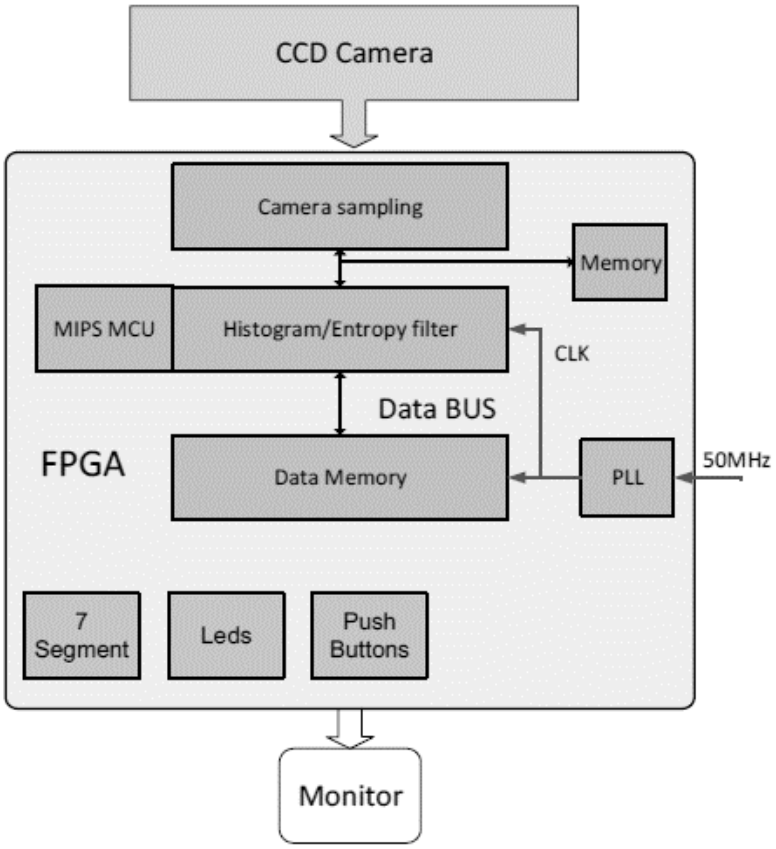
Final Project

Architecture of CPU

Entropy Filter and Histogram

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 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. Design, synthesize and analyze a digital filter. Understanding FPGA memory structure.

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.
- Design a Real Time entropy detection filter for an image received from a camera. The histogram of the image must be calculated.

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

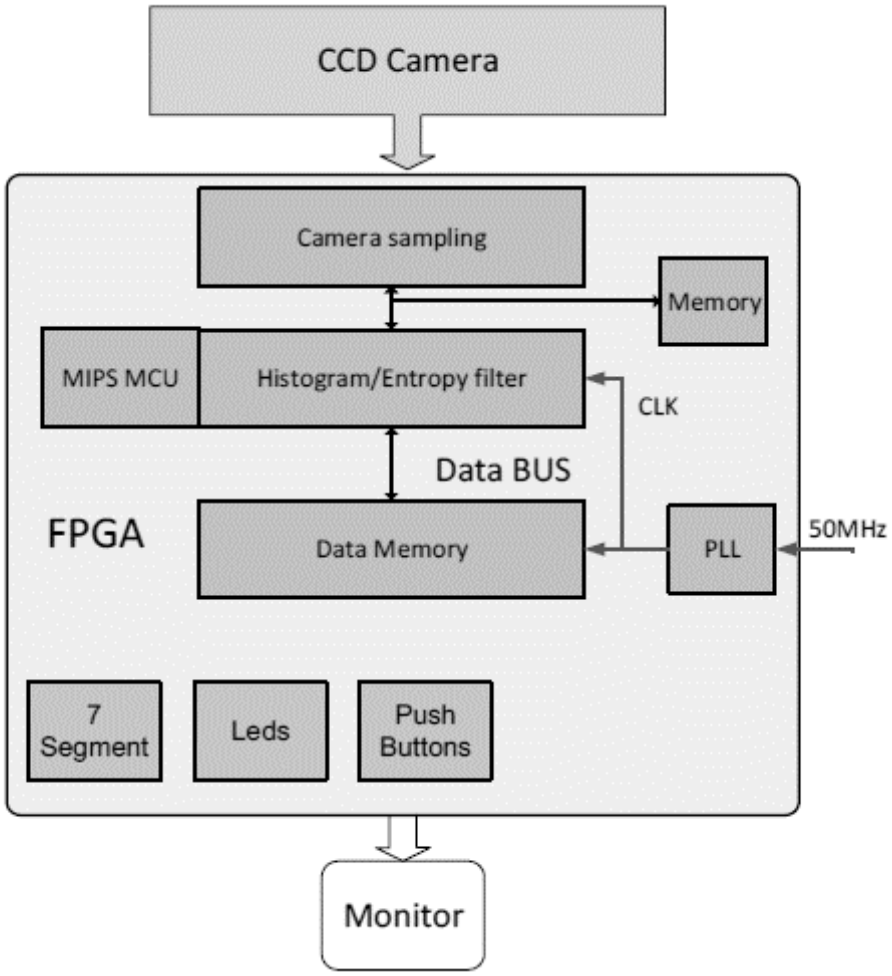


Figure 1.1: Overall system design

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**

5/9/2015 kolaman@post.bgu.ac.il עמיר קולמן 9

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 have initialize necessary interrupts from buttons/switches.
While performing the RT image processing the MIPS CPU must continue to work (run a program). The PLL is used to make a higher frequency from the 50MHz clock and is used in FPGA compilation only.

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

- Pipelined MIPS
- DE1-D5M – contain the sampling & filters units

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.

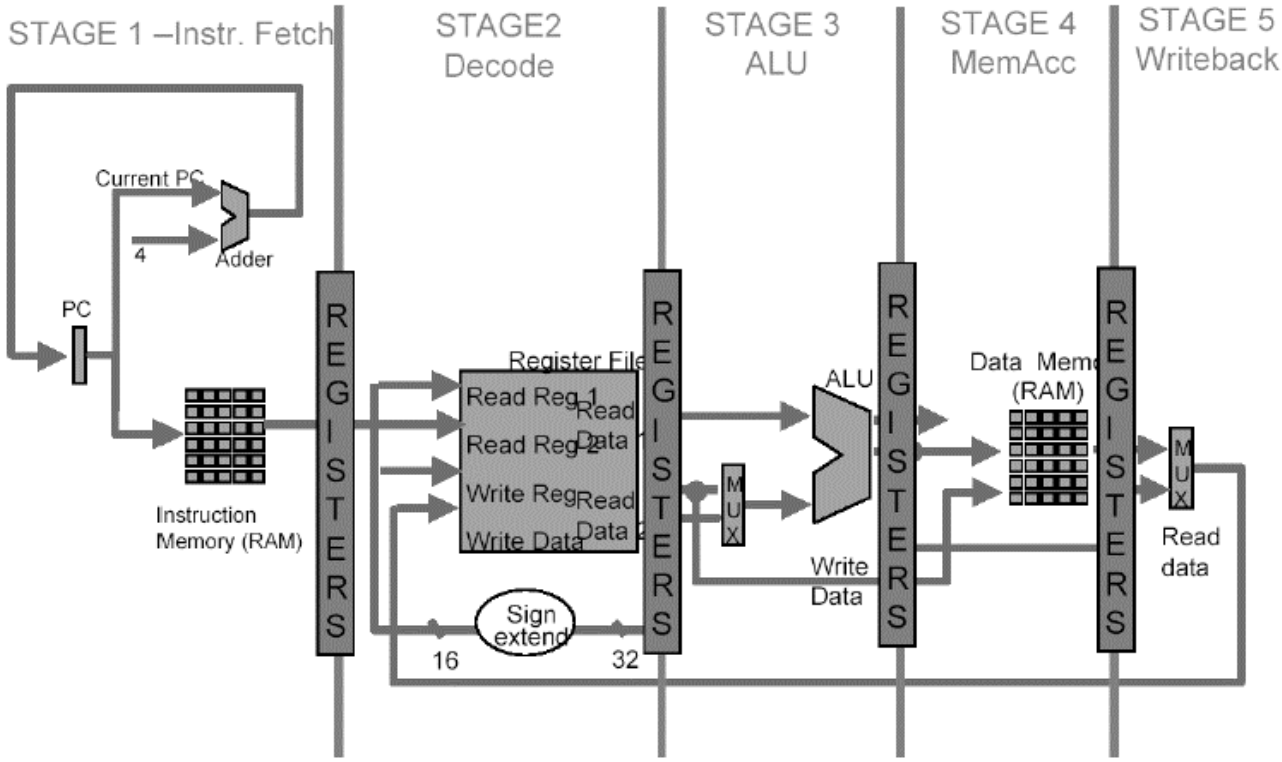


Figure 1.2: Pipeline stages

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.

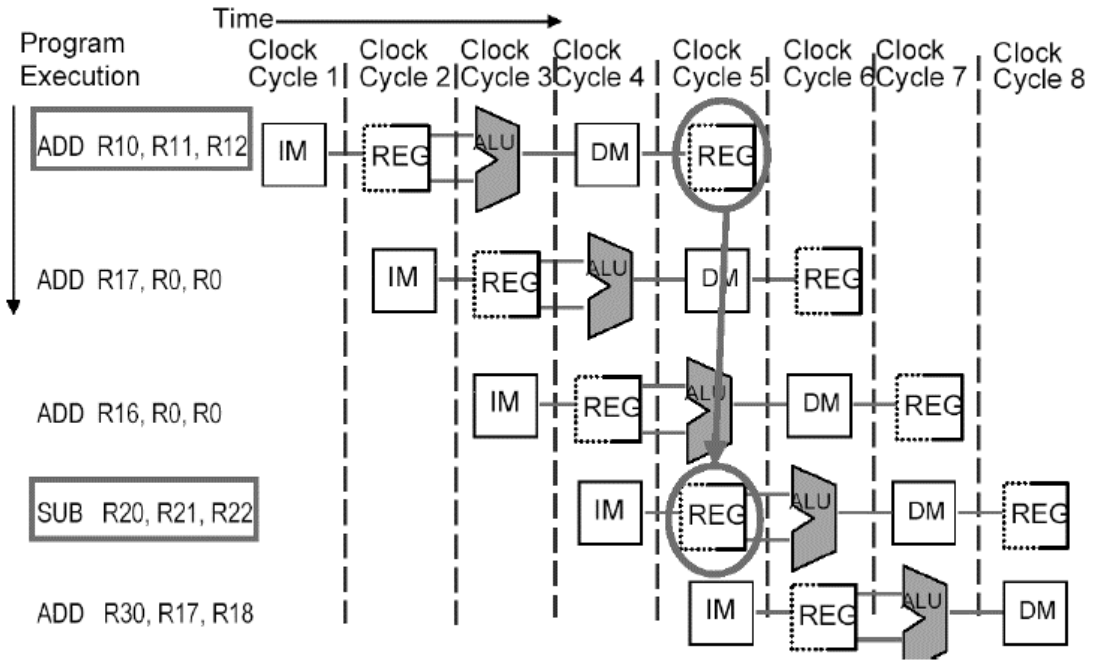


Figure 1.3: hazard example

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.

Image processing

As a test bench, you will have to perform an entropy filter [1],[6] on the image taken from camera. The size of the filter will be 6x6. In addition to the filtering process, the image histogram [5] will be calculated. Switches will allow to select the image presented on the monitor (original, filtered or histogram). A general diagram of the processing is shown. The test will be in the following steps:

1. Capture image from camera, convert to gray scale.
2. MCU Receive interrupt from the camera when a frame is saved into memory.
3. Send command to hardware to perform the filtering or histogram calculation.
4. Receive interrupt when hardware done, and filtered frame saved into memory
5. An original image filtered image or histogram will be presented on monitor
6. depending on switches.
7. MCU must run counter in main loop and print to 7S the time in resolution of seconds.

Image

Images may be two-dimensional, such as a photograph or screen display, or three-dimensional, such as a statue or hologram. They may be captured by optical devices – such as cameras, mirrors, lenses, telescopes, microscopes, etc. and natural objects and phenomena, such as the human eye or water.

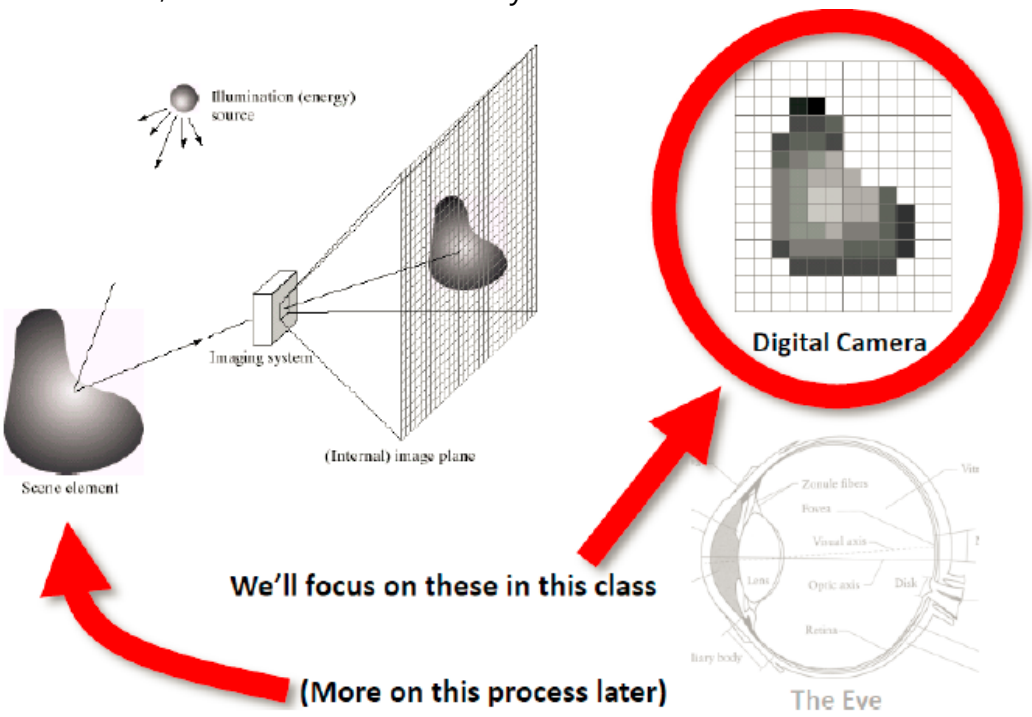
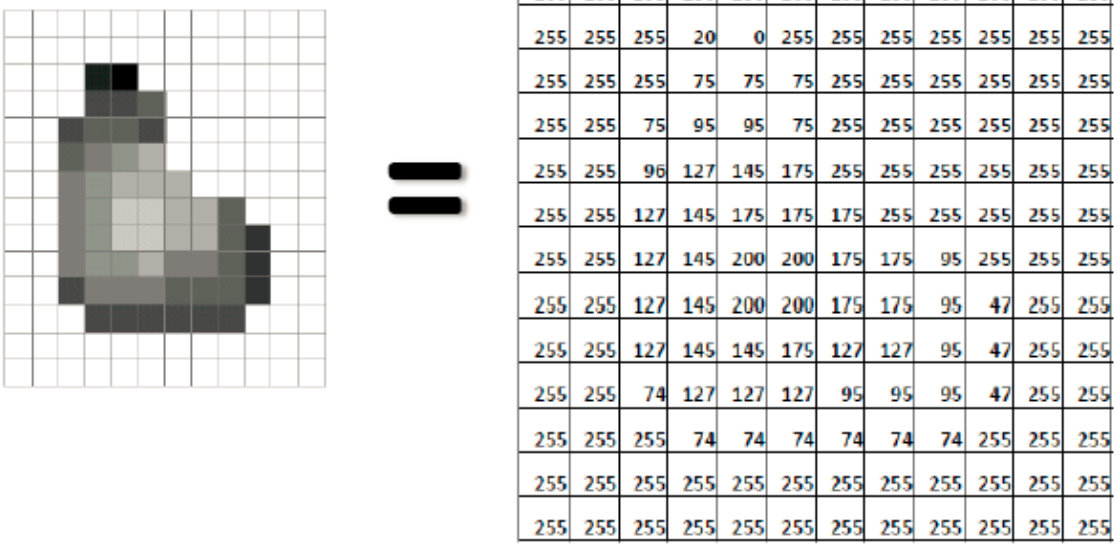


Figure 1.4: example of image sampling

The term resolution is often considered equivalent to pixel count in digital imaging, though international standards in the digital camera field specify it should instead be called "Number of Total Pixels" in relation to image sensors, and as "Number of Recorded Pixels" for what is fully captured. Hence, JCIA & CIPA suggest notation such as "Number of Recorded Pixels 1000 × 1500". According to the same standards, the "Number of Effective Pixels" that an image sensor or digital camera has is the count of pixel sensors that contribute to the final image (including pixels not in said image but nevertheless support the image filtering process), as opposed to the number of total pixels, which includes unused or light-shielded pixels around the edges.



255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	20	0	255	255	255	255	255	255	255
255	255	255	75	75	75	255	255	255	255	255	255
255	255	75	95	95	75	255	255	255	255	255	255
255	255	96	127	145	175	255	255	255	255	255	255
255	255	127	145	175	175	175	255	255	255	255	255
255	255	127	145	200	200	175	175	95	255	255	255
255	255	127	145	200	200	175	175	95	47	255	255
255	255	127	145	145	175	127	127	95	47	255	255
255	255	74	127	127	127	95	95	95	47	255	255
255	255	255	74	74	74	74	74	74	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255

Figure 1.5: A grid (matrix) of intesity values

Histogram

A histogram is an accurate representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable (quantitative variable) and was first introduced by Karl Pearson. It differs from a bar graph, in the sense that a bar graph relates two variables, but a histogram relates only one. To construct a histogram, the first step is to "bin" the range of values—that is, divide the entire range of values into a series of intervals—and then count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable. The bins (intervals) must be adjacent and are often (but are not required to be) of equal size.

Histograms give a rough sense of the density of the underlying distribution of the data, and often for density estimation: estimating the probability density function of the underlying variable. The total area of a histogram used for probability density is always normalized

to 1. If the length of the intervals on the x-axis are all 1, then a histogram is identical to a relative frequency plot.

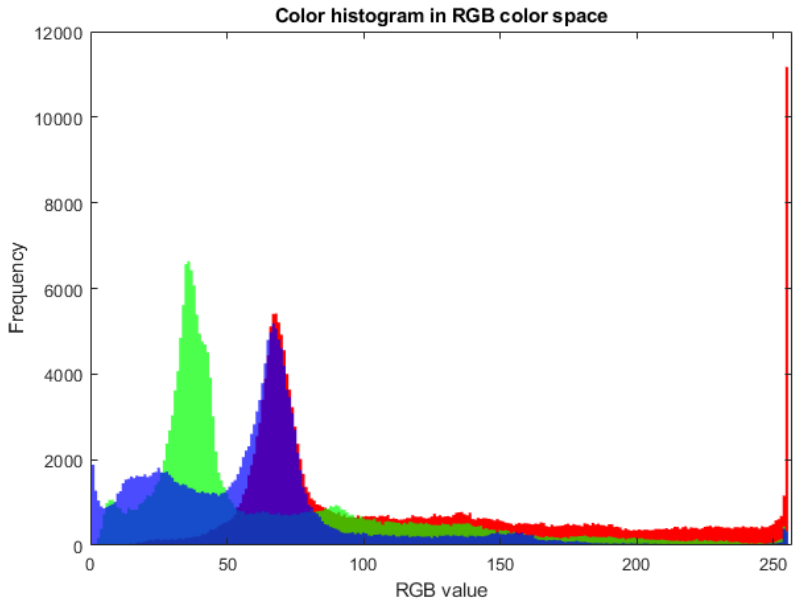


Figure 1.6: example of Color histogram in RGB

Suppose we choose k levels of classification in the histogram, therefor :

$H(k)$ = specifies the number of pixels with gray-value k
let N be the number of total pixels : N= $\sum_{i=1}^{k} H(i)$
$P(k) = \frac{H(k)}{N}$ defines the normalized histogram

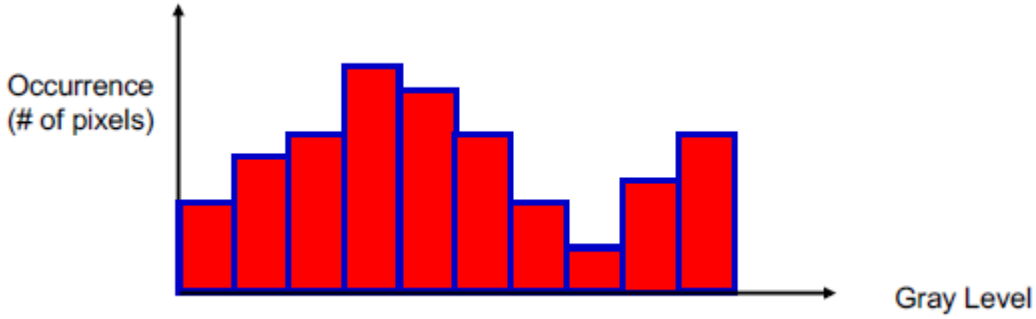


Figure 1.7: example of Color histogram in RGB

Image filtering

In computer science, Digital image processing is the use of computer algorithms to perform image processing on digital images. As a subcategory or field of digital signal processing, digital image processing has many advantages over analog image processing. It allows a much wider range of algorithms to be applied to the input data and can avoid problems such as the build-up of noise and signal distortion during processing. Since images are defined over two dimensions (perhaps more) digital image processing may be modeled in the form of multidimensional systems.

Gray-scale

In photography, computing, and colorimetry, a grayscale or greyscale image is one in which the value of each pixel is a single sample representing only an amount of light, that is, it carries only intensity information. Images of this sort, also known as black-and-white or monochrome, are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest.

Grayscale images can be the result of measuring the intensity of light at each pixel according to a weighted combination of frequencies (or wavelengths), and in such cases they are monochromatic proper when only a single frequency (in practice, a narrow band of frequencies) is captured. The frequencies can in principle be from anywhere in the electromagnetic spectrum (e.g. infrared, visible light, ultraviolet, etc.).

All grayscale algorithms utilize the same basic three-step process:

1. Get the red, green, and blue values of a pixel
2. Use fancy math to turn those numbers into a single gray value

3. Replace the original red, green, and blue values with the new gray value

Average method is the simplest one. You just must take the average of three colors. Since it's an RGB image, so it means that you have add r with g with b and then divide it by 3 to get your desired grayscale image.

But the results were not as expected. We wanted to convert the image into a grayscale, but this turned out to be a rather black image. This problem arises due to the fact, that we take average of the three colors. Since the three different colors have three different wavelength and have their own contribution in the formation of image, so we must take average according to their contribution.

The chosen formula :

Some modern digital image and video formats use a different recommendation (BT.601), which calls for slightly different coefficients:

$$Gray = (Red * 0.299 + Green * 0.587 + Blue * 0.114)$$

Figure 1.8: A desaturated image. Desaturating an image takes advantage of the ability to treat the (R, G, B) colorspace as a 3-dimensional cube.

Enthropy filter

Entropy is the quantitative measure of disorder in a closed but changing system, a system in which energy can only be transferred in one direction from an ordered state to a disordered state.

The higher the entropy, the higher the disorder and lower the availability of the system's energy to do useful work.

Although the concept of entropy originated in thermodynamics and statistical mechanics, it has found applications in a lot of subjects such as communications, economics, information science and technology.

The exact definition is:

$$Entropy(I) = -\sum_{k} P(k)\log P(k)$$

Since the logarithm of a number always increases as the number increases, the more possible states that the system can be in (given that it has a volume, energy, pressure, and temperature), the greater the entropy.

As a mathematical function, entropy is used to measure the level of disorder within a certain sample of values, a certain entropy value is assigned to a pixel by measuring entropy of a sample of pixel values present in a given window around that pixel.

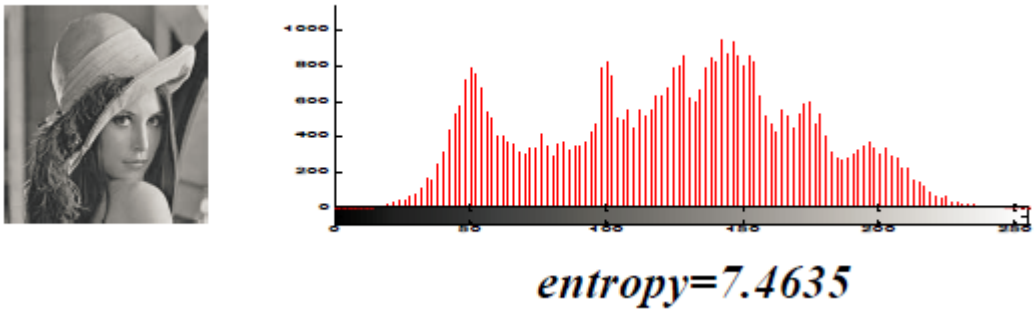


Figure 1.9: A desaturated image by entropy filter.

Modules Description

3.1 Full Adder

File name: full_adder.vhd

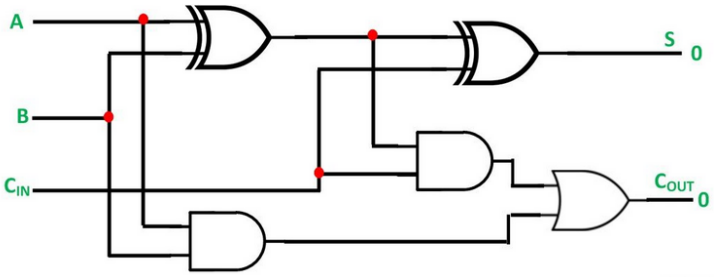


Figure 3.1 : Graphical description for : Full Adder

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

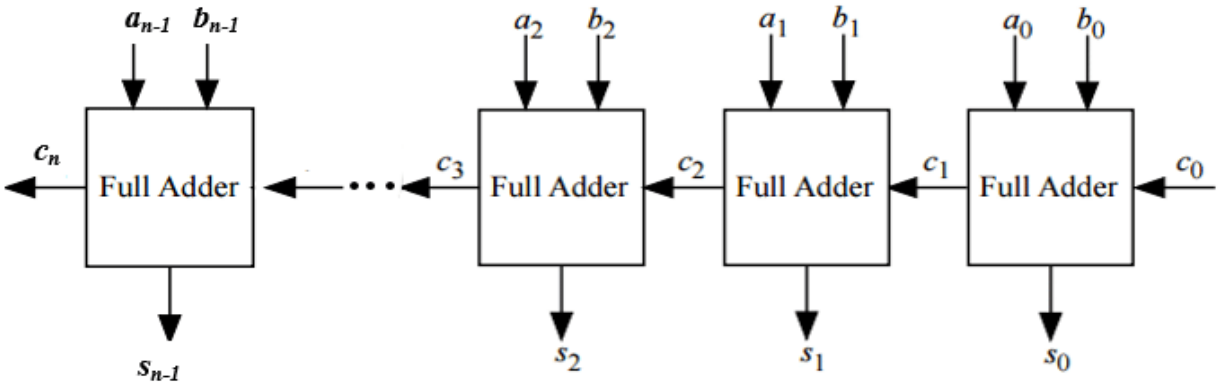


Figure 3.2 : Graphical description for : Adder n-bit

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

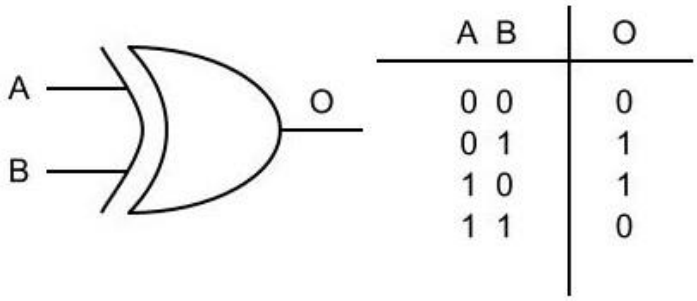


Figure 3.3: Graphical description for: XOR gate

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

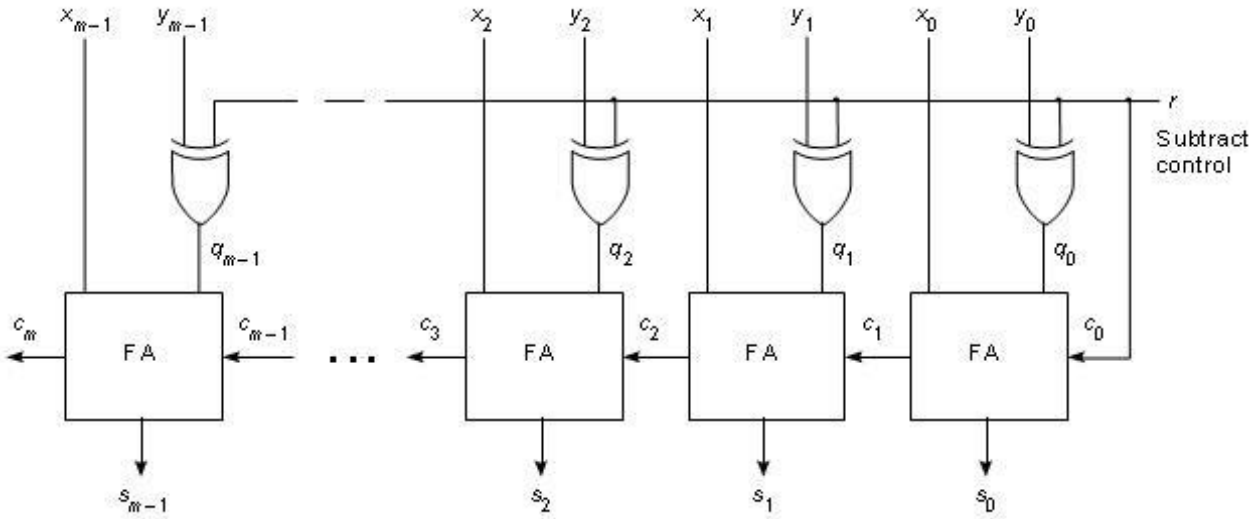


Figure 3.4: Graphical description for: Adder/Subtractor n-bit

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

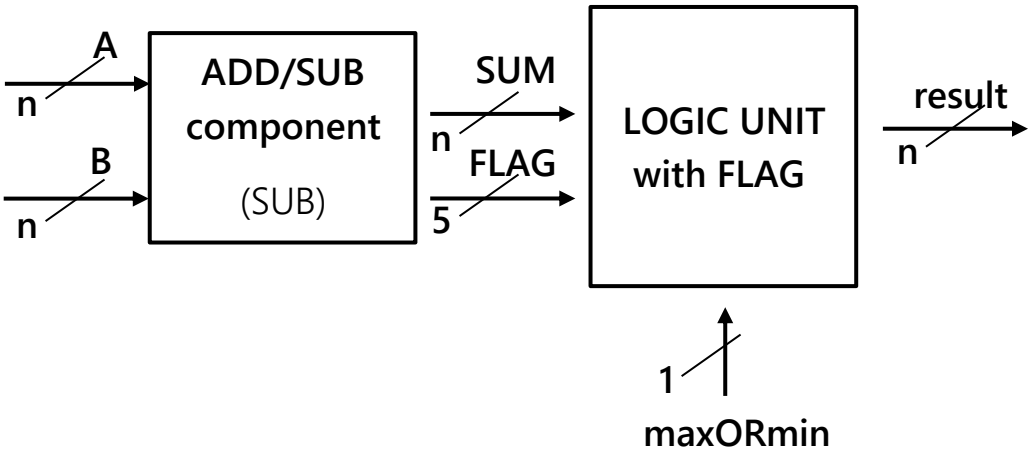


Figure 3.5: Graphical description for: MAX/MIN operation

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

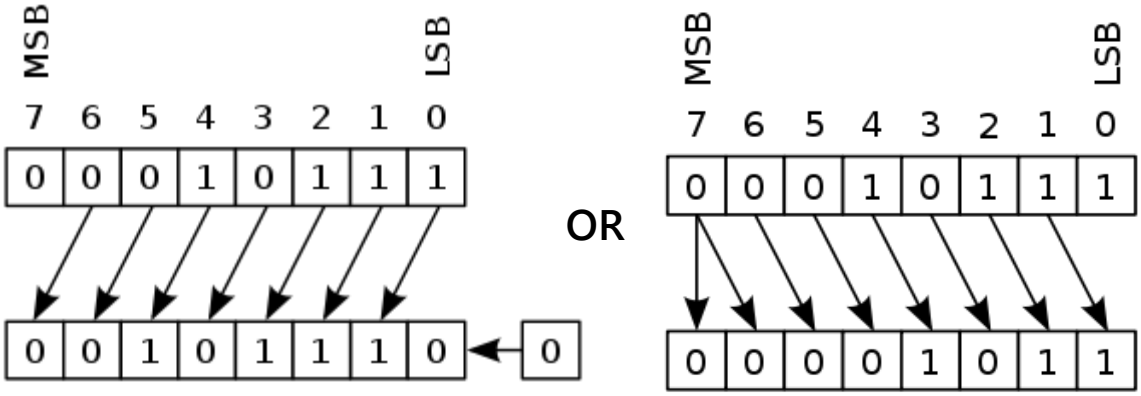


Figure 3.6: Graphical description for: Shift Left/Right (1-bit shifter)

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

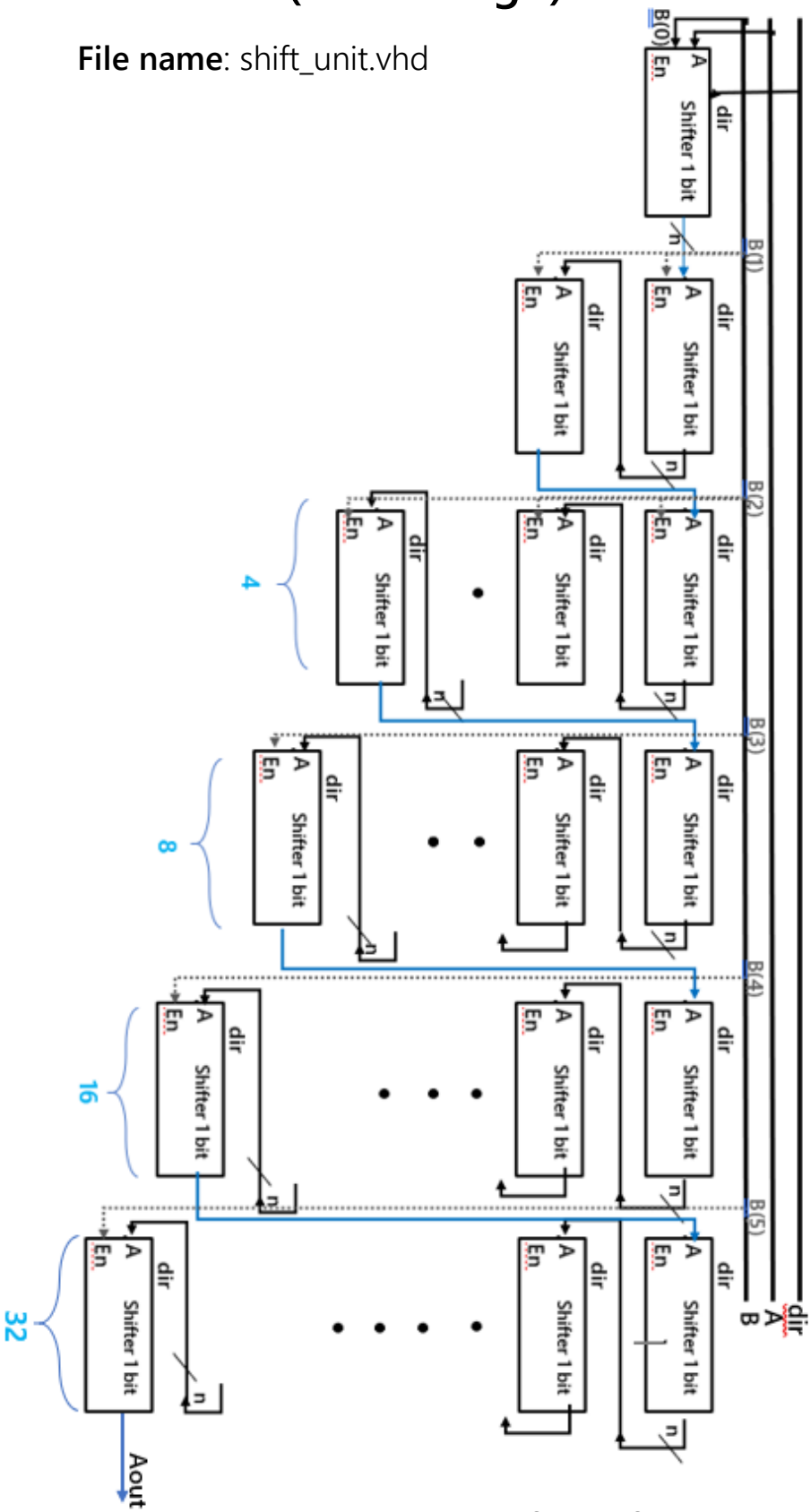


Figure 3.7: Graphical description for: Shift Unit (B-bits shifter)

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

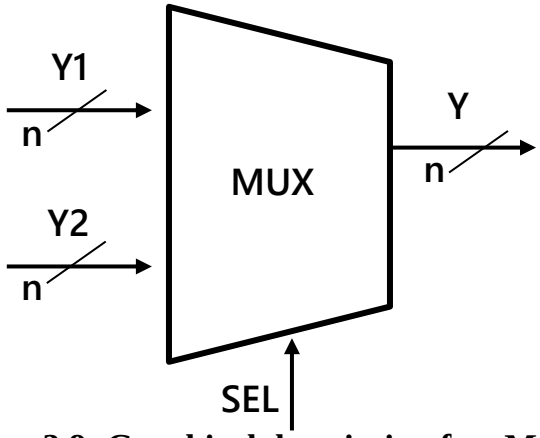


Figure 3.8: Graphical description for: Mux 2N-N bit

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

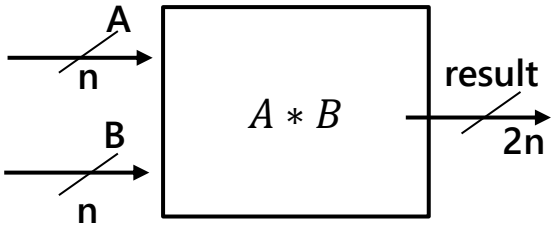


Figure 3.9: Graphical description for: MUL operation

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

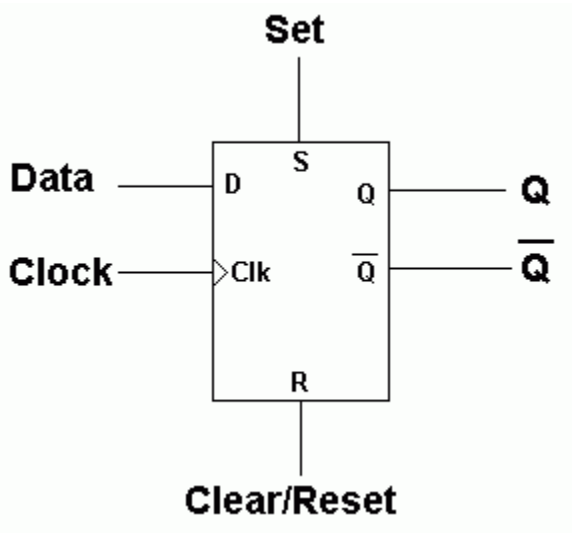


Figure 3.10: Graphical description for: 1-bit dff entity.

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

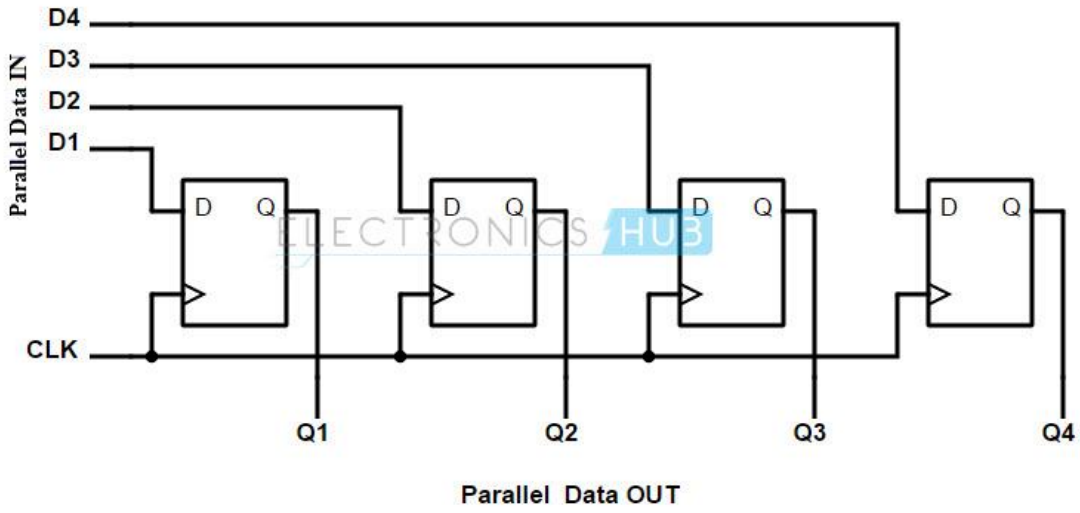


Figure 3.11: Graphical description for:(example N=4) N-bit dff entity.

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

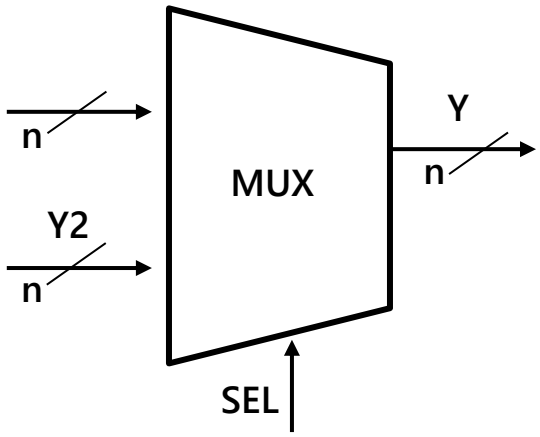


Figure 3.12: Graphical description for Swap Nbits numbers entity.

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

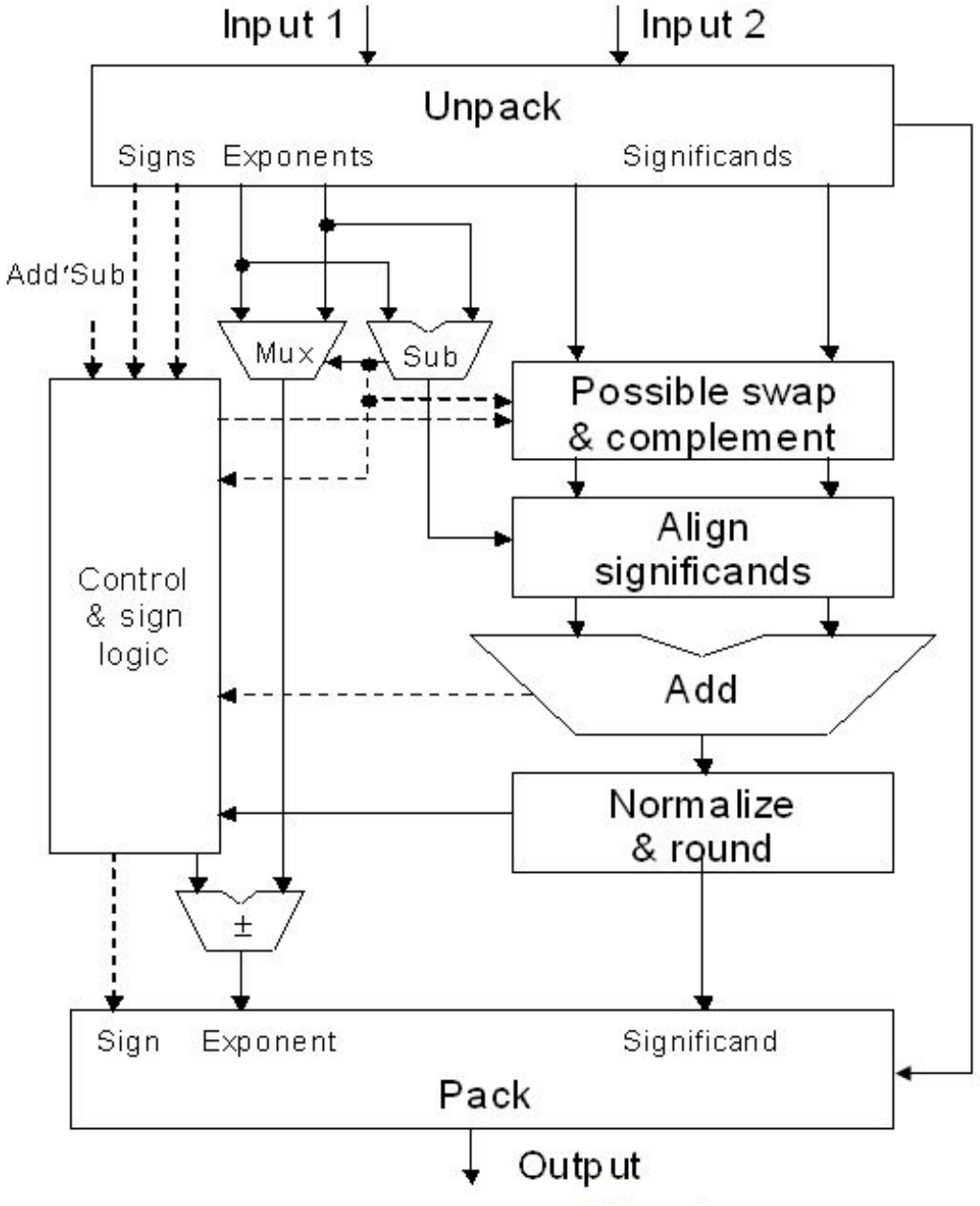


Figure 3.13: Graphical description for ADD\SUB Floating Point numbers entity.

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

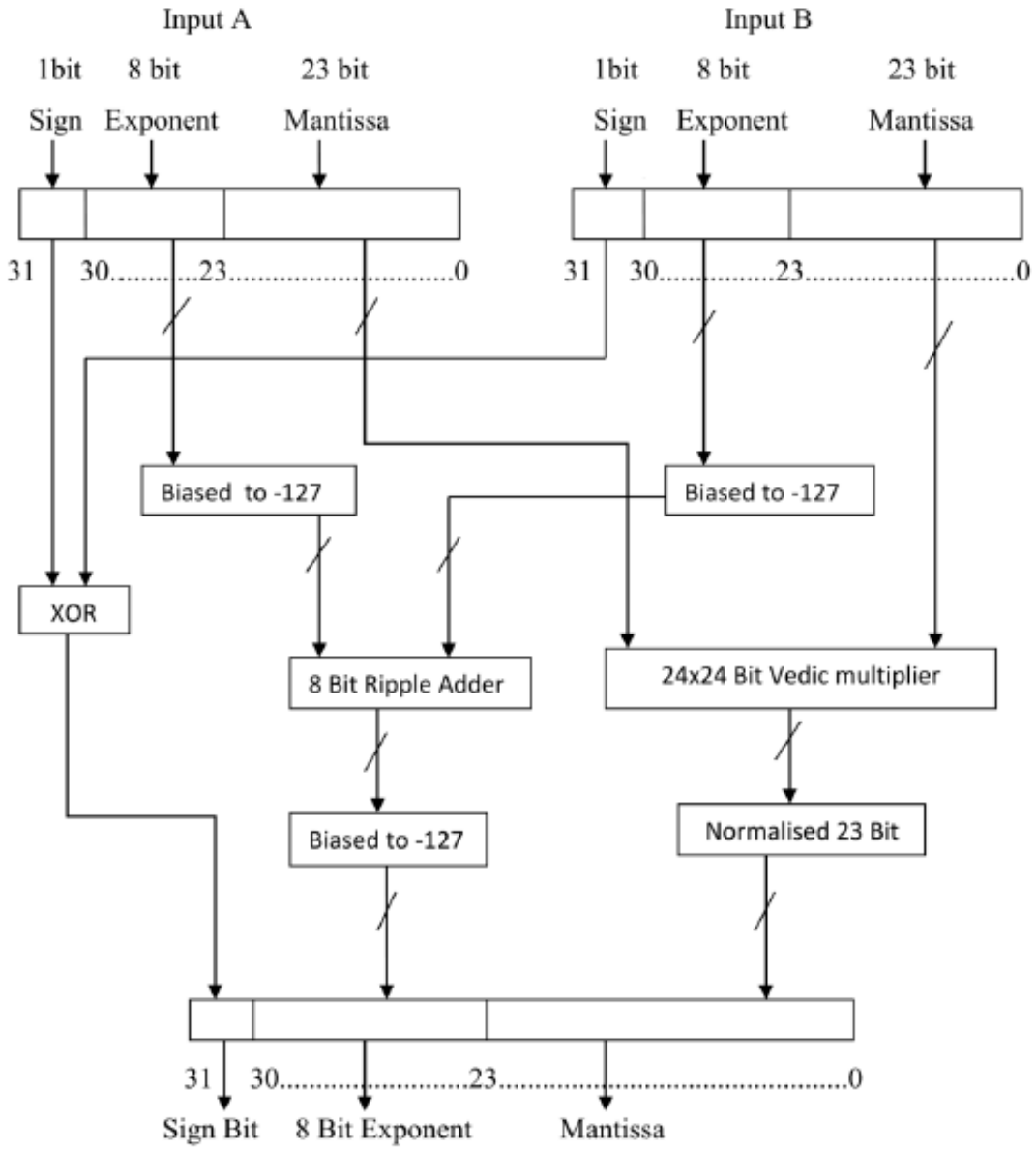


Figure 3.14: Graphical description for: MUL Floating Point numbers entity.

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

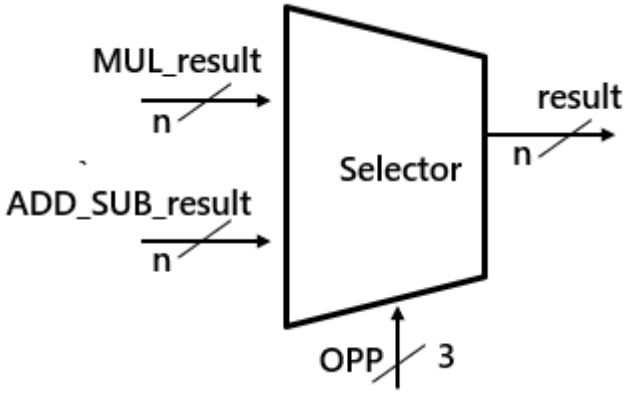


Figure 3.15: Graphical description for: FPU output selector entity.

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

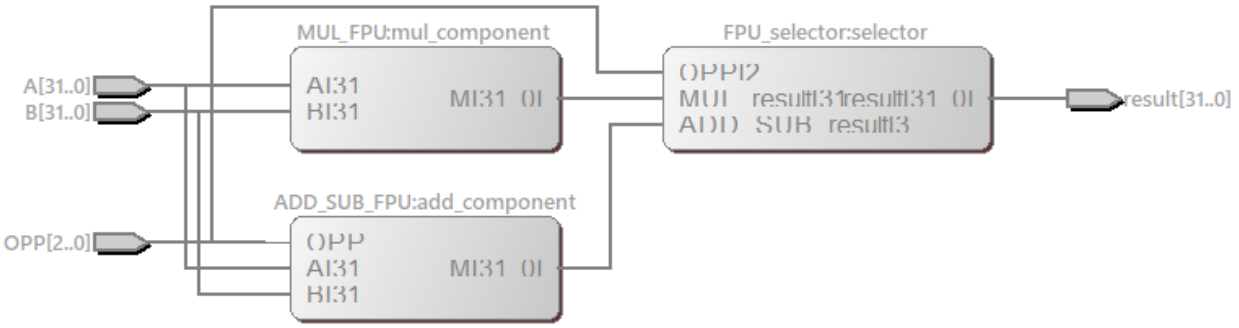


Figure 3.16: Graphical description for FPU top design unit entity.

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)*	*leee 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

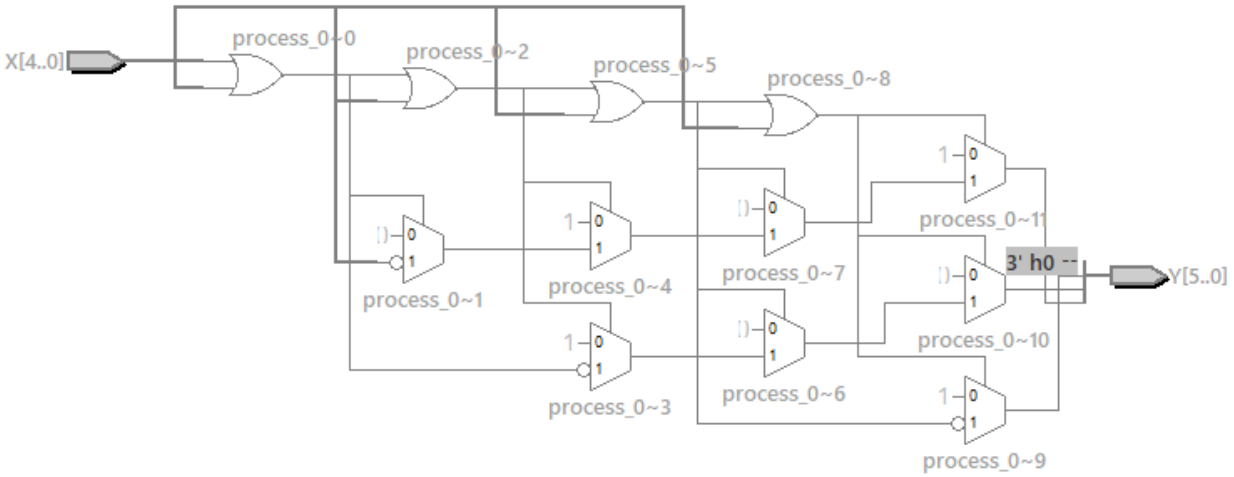


Figure 3.17: Graphical description for LeadingZeros counter entity.

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

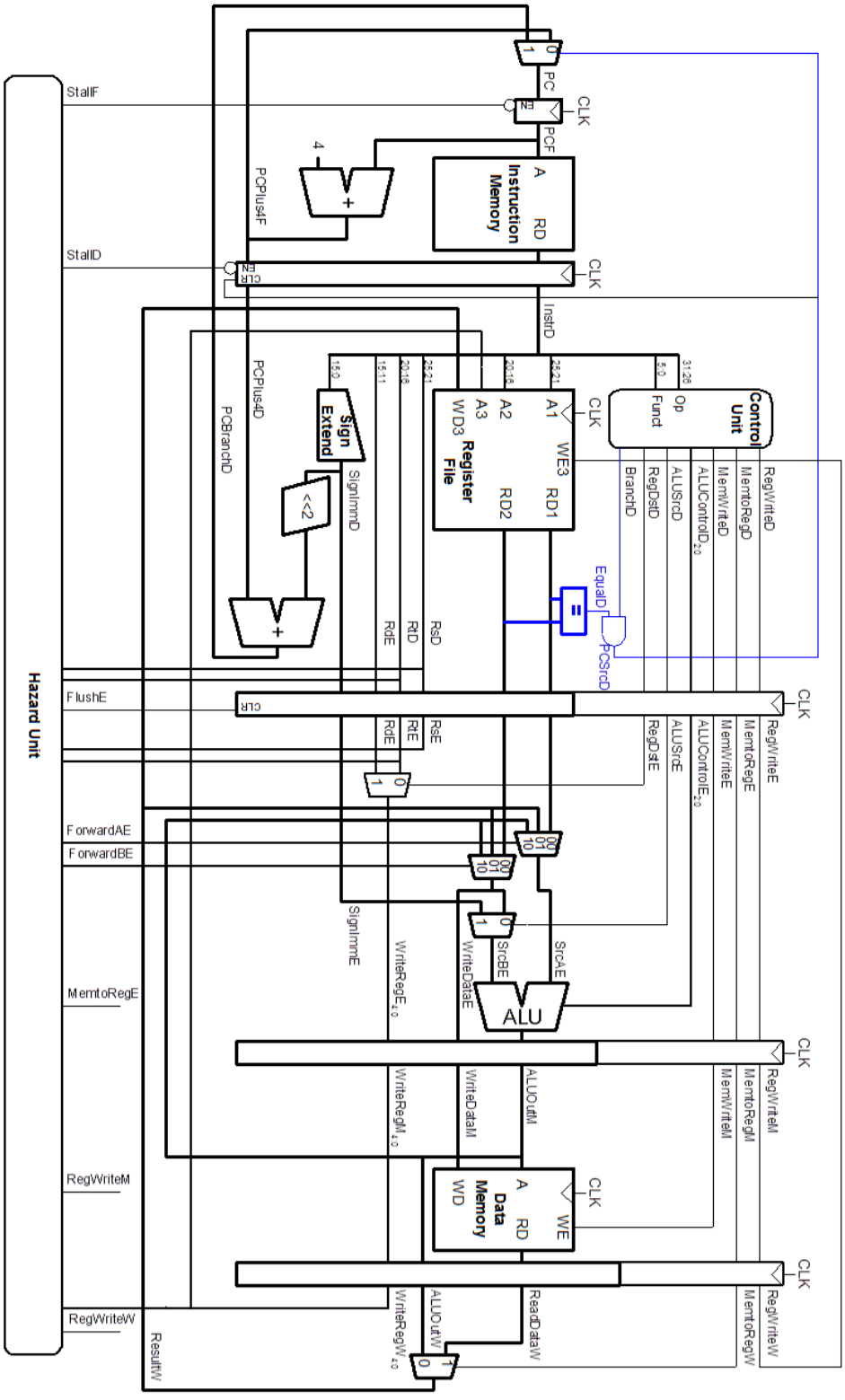


Figure 3.18: Graphical description for pipelined MIPS entity.

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

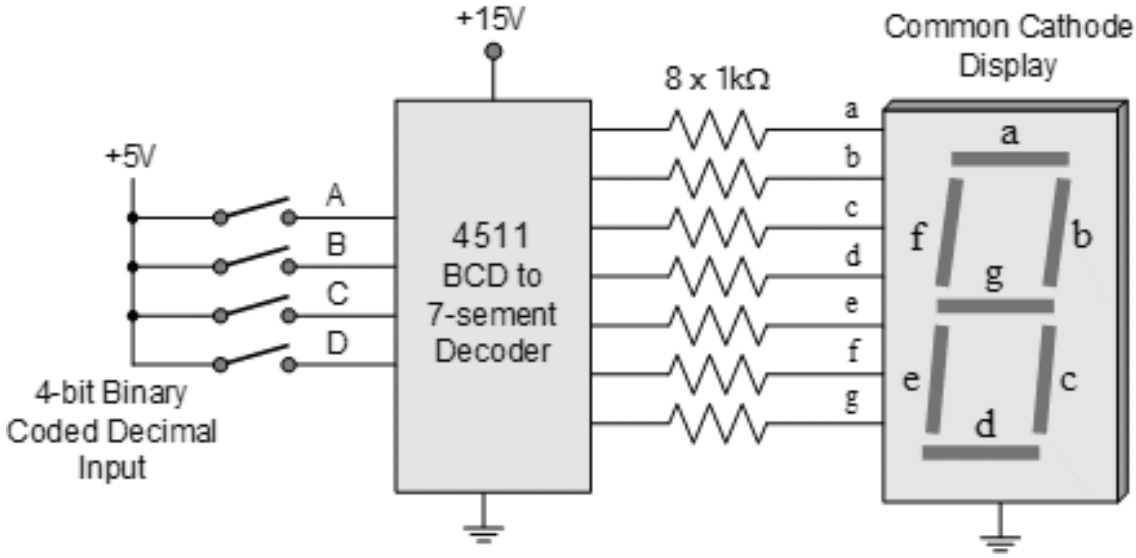


Figure 3.19: Graphical description for: Hex to 7-Segment entity.

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.

3.20 RWA2RGB

File name: RWA2RGB.vhd

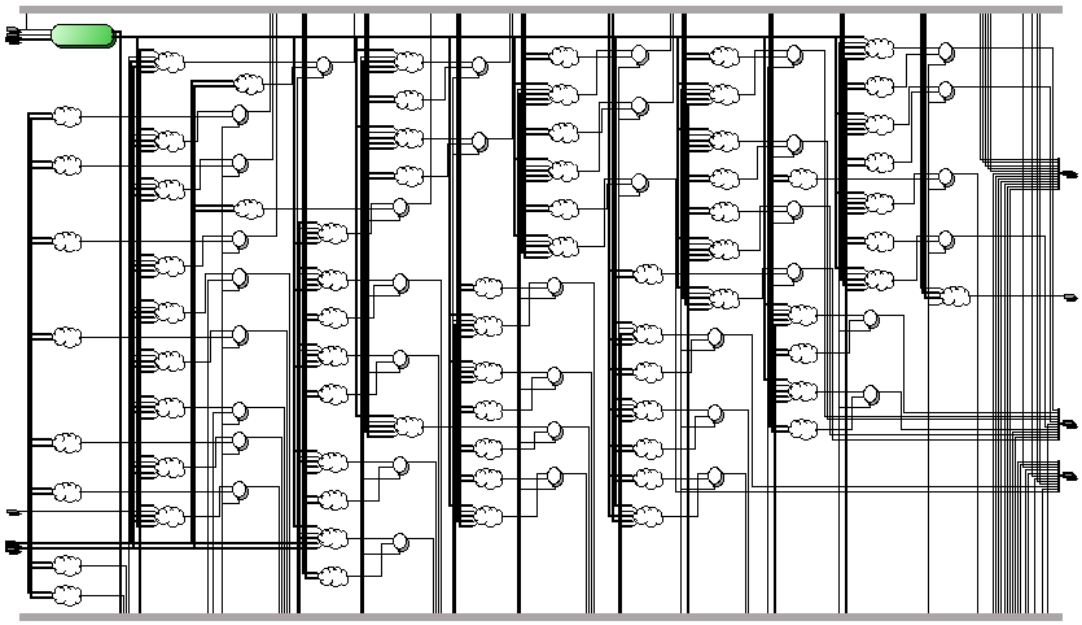


Figure 3.20.1: Graphical description for entity.

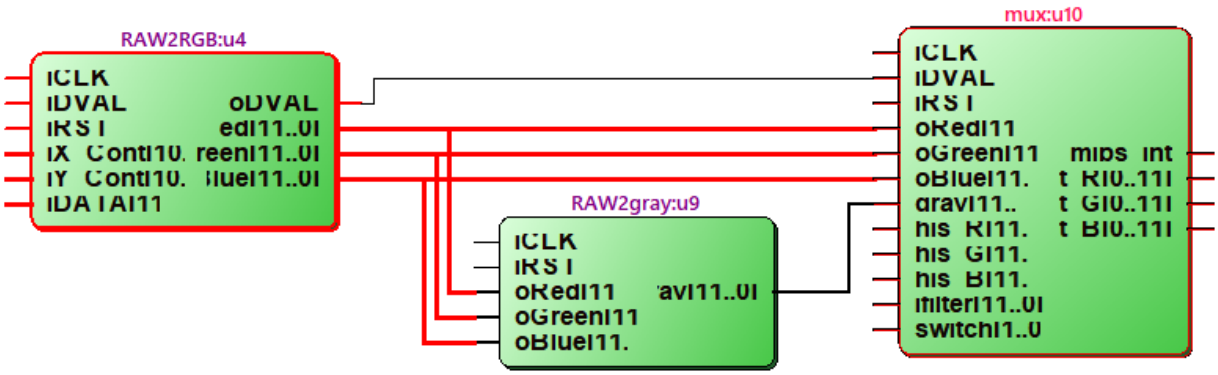


Figure 3.20.2: Graphical description for entity.

Port name	direction	type & size	functionality
oRed	Output	Std_logic 12 bits	Output of red
oGreen	Output	Std_logic 12 bits	Output of green
oBLue	Output	Std_logic 12 bits	Output of blue

oDVAL	Output	Std_logic 1 bit	Output of Data value
iX_Count	input	Std_logic 11 bits	Counter of X row
iY_Count	input	Std_logic 11 bits	Counter of Y column
iDATA	input	Std_logic 12 bits	Input data
mode_sel	input	Std_logic 2 bits	Selector of mode switch (4,3)
iDVAL	input	Std_logic 1 bit	Input data value
iCLK	input	Std_logic 1 bit	Input clock
iRST	input	Std_logic 1 bit	Input reset

Table 3.20: Port Table for: RWA2RGB entity.

Description: This entitiy get the pixel DATA and the location on the 2D plot. The first stage is the grayscale conversion according to this formula :

$$Gray = Red * 0.299 + Green * 0.587 + Blue * 0.114$$

$$Gray \in [0,4095].$$

Then, we compute the histogram according to the Image processing section. We choose $k = 10$ categories of grayscale and each pixel Associated to the right range :

$$Choosen\ categorie\ i = i * 255 \leq grayscale < 255(i+1)$$

Finally, we loop through the histogram array and for each pixel location we output black pixel/white pixel according to :

$$for\ i = 0\ to\ 9:$$

$$pixel = (x, y),\ \ i \in [0,9]$$

$$black\ pixel\ = x\ cordinate : 63 * i \leq x < 63(i+1)$$

$$y\ cordinate : 640 * (480 - y) < histogram(i)$$

3.21 Histogram

File name: histogram.vhd

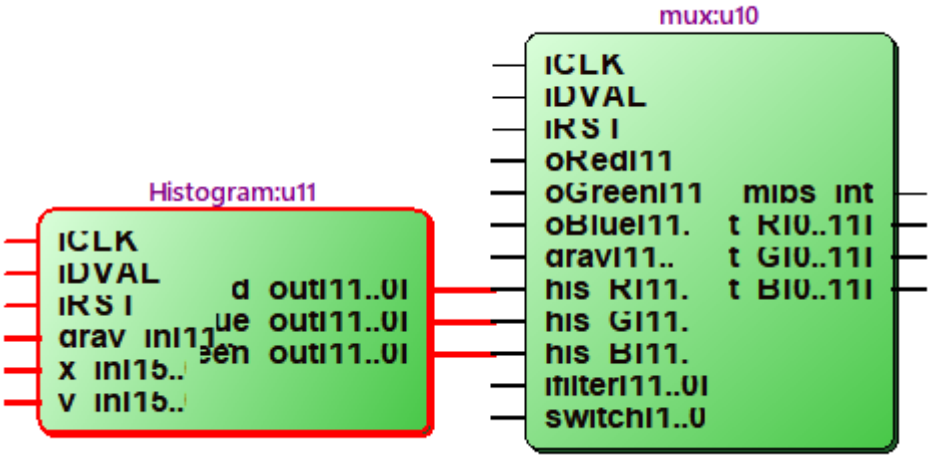


Figure 3.21 : Graphical description for Histogram entity.

Port name	direction	type & size	functionality
Gray_in	*in*	*std_logic (12 bits)*	*Gray scale value*
X_in	*in*	*std_logic (16 bits)*	*Pixel index x*
Y_in	*in*	*std_logic 1(6 bits)*	*Pixel index y*
Idval_in	*in*	*std_logic*	*Idva bit*
iclk_in	*in*	*std_logic*	*Clock bit*
irst_in	*in*	*std_logic*	*Reset bit*
Red_out	*Out*	*std_logic (12 bits)*	*VGA red value*
Blue_out	*Out*	*std_logic (12 bits)*	*VGA blue value*
Green_out	*out*	*std_logic (12 bits)*	*VGA green value*

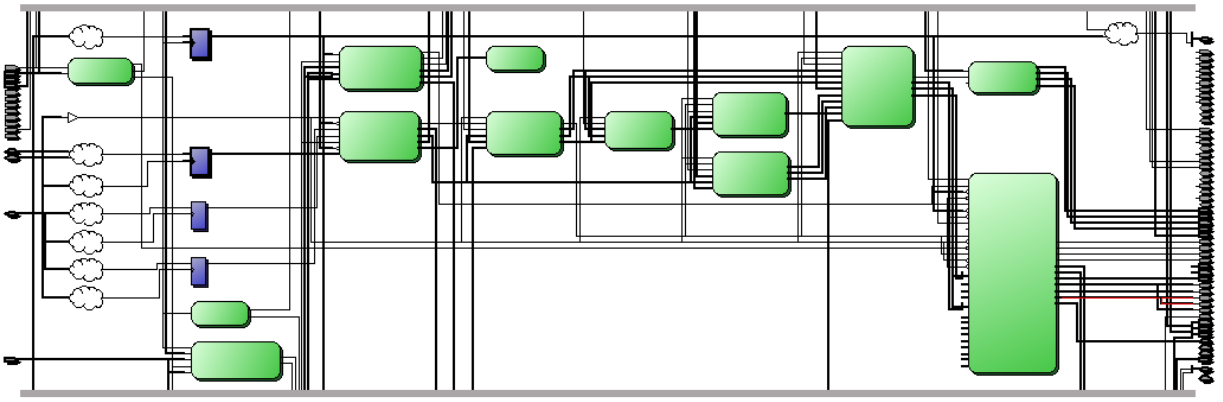
Table 3.21: Port Table for: entity.

Description: Histogram, with 16 categories of grayscale.

Analyzing

4.1 overall system

File name: DE1_D5.vhd



Full Image in DOC/TOP.png

Flow Summary	
Flow Status	Successful - Thu Jun 21 17:14:14 2018
Quartus II 32-bit Version	12.1 Build 177 11/07/2012 SJ Web Edition
Revision Name	DE1_D5M
Top-level Entity Name	DE1_D5M
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	9,550 / 18,752 (51 %)
Total combinational functions	9,140 / 18,752 (49 %)
Dedicated logic registers	2,053 / 18,752 (11 %)
Total registers	2067
Total pins	301 / 315 (96 %)
Total virtual pins	0
Total memory bits	133,488 / 239,616 (56 %)
Embedded Multiplier 9-bit elements	9 / 52 (17 %)
Total PLLs	2 / 4 (50 %)

Figures 4.1.2: Flow Summary for: overall system entity.

Analysis & Synthesis Resource Usage Summary		
	Resource	Usage
1	Estimated Total logic elements	9,662
2		
3	Total combinational functions	9131
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	4835
2	-- 3 input functions	2299
3	-- <=2 input functions	1997
5		
6	Logic elements by mode	
1	-- normal mode	7332
2	-- arithmetic mode	1799
7		
8	Total registers	2099
1	-- Dedicated logic registers	2099
2	-- I/O registers	0
9		
10	I/O pins	301
11	Total memory bits	133488
12	Embedded Multiplier 9-bit elements	9
13	Total PLLs	2
1	-- PLLs	2
14		
15	Maximum fan-out	1040
16	Total fan-out	40692
17	Average fan-out	3.45

Figures 4.1.2: Logic usage for: overall system entity.

Entity	Dedicated Logic	I/O Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18	Pins	LUT-Only LCs	Register-Only LCs
Cyclone II: EP2C20F484C7									
DE1_D5M	2053 (1)	14 (14)	133488	9	1	4	301	7497 (2)	410 (0)
VGA_Controller:u1	27 (27)	0 (0)	0	0	0	0	0	34 (34)	0 (0)
Reset_Delay:u2	35 (35)	0 (0)	0	0	0	0	0	13 (13)	0 (0)
CCD_Capture:u3	48 (48)	0 (0)	0	0	0	0	0	16 (16)	10 (10)
RAW2RGB:u4	11 (0)	0 (0)	30672	0	0	0	0	142 (137)	0 (0)
SEG7_LUT_8:u5									
sdram_pll:u6	0 (0)	0 (0)	0	0	0	0	0	0 (0)	0 (0)
Sdram_Control_4Por...	665 (130)	0 (0)	22528	0	0	0	0	239 (96)	241 (15)
I2C_CCD_Config:u8	132 (94)	0 (0)	0	0	0	0	0	123 (77)	16 (0)
RAW2gray:u9	0 (0)	0 (0)	0	0	0	0	0	1178 (24)	0 (0)
mux:u10	0 (0)	0 (0)	0	0	0	0	0	70 (70)	0 (0)
Histogram:u11	673 (673)	0 (0)	0	2	0	1	0	472 (472)	16 (16)
entropy_filter:u12	102 (91)	0 (0)	61344	0	0	0	0	3918 (3913)	18 (18)
MIPS:u13	359 (0)	0 (0)	18944	7	1	3	0	1290 (1)	109 (0)

Figures 4.1.2: Logic usage for: each main entity.

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

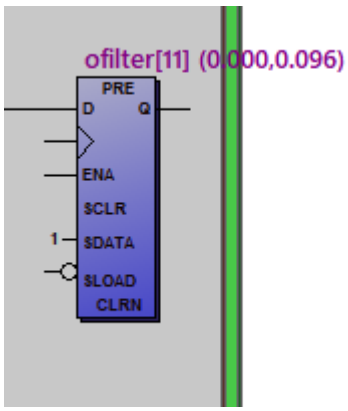


Figure 4.1.5: Critical path for: overall system entity, full-png in DOC.

Analyze: We zoom in on the main components. We can see that the entropy filter is the responsible of the critical path in the system.

Analyze: $DE1_{D5} \rightarrow VGA\ controller \rightarrow RAW2RGB \rightarrow RAW2GRAY \rightarrow entropy\ filter \rightarrow SDRAM\_control$

Frequency limiting operation: Entropy filter, with the linebuffers and matrix assignments.

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

1. The problematic operation is commonly used in software :
   ⇨ Design this unit with dedicated hardware (with separate clock).

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 : (with floating point)

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	7.45 MHz	7.45 MHz	GPIO_1[0]
2	28.79 MHz	28.79 MHz	u13|m1|altpll_component|pll|clk[0]
3	134.7 MHz	134.7 MHz	u6|altpll_component|pll|clk[0]
4	140.98 MHz	140.98 MHz	CLOCK_50
5	141.28 MHz	141.28 MHz	rClk[0]
6	167.06 MHz	167.06 MHz	MIPS:u13|Idecode:ID|counter:Counter_
7	220.95 MHz	220.95 MHz	I2C_CCD_Config:u8|mI2C_CTRL_CLK

Figure 4.1.4: Fmax Summary

Maximal operating clock for mips only :

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	30.32 MHz	30.32 MHz	m1|altpll_component|pll|clk[0]

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

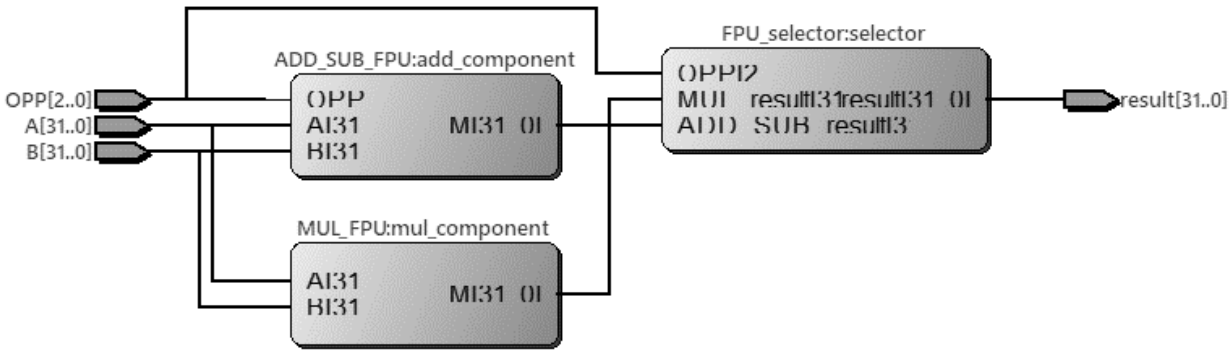


Figure 4.2.1: RTL Viewer for: FPU top design entity.

4.4 Floating Point Adder

File name: ADD_SUB_FPU.vhd

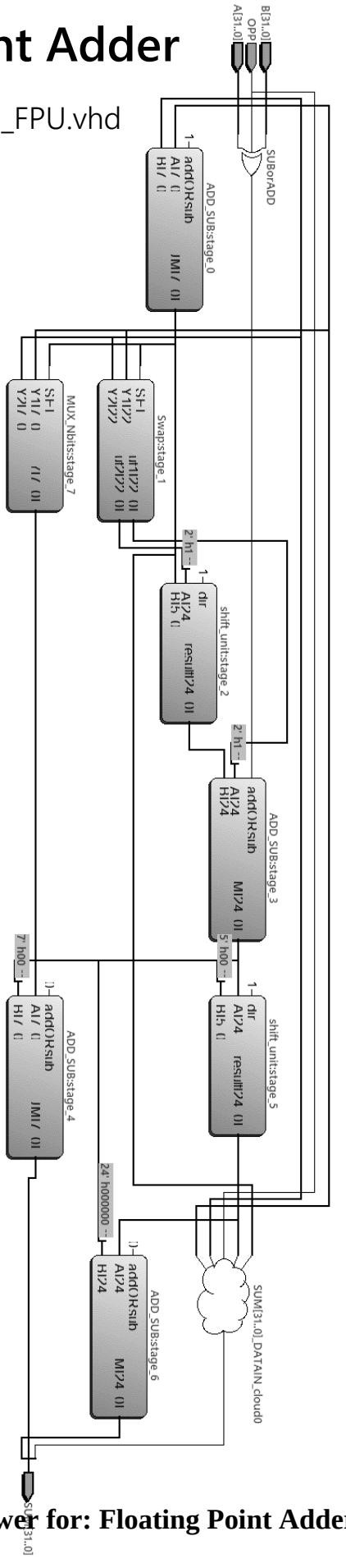


Figure 4.4.1: RTL Viewer for: Floating Point Adder entity.

4.5 Floating Point Multiplier

File name: MUL_FPU.vhd

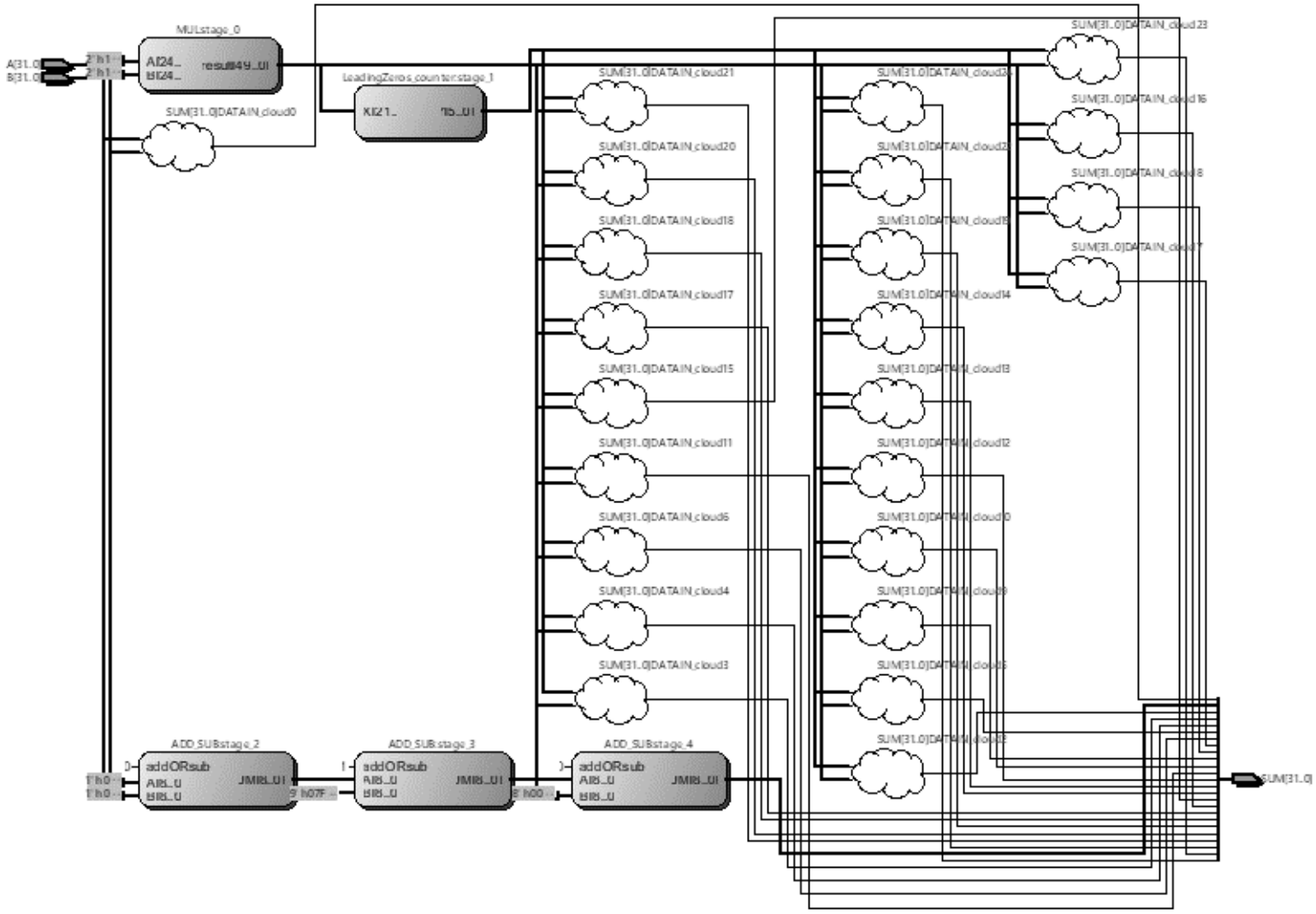


Figure 4.5.1: RTL Viewer for: Floating Point Multiplier entity.

4.6 Fetch

File name: IFETCH.vhd

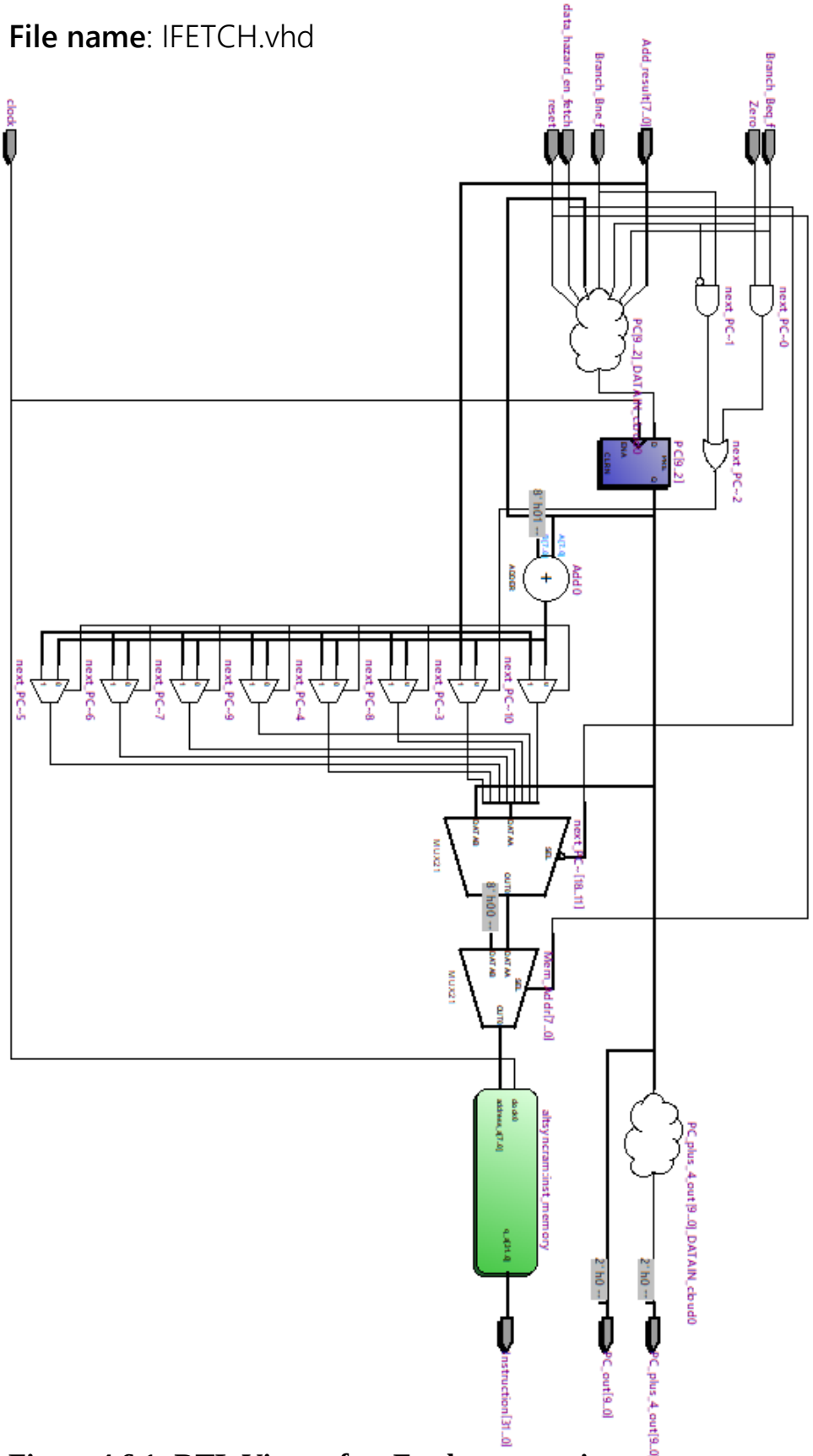


Figure 4.6.1: RTL Viewer for: Fetch stage entity.

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

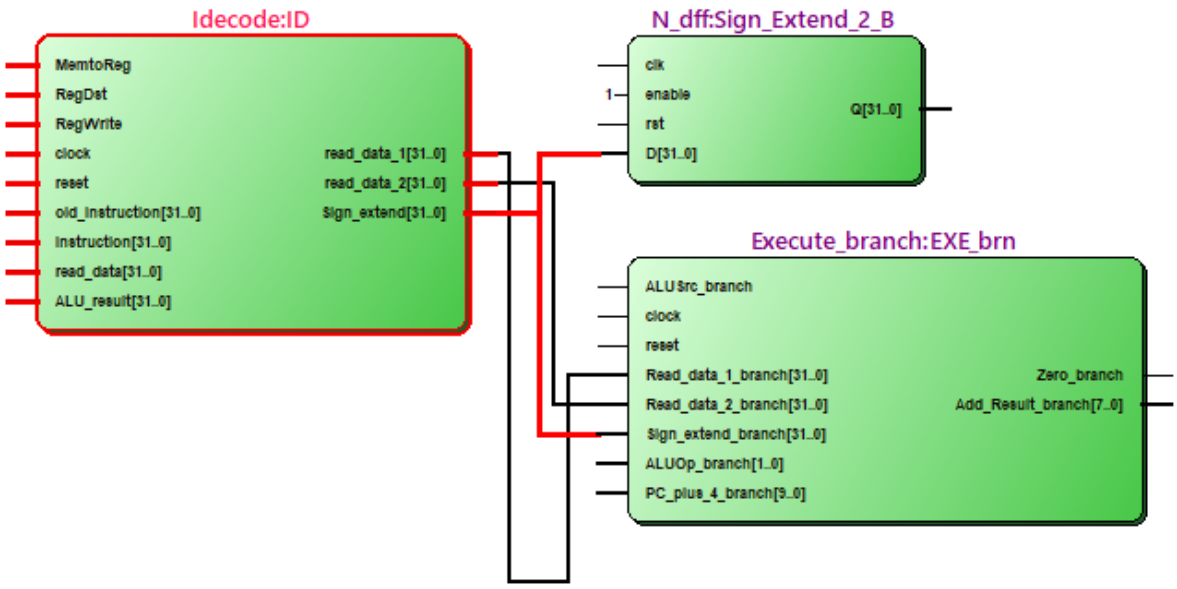


Figure 4.7.1: RTL Viewer for: Decode stage entity.

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

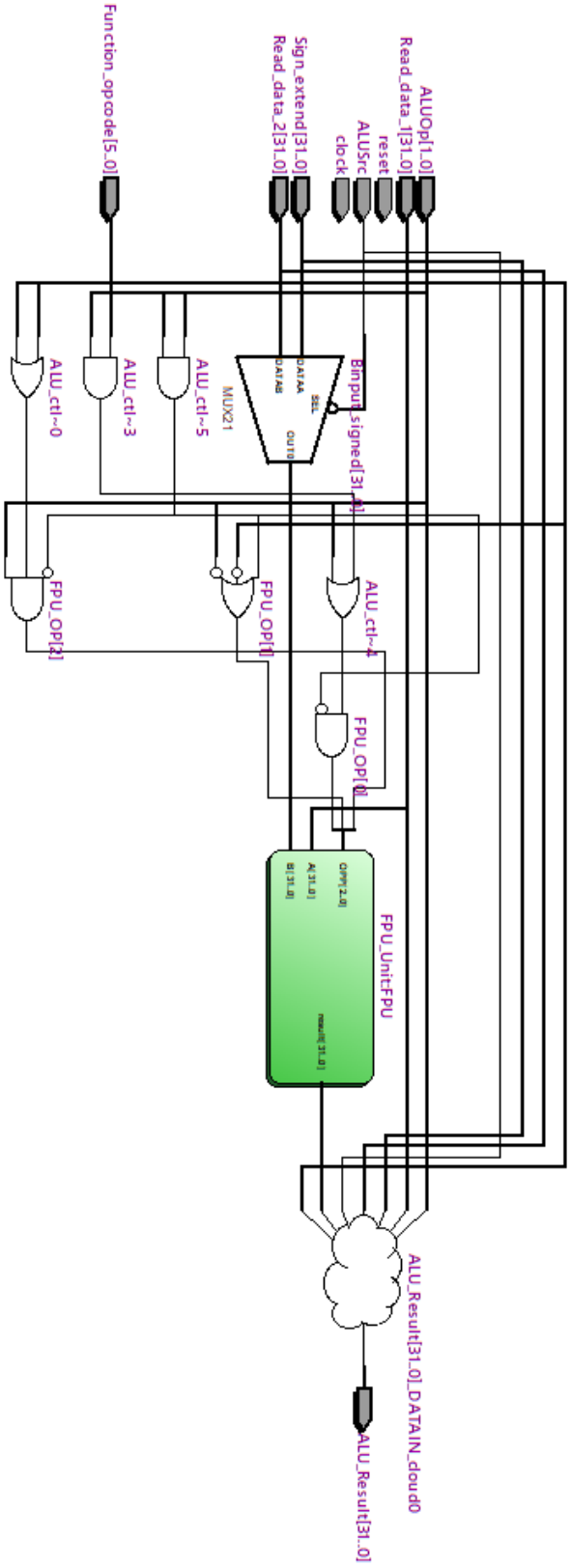


Figure 4.8.1: RTL Viewer for: Execute stage entity.

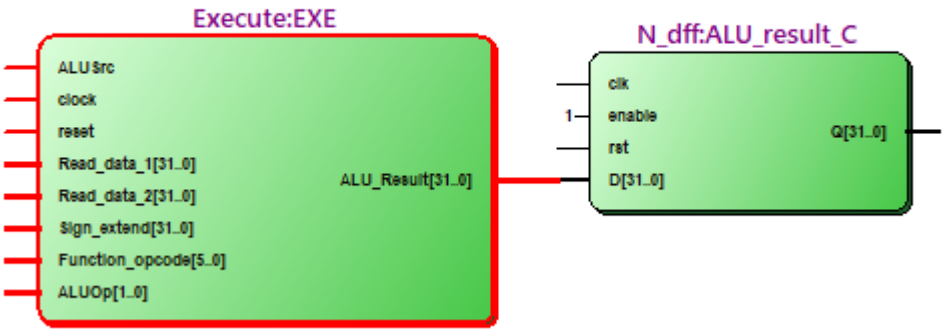


Figure 4.8.2: RTL Viewer for: Execute stage entity.

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

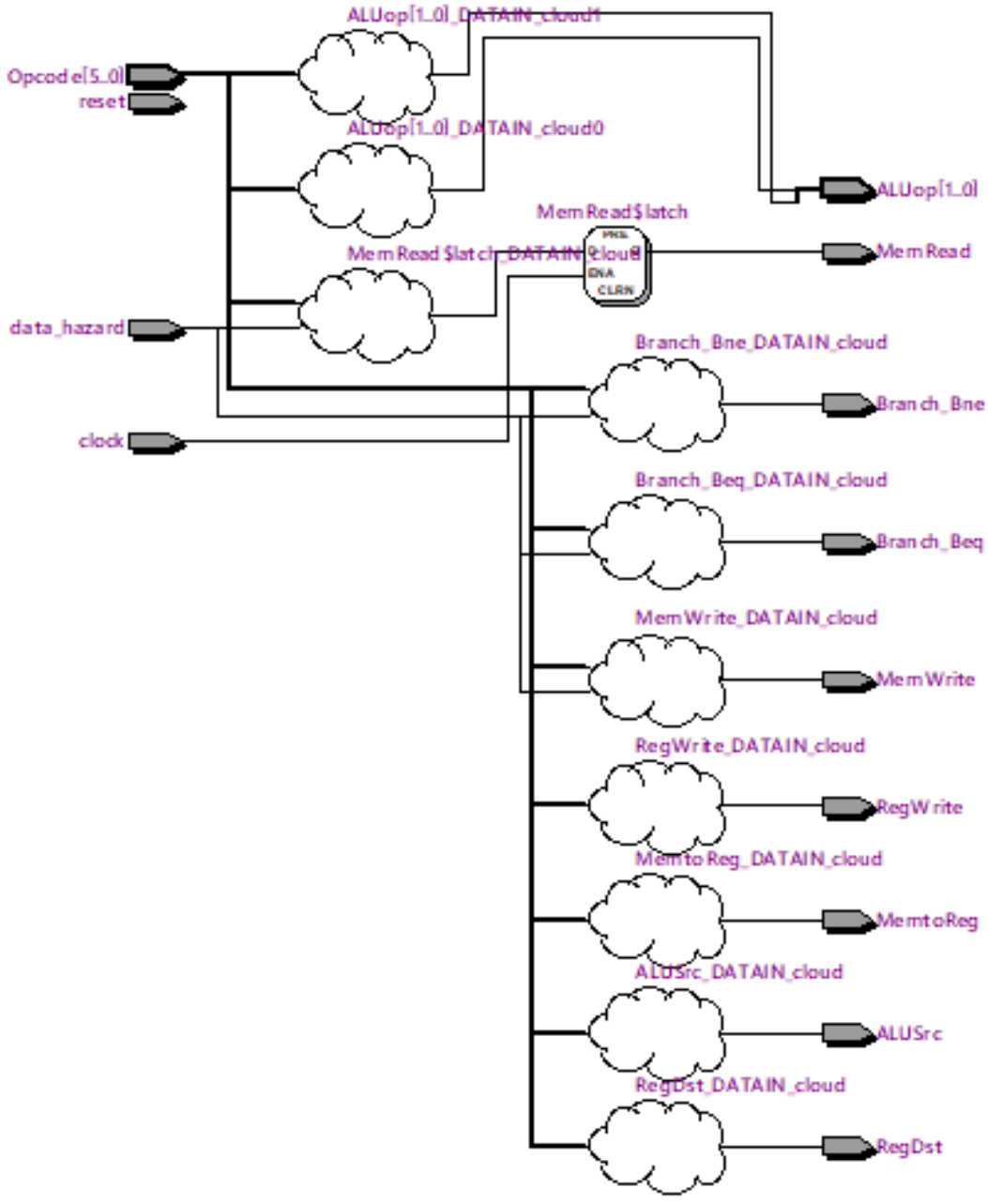


Figure 4.9.1: RTL Viewer for: Control stage entity.

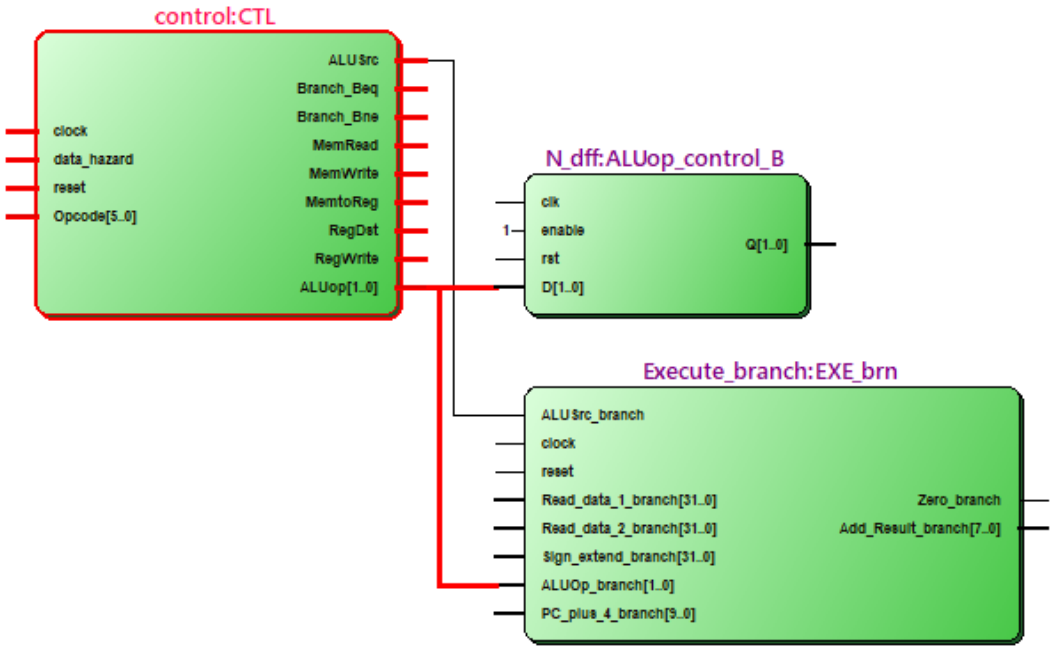


Figure 4.9.2: RTL Viewer for: Control stage entity.

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

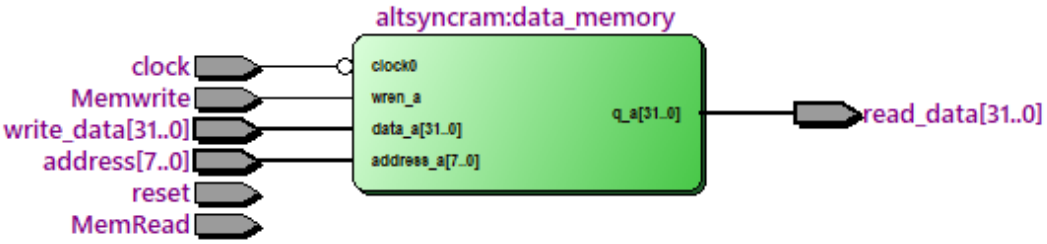


Figure 4.10.1: RTL Viewer for: W\R memory stage entity.

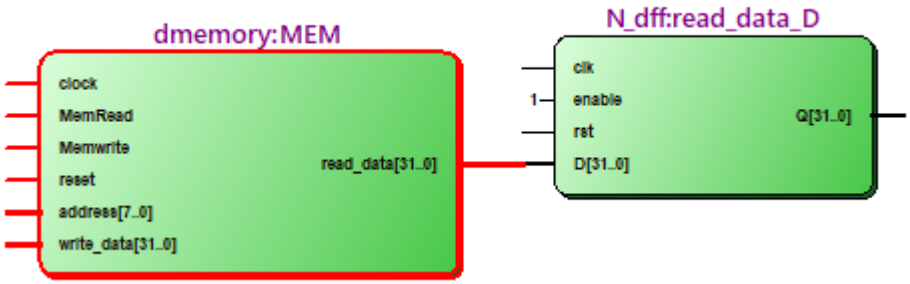


Figure 4.10.2: RTL Viewer for: W\R memory stage entity

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

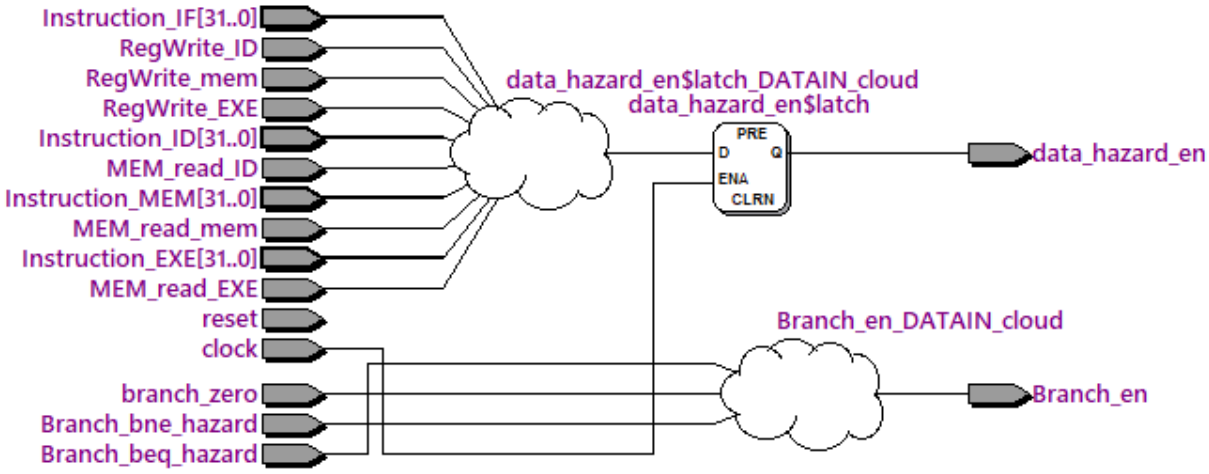


Figure 4.11.1: RTL Viewer for: Hazard unit entity.

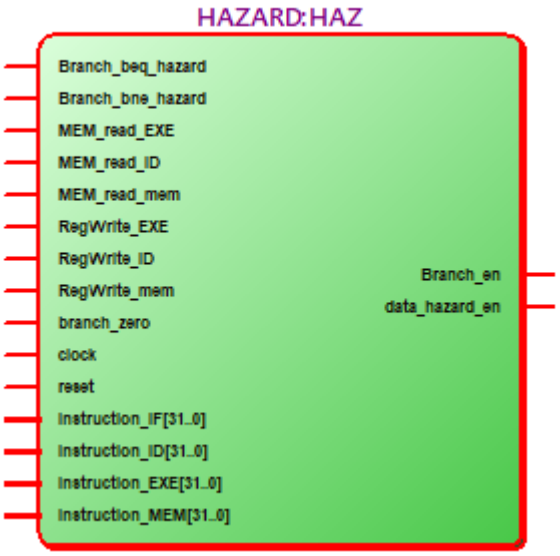


Figure 4.11.2: RTL Viewer for: Hazard unit entity.

Example: Data Hazard that required stalling

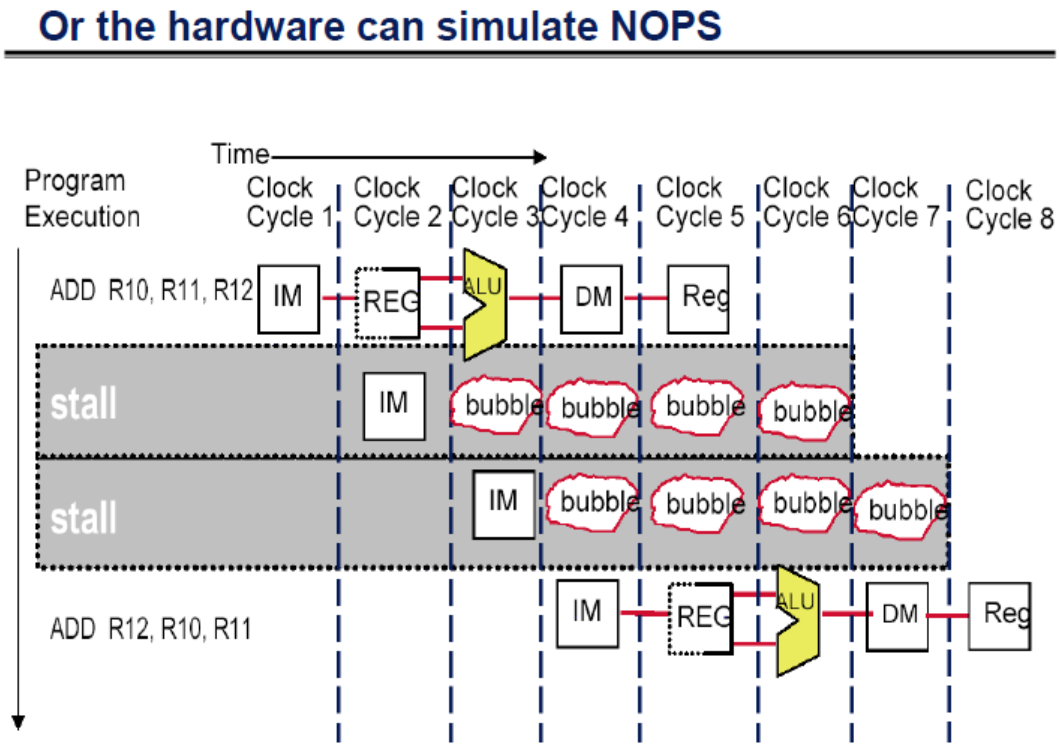


Figure 4.11.2: Data hazard example

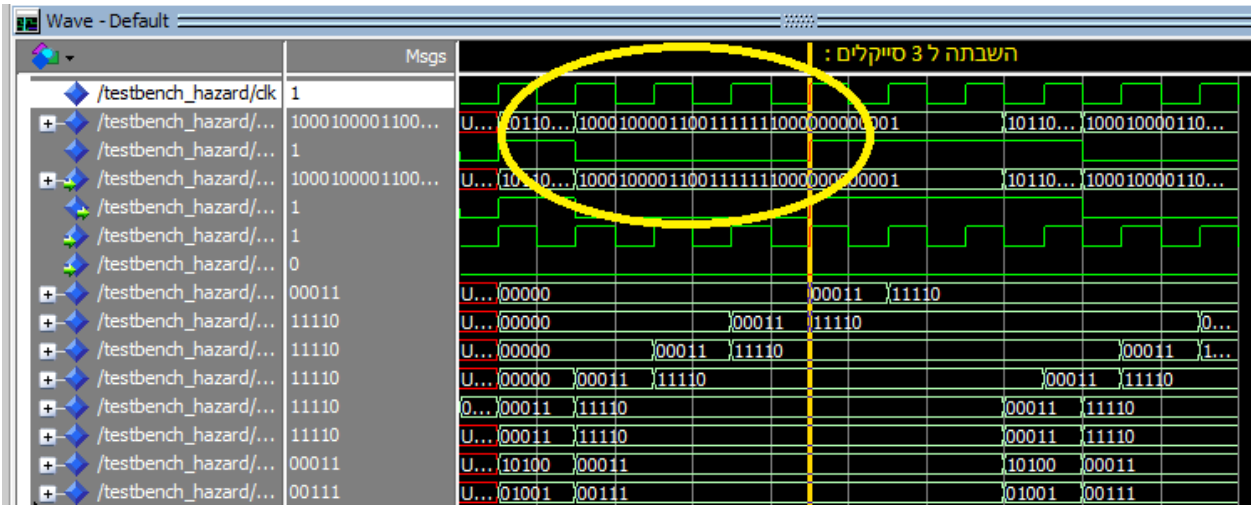


Figure 4.11.2: Hazard unit stalling

4.12 Execute Branch

File name: Execute_branch.vhd

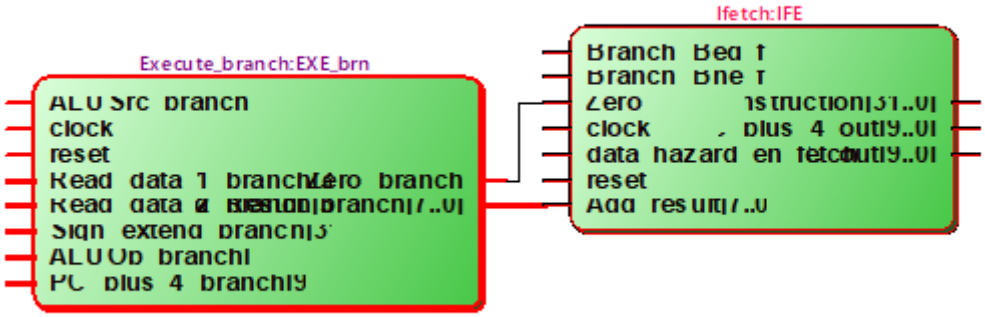


Figure 4.12.1: RTL Viewer for: Execute Branch entity.

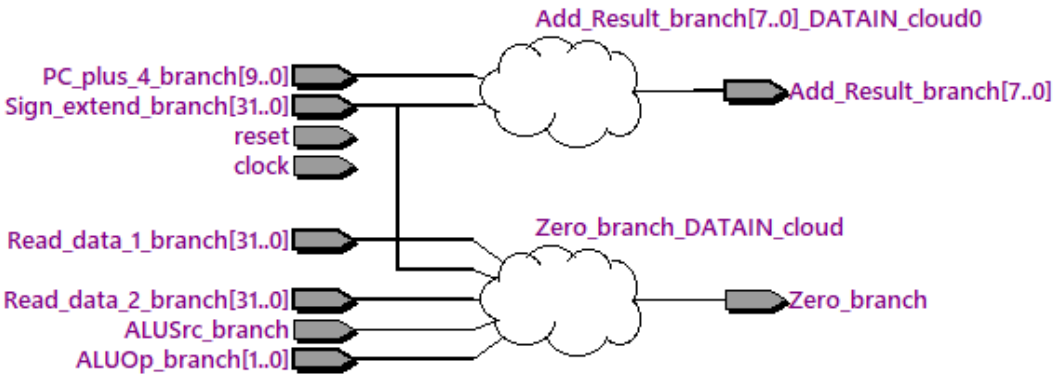
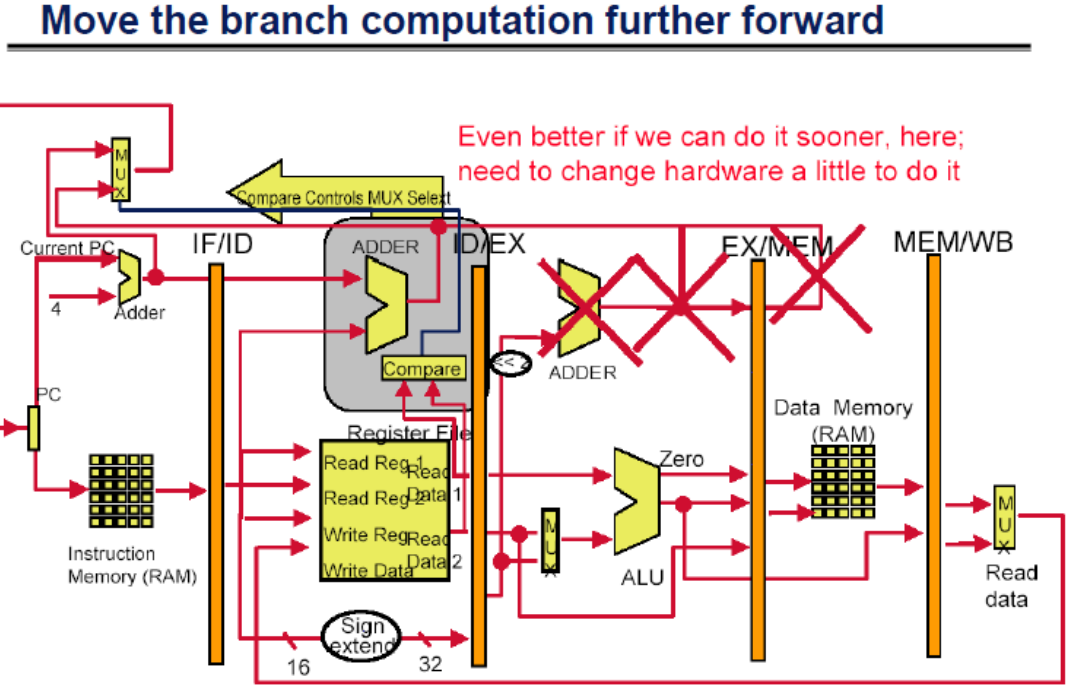


Figure 4.12.2: RTL Viewer for: Execute Branch entity.

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



Filters demo

Grayscale

Figure 5.1.1: Before grayscaling

Figure 5.1.2: After grayscaling

Histogram

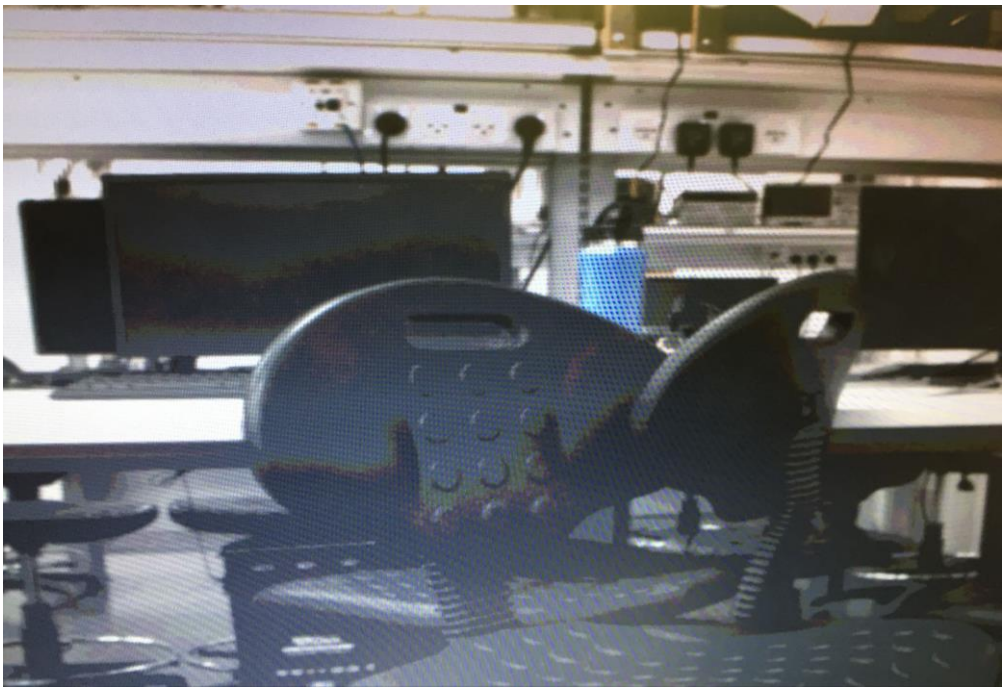


Figure 5.2.1: current frame

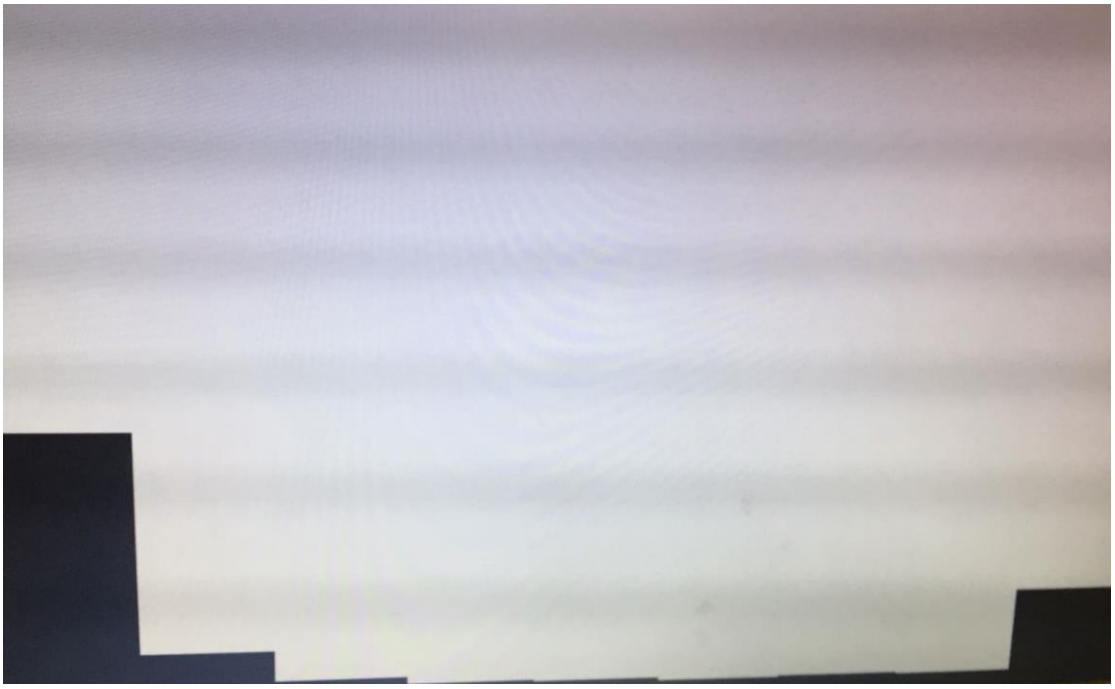


Figure 5.2.2: Histogram

Prewitt operator

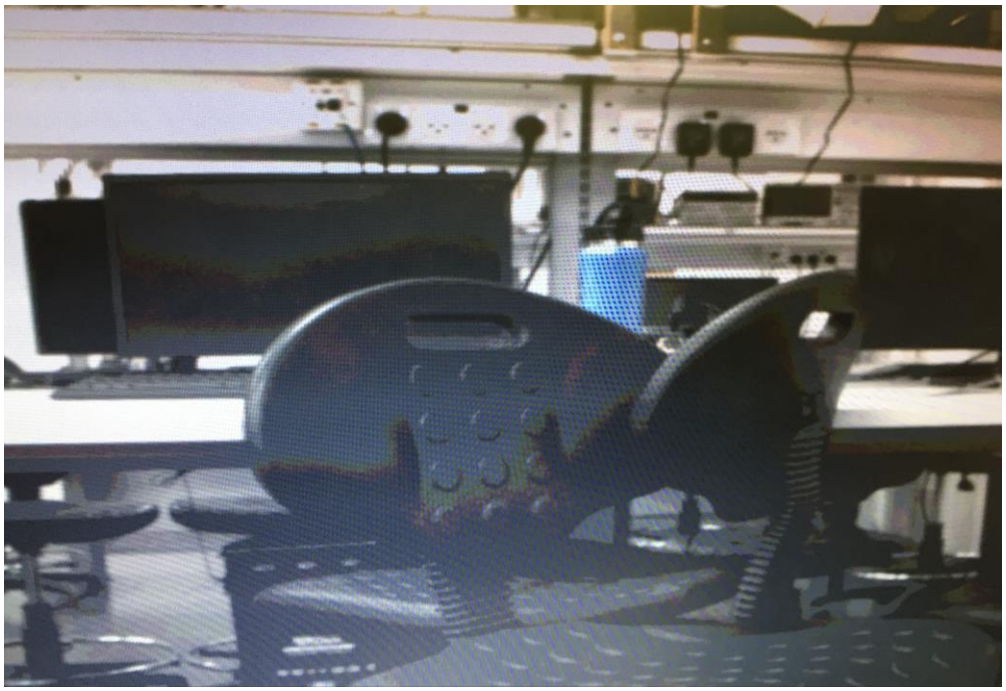


Figure 5.3.1: current frame

Figure 5.3.2: After Prewitt filter

Entropy filter

Figure 5.3.1: current frame

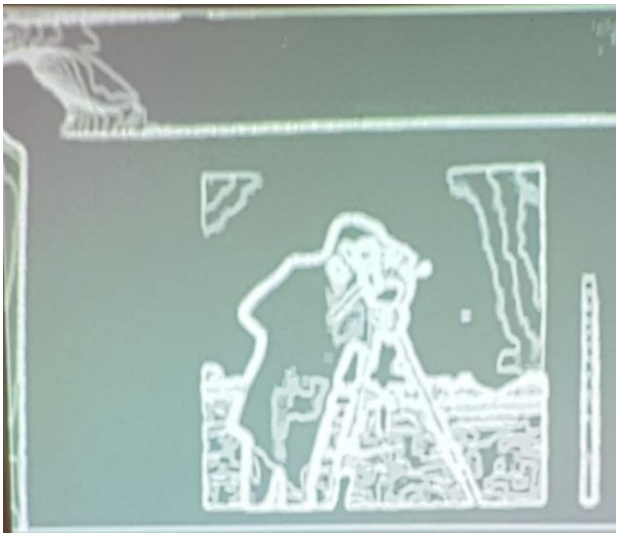


Figure 5.3.2: After entropy filter

Test Bench's

6.1 full_adder.VHD

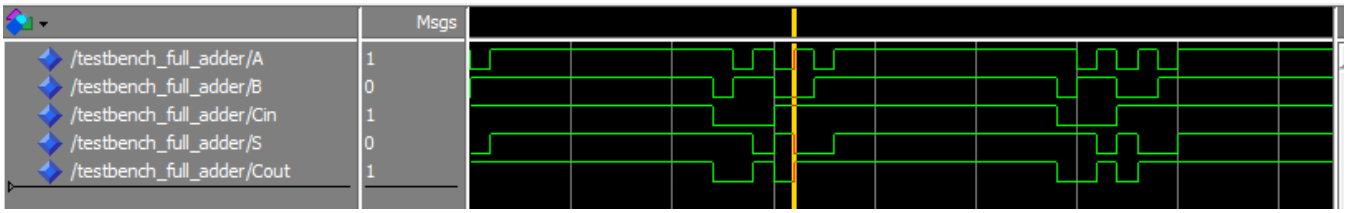


Figure 6.1: Test Bench for: full_adder entity

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

6.2 ADD_SUB.VHD

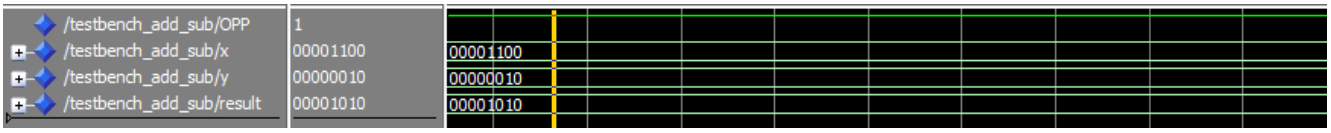


Figure 6.2: Test Bench for: ADD_SUB operation

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

6.3 ADD.VHD

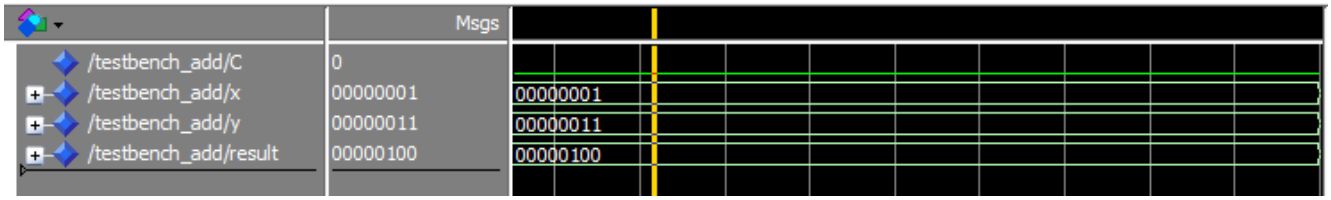


Figure 6.3: Test Bench for: ADD entity

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

6.4 MAX_MIN.VHD

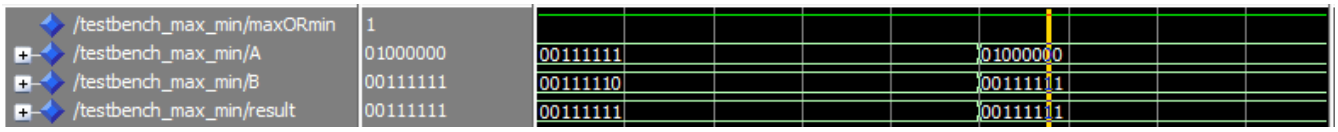


Figure 6.4: Test Bench for: MAX_MIN operation

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

6.5 MUL.VHD

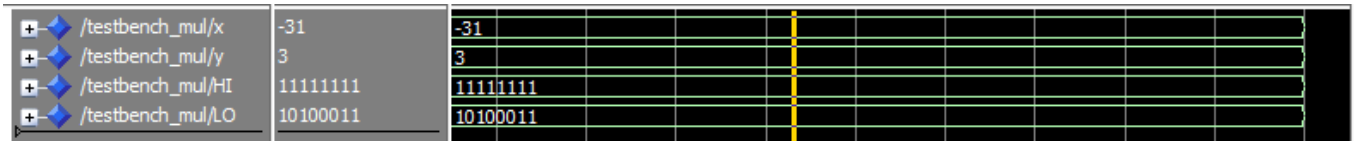


Figure 6.5: Test Bench for: MUL entity

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

6.6 diff_1bit.VHD

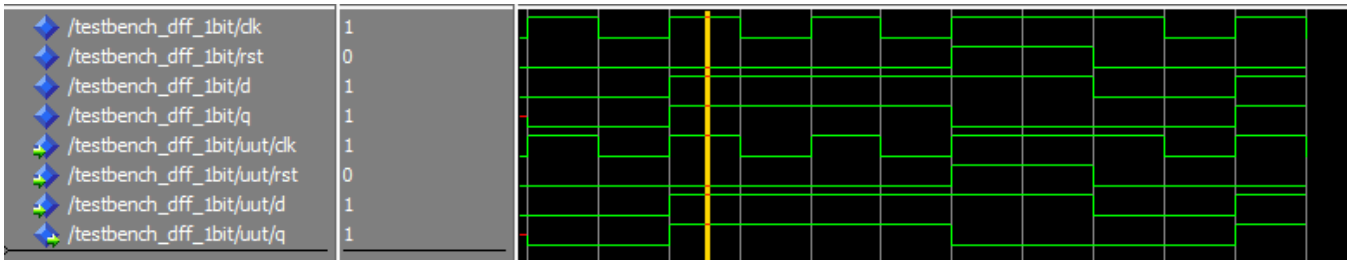


Figure 6.6: Test Bench for: diff_1bit entity

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

6.7 N_dff.VHD

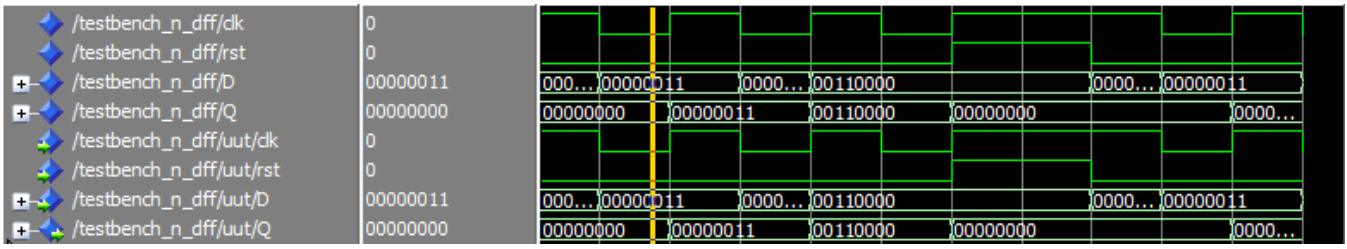


Figure 6.7: Test Bench for: N_dff entity

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

6.8 mux_Nbits.VHD

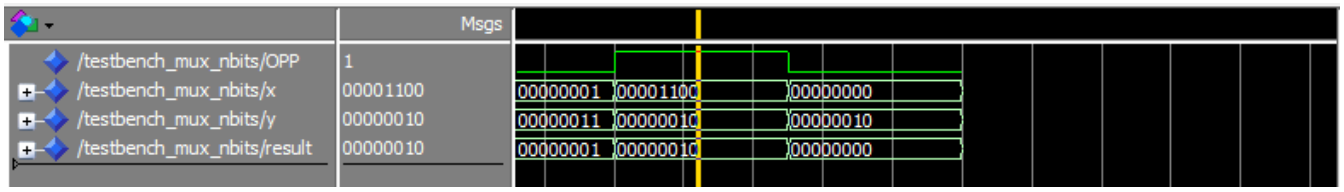


Figure 6.8: Test Bench for: mux_Nbits entity

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

6.9 shift_Nbits.VHD

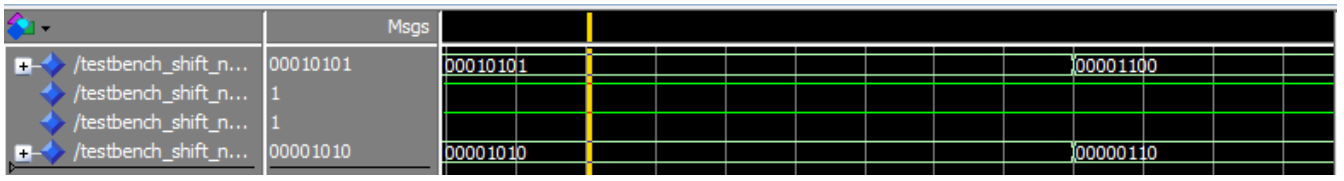


Figure 6.9: Test Bench for: shift_Nbits entity

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

6.10 ADD_SUB_FPU.VHD

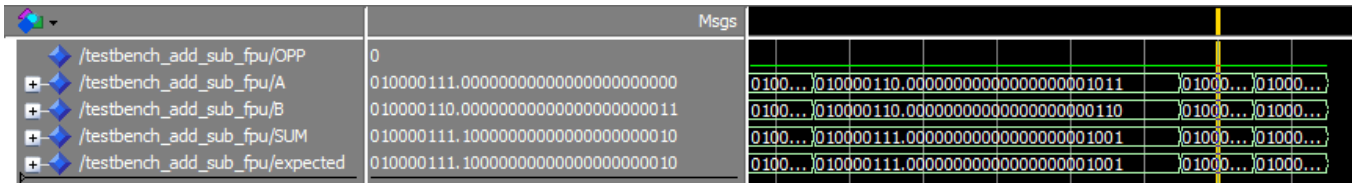


Figure 6.10: Test Bench for: ADD\SUB FPU entity

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

6.11 MUL_FPU.VHD

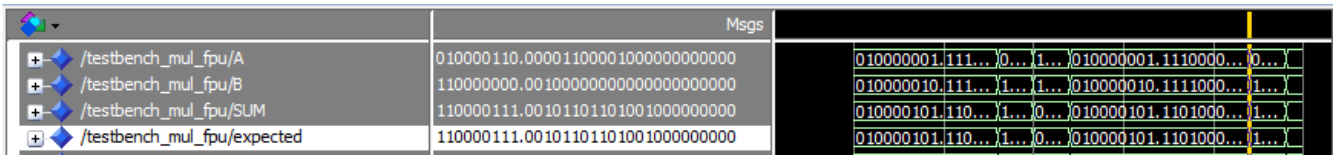


Figure 6.11: Test Bench for: MUL FPU entity

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

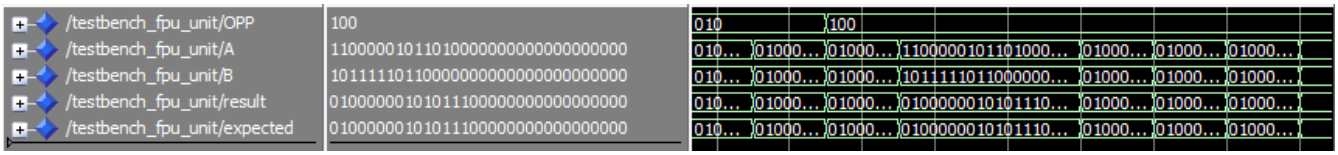


Figure 6.13: Test Bench for: FPU_Unit entity

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

6.14 shift_Unit.VHD

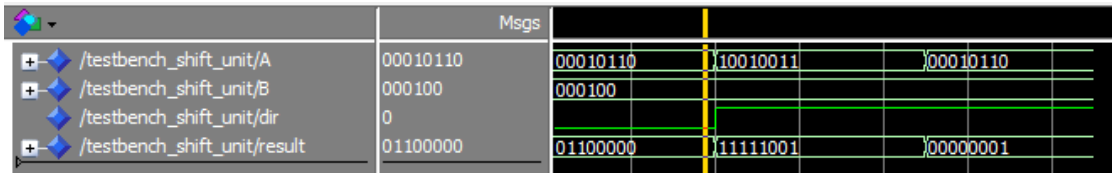


Figure 6.14: Test Bench for: shift_Unit entity

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

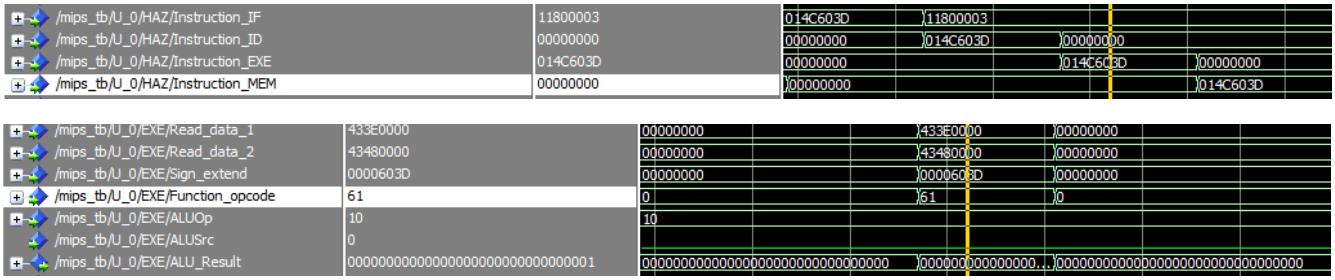


Figure 6.15: Test Bench for: MIPS

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

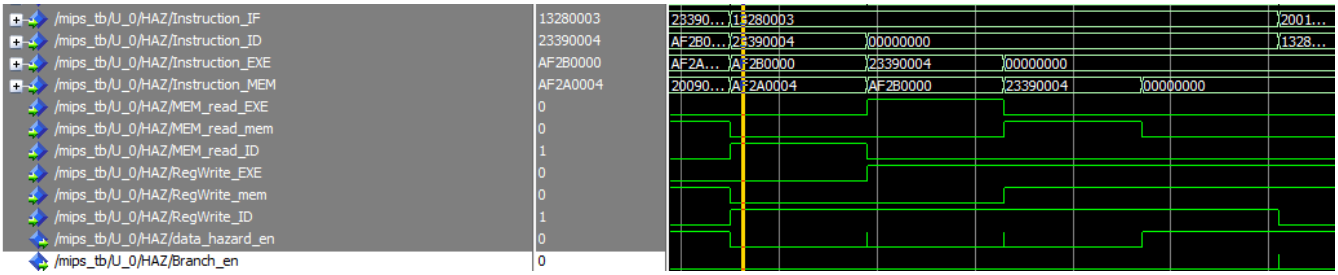


Figure 6.16: Test Bench for: HAZARD

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