL Viewer for: Control stage entity.

Description: The third stage in the pipeline is where the arithmetic- and logic-instructions will be executed. All instructions are executed with 32-bit operands and the result is a 32-bit word. An overflow event handler was not included in this project.

4.10 Memory

File name: DMEMORY.vhd

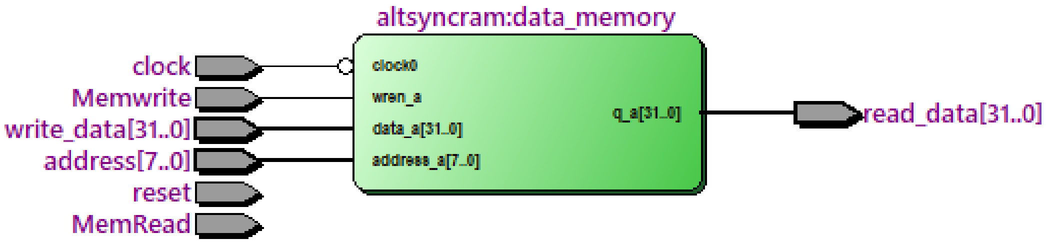


Figure 4.10.1: RTL Viewer for: W\R memory stage entity.

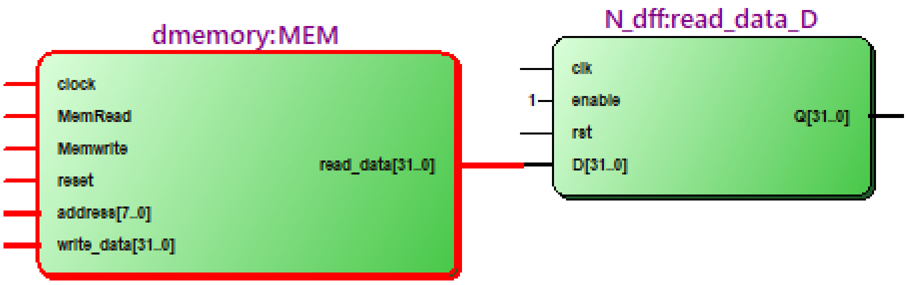


Figure 4.10.2: RTL Viewer for: W\R memory stage entity

Description: The Memory Access stage is the fourth stage of the pipeline. This is where load and store instructions will access data memory.

4.11 Hazard Unit

File name: HAZARD.vhd

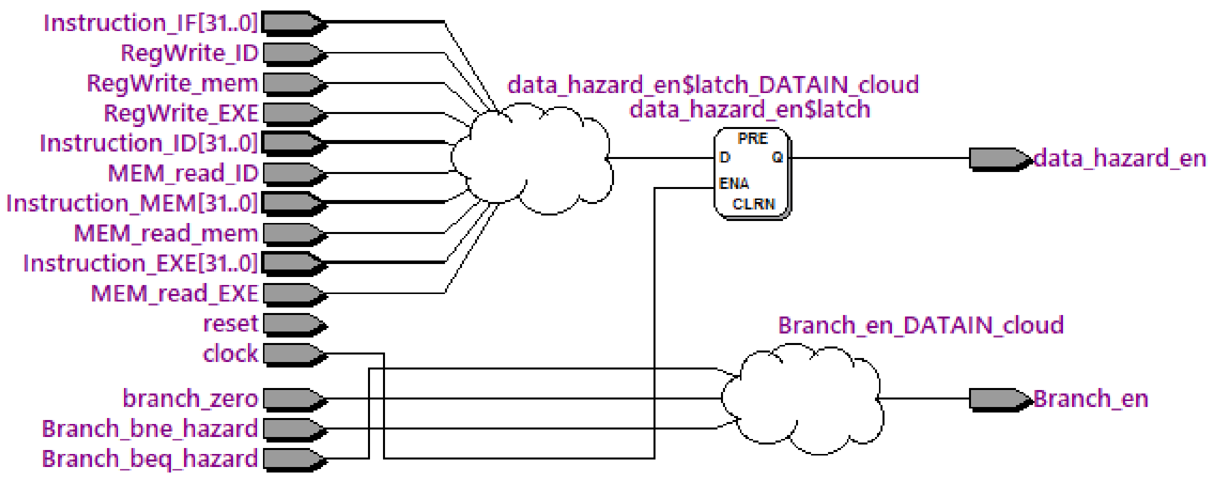


Figure 4.11.1: RTL Viewer for: Hazard unit entity.

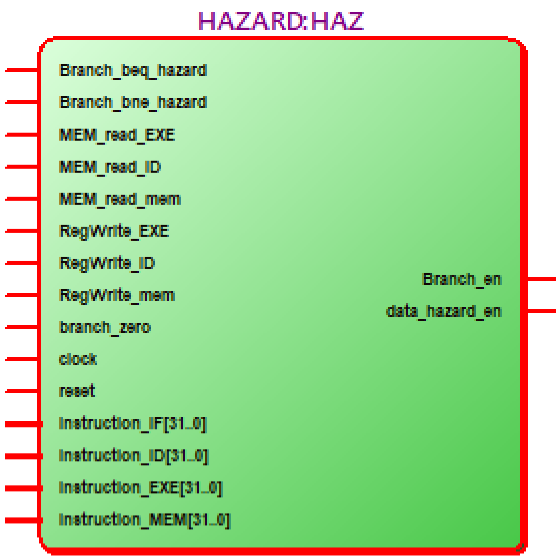


Figure 4.11.2: RTL Viewer for: Hazard unit entity.

Example: Data Hazard that required stalling

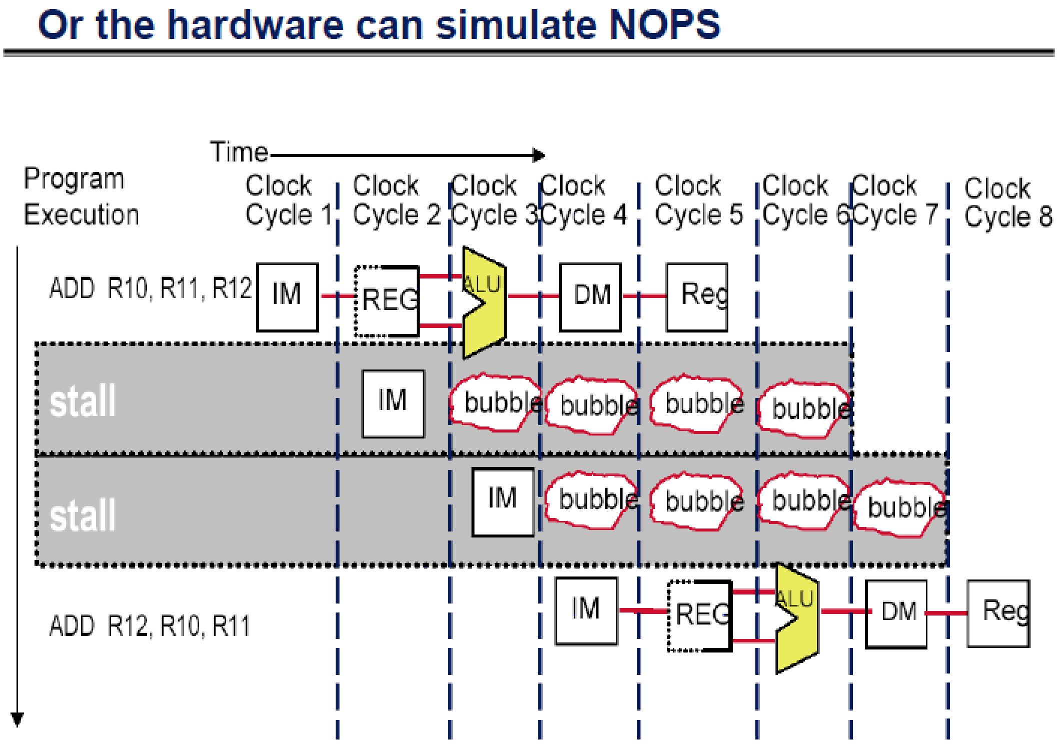


Figure 4.11.2: Data hazard example

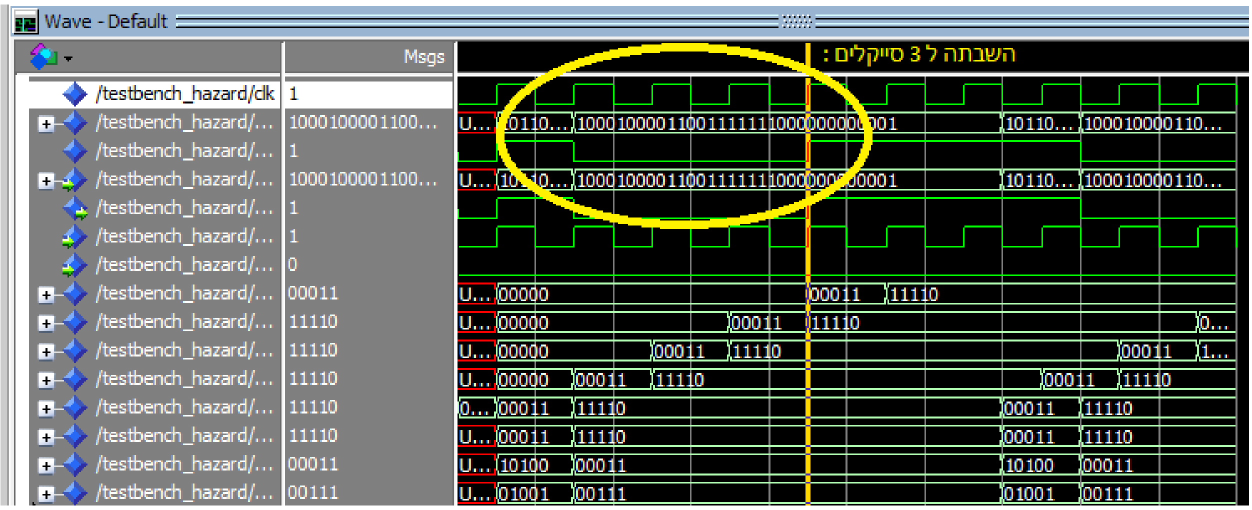


Figure 4.11.2: Hazard unit stalling

4.12 Execute Branch

File name: Execute_branch.vhd

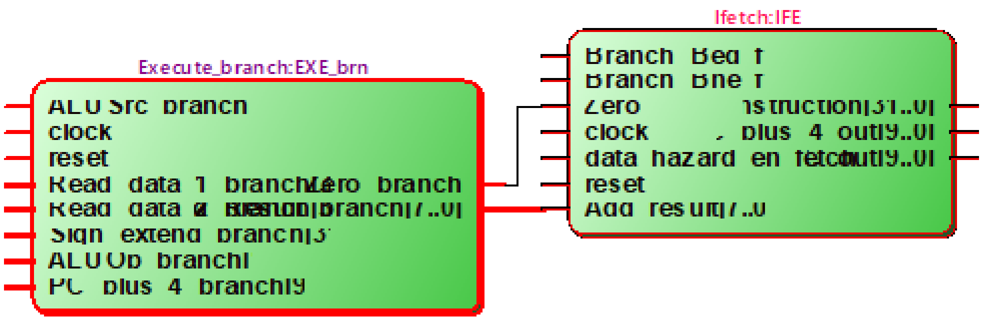


Figure 4.12.1: RTL Viewer for: Execute Branch entity.

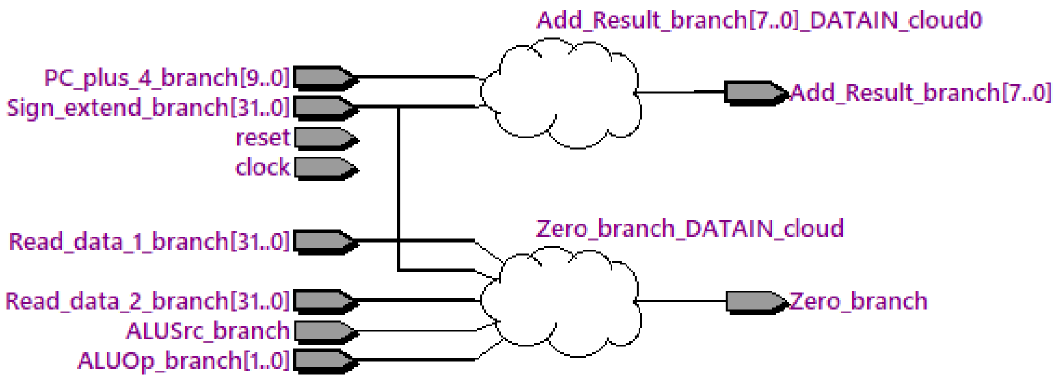
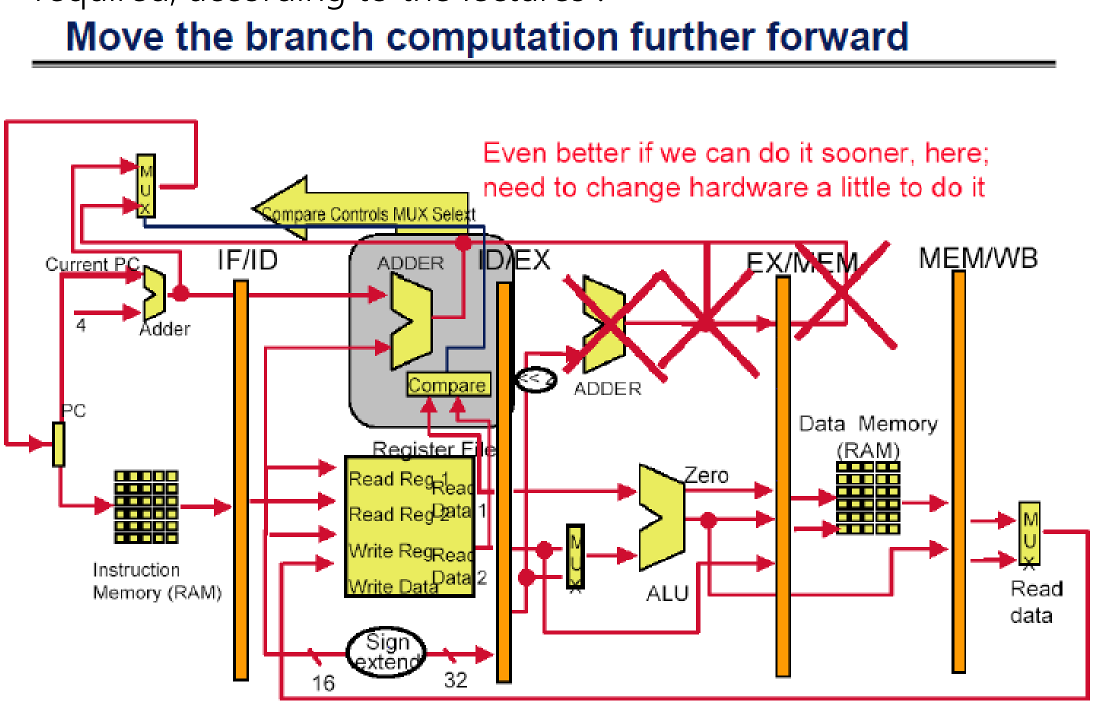


Figure 4.12.2: RTL Viewer for: Execute Branch entity.

Description: This component will compute in branch is required, according to the lectures :



Sorting Test

5.1 Assembly

As a test bench you need to write code which first of all have initialize necessary interrupts from buttons/switches. Secondary sort floating point vector (pre-stored in data memory, vector size is 8) and show the sorting vector on 7 segments display with delay of 1s.

BubbleSort_ieeeNumbers_DisplayOnModelSim.asm - 1 - 30 מאי 2018, 10:36

```
 .data
Array:      .word    0x433e0000,0x43480000,0x43340000,0x43200000,0xc3160000,0x432a0000,0x430c0000,0xc3020000 # vector size is 8 numbers
 .text
main:
    addi $t0, $0, 28      # 4 bytes per int * 10 ints = 32 bytes   , 28 to 32 is the last place
outterLoop:               # Used to determine when we are done iterating over the Array
    add $t1, $0, $0       # $t1 holds a flag to determine when the list is sorted
    add $t9, $0, $0    # Set $t9 to the base address of the Array
innerLoop:                  # The inner loop will iterate over the Array checking if a swap is needed
     lw  $t2, 0($t9)          # sets $t0 to the current element in array
     lw  $t3, 4($t9)         # sets $t1 to the next element in array
     lw  $t4, 4($t9)         # sets $t1 to the next element in array
     slt $t4, $t2, $t4       # $t5 = 1 if $t2 < $t3
     beq $t4, $0, continue   # if $t5 = 1, then swap them
     addi $t1, $0, 1             # if we need to swap, we need to check the list again
     sw  $t2, 4($t9)          # store the greater numbers contents in the higher position in array (swap)
     sw  $t3, 0($t9)         # store the lesser numbers contents in the lower position in array (swap)
continue:
     addi $t9, $t9, 4                # advance the array to start at the next location from last time
     beq  $t9, $t0, end    # If $t9 != the end of Array, jump back to innerLoop
     beq  $t1,  1,  outterLoop # $t1 = 1, another pass is needed, jump back to outterLoop
     beq  $t9, $t9, innerLoop     # If $t9 != the end of Array, jump back to innerLoop
```

Figure 5.1.1: Bubble sort for non-floating point numbers in assembly.

```
end :
    add $t5, $0, $0        # just fot display on ModelSim - load memory to the registes
    lw $t6, 0($t5)
    lw $t7, 4($t5)
    lw $s0, 8($t5)
    lw $s1, 12($t5)
    lw $s2, 16($t5)
    lw $s3, 20($t5)
    lw $s4, 24($t5)
    lw $s5, 28($t5)
```

Figure 5.1.1: Optional – load the memory to the register for ModelSIM view .

To be able to sort floating point numbers we implement a new R format command : sltF – set if less than (floating point numbers).

```
lw   $t2, 0($t9)          # sets $t0 to the current element in array
lw   $t3, 4($t9)          # sets $t1 to the next element in array
lw   $t4, 4($t9)          # sets $t1 to the next element in array
slt $t4, $t2, $t4         # $t5 = 1 if $t2 < $t3
beq $t4, $0, continue     # if $t5 = 1, then swap them
addi $t1, $0, 1              # if we need to swap, we need to check the list again
sw   $t2, 4($t9)          # store the greater numbers contents in the higher positi
```

Figure 5.2.2: slt command in the algorithm.

Mnemonic	Format	OP code	Function field	instruction
SltF	R	0	61	Slt in FPU

Table 5.2.1: slt command format

After importing the program.hex file from MARS we changed the hex code of slt command (more precisely, we changed only the function field section) **from 42 (0x2A) to 61 (0x3D).**

Slt	R	0	42	Set if Less Than

Figure 5.2.3: slt command in the overview – function field is 32.

```
0x014b682a  slt $13,$10,$11    14:    slt $t5, $t2, $t3
```

Figure 5.2.4: slt command in MARS analyze– function field is 32(0x2A - 6 LSB bits).

```
:0400000020080000D4
:0400040021080028A7
:04000800000048208C
:04000C0020040000CC
:040010008C8A0000D6
:040014008C8B0004CD
:04001800014B682A05
:04001C0011A000032C
:0400200020090001B2
:04002400AC8A00049E
:04002800AC8B00009D
:04002C002084000428
:040030001488FFF73A
:040034001520FFF4A0
:00000001FF
```

```
:0400000020080000D4
:0400040021080028A7
:04000800000048208C
:04000C0020040000CC
:040010008C8A0000D6
:040014008C8B0004CD
:04001800014B683D05
:04001C0011A000032C
:0400200020090001B2
:04002400AC8A00049E
:04002800AC8B00009D
:04002C002084000428
:040030001488FFF73A
:040034001520FFF4A0
:00000001FF
```

Figure 5.2.5: slt command in hex dump– function field is 32(0x2A - 6 LSB bits).

Data memory numbers & Sorting

Data Memory :		The sorting should be:	
0x43480000	200	0x43480000	200
0x433e0000	190	0x433e0000	190
0x43340000	180	0x43340000	180
0x432a0000	170	0x432a0000	170
0x43200000	160	0x43200000	160
0xc3160000	-150	0x430c0000	140
0x430c0000	140	0xc3020000	-130
0xc3020000	-130	0xc3160000	-150

(14)	43480000	43480000
(15)	433E0000	433E0000
(16)	43340000	43340000
(17)	432A0000	432A0000
(18)	43200000	43200000
(19)	430C0000	430C0000
(20)	C3020000	C3020000
(21)	C3160000	C3160000

Figure 5.2.6: Registers after MIPS test bench – sorted values !

Test Benchs

6.1 full_adder.VHD

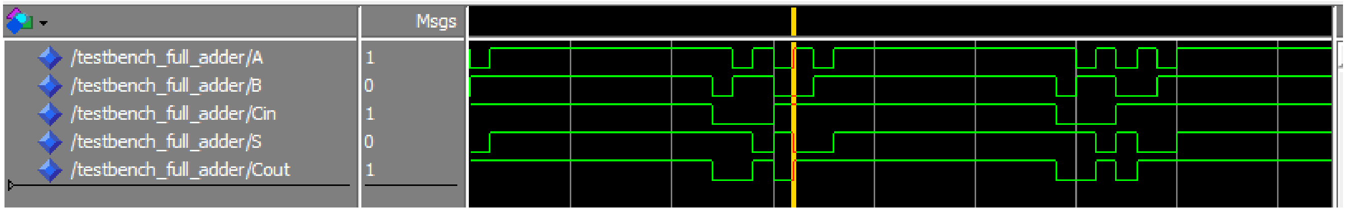


Figure 6.1: Test Bench for: full_adder entity

Description: '1'(A) + '0'(B) + '1'(Cin) = '0' ('1'' Cout)

6.2 ADD_SUB.VHD

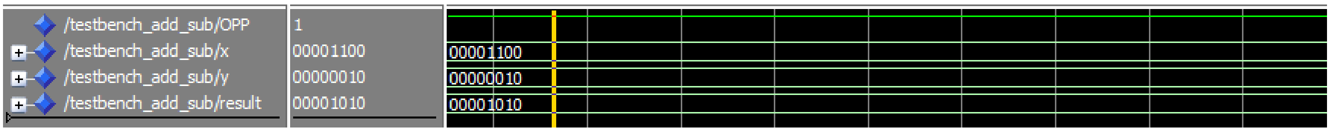


Figure 6.2: Test Bench for: ADD_SUB operation

Description: OPP: SUB ('1'): "1100"(12, x) – "0010"(2, y) = "1010"(10, result)

6.3 ADD.VHD

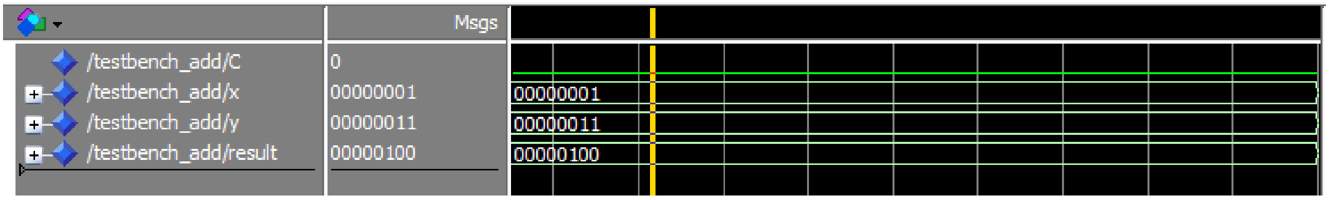


Figure 6.3: Test Bench for: ADD entity

Description: '0' (C) + "001"(1, x) + "011"(3, y) = "100"(4, result)

6.4 MAX_MIN.VHD

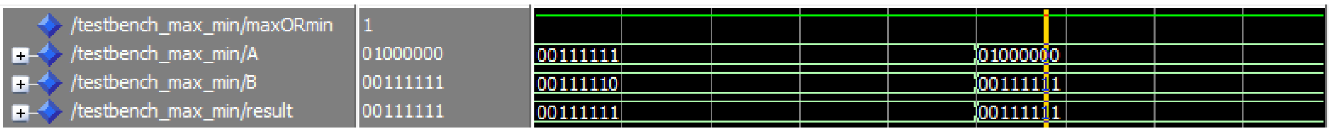


Figure 6.4: Test Bench for: MAX_MIN operation

Description: OPP : MIN (maxORmin = '1') : result = min(A,B) = B.

6.5 MUL.VHD

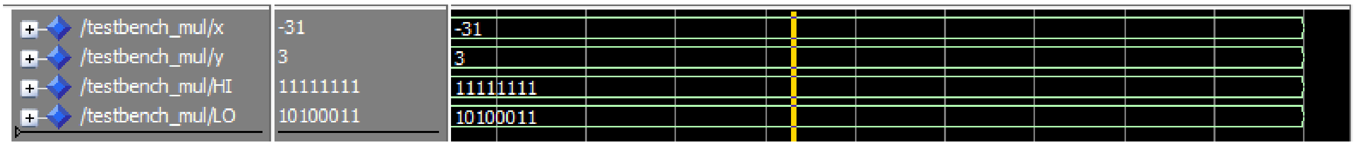


Figure 6.5: Test Bench for: MUL entity

Description: result = (HI,LO) = "1111111110100011" (-93) = -31(x) * 3(y).

6.6 diff_1bit.VHD

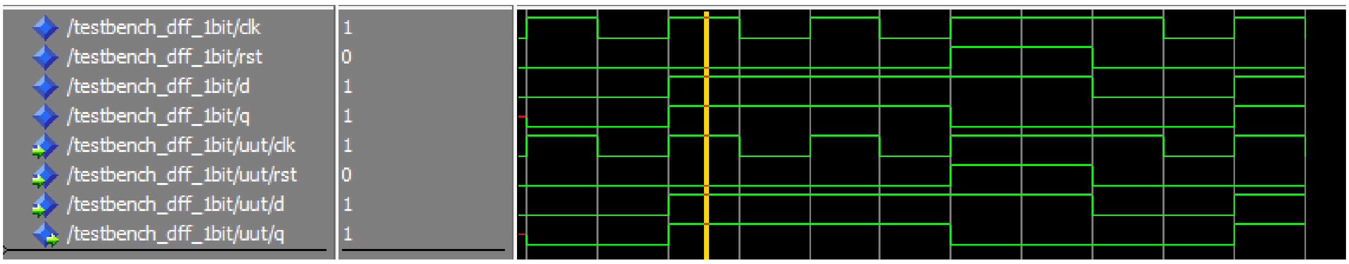


Figure 6.6: Test Bench for: diff_1bit entity

Description: if clk is rising edge then $d \rightarrow q$.

6.7 N_dff.VHD

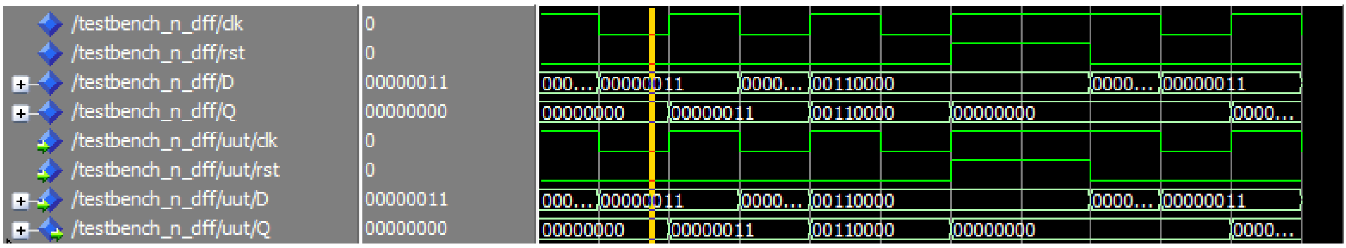


Figure 6.7: Test Bench for: N_dff entity

Description: if clk is rising edge then $D \rightarrow Q$.

6.8 mux_Nbits.VHD

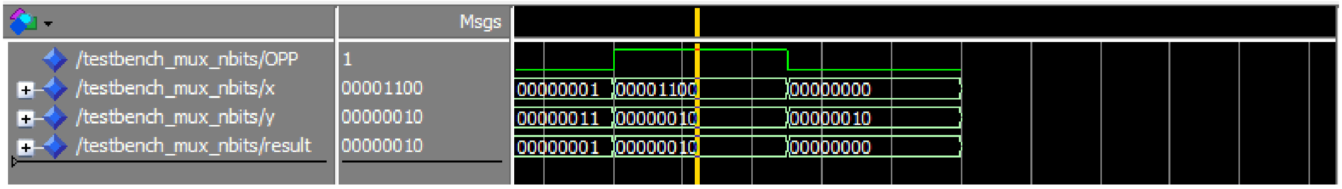


Figure 6.8: Test Bench for: mux_Nbits entity

Description: OPP(local signal in the test bench, it is SEL) : ='1', then result = y ('0' for result = x).

6.9 shift_Nbits.VHD

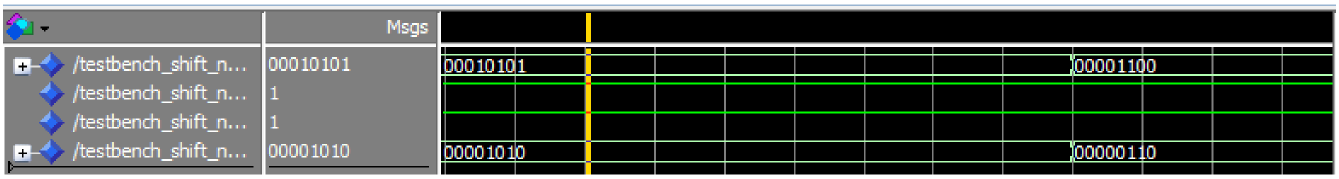


Figure 6.9: Test Bench for: shift_Nbits entity

Description: dir = '1' (shift to the right), enable = '1' (then shift the number instead of result=input).

6.10 ADD_SUB_FPU.VHD

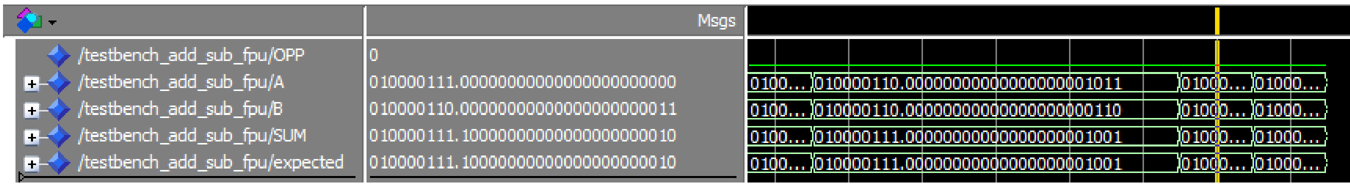


Figure 6.10: Test Bench for: ADD\SUB FPU entity

Description: As shown the SUM is as expected (expected result calculated using online calculators).

6.11 MUL_FPU.VHD

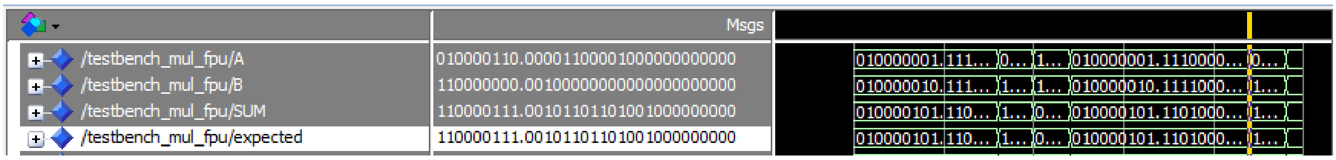


Figure 6.11: Test Bench for: MUL FPU entity

Description: As shown the result is as expected (expected result calculated using online calculators).

6.12 LeadingZeros_Counter.VHD

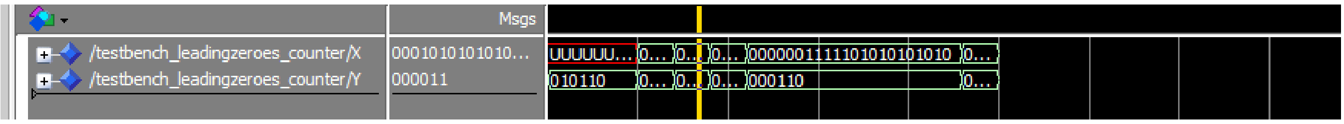


Figure 6.12: Test Bench for: Leading Zeros Counter entity

Description: The input number are with 3 zeros, and the result is 3 as expected.

6.13 FPU_Unit.VHD

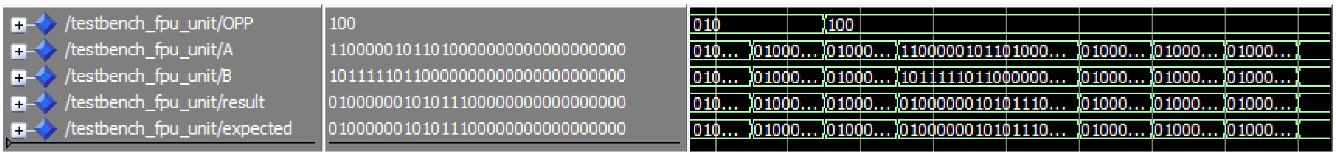


Figure 6.13: Test Bench for: FPU_Unit entity

Description: OPP = "1100" -> MUL F. As shown the result is as expected (expected result calculated using online calculators).

6.14 shift_Unit.VHD

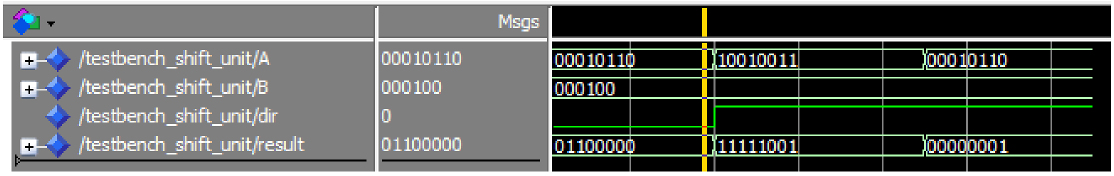


Figure 6.14: Test Bench for: shift_Unit entity

Description: dir = '0' (shift to the left), B = 4, the output is the number A shifted B times to the left as expected.

6.15 MIPS.VHD

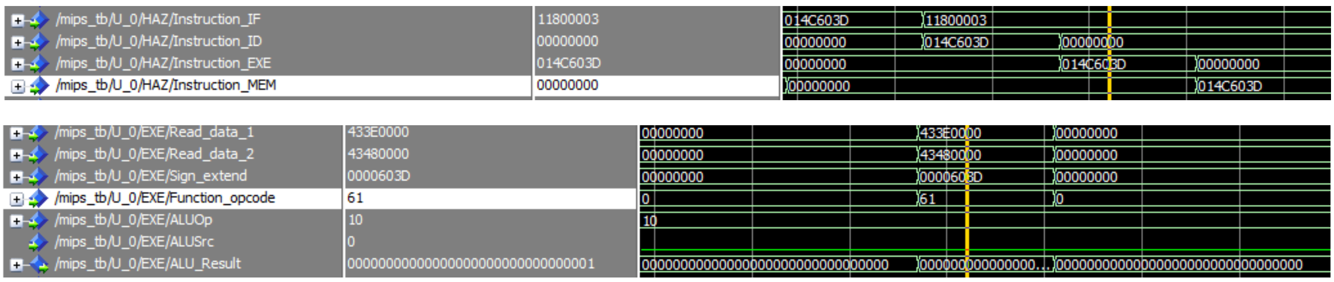


Figure 6.15: Test Bench for: MIPS

Description: the instruction registers is shown in the hazard unit. Currently the Instruction_exe is the executed insturctuion. In Execute component we get the instruction OPCode (0x3D = 111101 binary = 61 decimal) which is the sltF command. Operand 1 is 0x433e0000 190, and operand 2 is 0x43480000 200– the result should be the MSB of operand1-opernd 2, which is 1 as required (so swap in memory is going to executed).

6.16 HAZARD.VHD

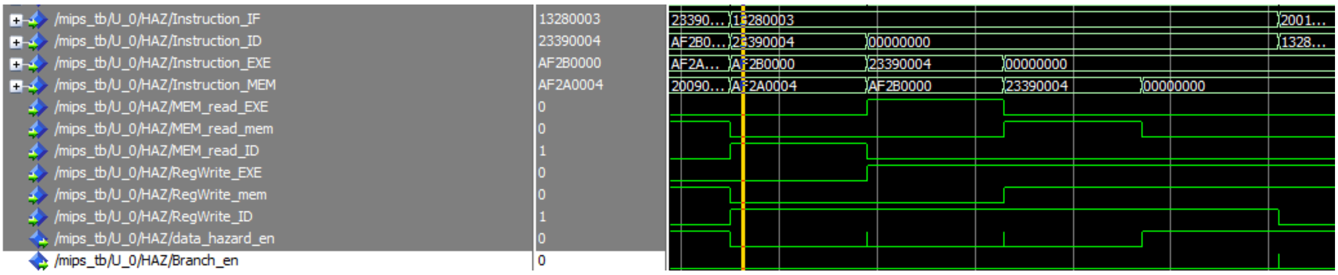


Figure 6.16: Test Bench for: HAZARD

Description: the command in Instruction_IF is the current feteched command – the command has operands that are the target/destintion of the previous commands (instruction_ID) – data hazard is enabled (data_hazard_en -> '0') and stalling be executed.

Attached files

- VHDL/rtlMIPS/ - VHDL files
- VHDL/aidMIPS/ - VHDL files
- TB/ Test Bench files
- DOC/ readme.txt – compilation order