**IMPLEMENTATION OF CUSTOM PROCESSOR IN VERILOG FOR IMAGE PROCESSING**

By

S.Vijay Vighnesh 21ECB0B45

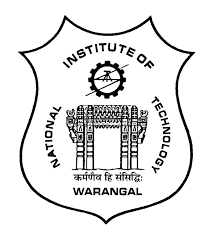
Shashank Viswas Damle 21ECB0B46

Leela Krishna Prasad 21ECB0B47

Guided by:

**Dr. P. Prithvi Dr V. Narendra**

**Assistant Professor - ECE Assistant Professor - ECE**



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL**

**2022-2023**

**CERTIFICATE**

This is to certify that the dissertation work **IMPLEMENTATION OF CUSTOM PROCESSOR IN VERILOG FOR IMAGE PROCESSING** is a bonafide record of work carried out work by S.Vijay Vighnesh (21ECB0B45), Shashank Viswas Damle (21ECB0B46) and Leela Krishna Prasad (21ECB0B47) submitted to faculty of “Electronics and Communication Engineering Department”, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in “Electronics and Communication Engineering” at National Institute of Technology, Warangal during the academic year (2022-2023).

**Dr. P. Prithvi Dr. V. Narendra**

Assistant Professor Assistant Professor

Department of Electronics and Department of Electronics and

Communication Engineering Communication Engineering

National Institute of Technology National Institute of Technology

Warangal Warangal

**ACKNOWLEDGEMENT**

I would like to express my deepest gratitude to my faculty in-charge **Dr. P. Prithvi,** **Assistant Professor**, Department of Electronics and Communication Engineering and **Dr. V. Narendar**, **Assistant Professor**, Department of Electronics and Communication Engineering, National Institute of Technology, Warangal for their constant supervision, guidance, suggestion and encouragement during this semester.

**ABSTRACT**

In recent years, the demand for high-speed and efficient image processing has increased significantly, driven by various applications such as medical images, surveillance, and robotics. To meet this demand processors designed specifically for image processing have been developed which can provide superior performance compared to general-purpose processors.

This document presents the design and implementation of a custom processor optimized for image processing The processor architecture is based on a pipeline approach where different stages of image processing are performed in parallel The processor is designed to handle 256X256 grayscale padded images upon which several different filters will be applied. The processor is implemented using Verilog language and is simulated using Nexus 4 DDR Arctic 7 FPGA.

Along with Verilog other programming languages like Python is also used to change the format of image from png/jpg to .mem format so that the image data can be fed and synthesized by the FPGA. the final convolved image which is 256X256 in size is displayed on the VGA using the VGM monitor.

# Table of Contents

1. Introduction
   1. Problem Statement
   2. Overview of Processor Design
   3. Methodology
2. The Algorithm
   1. Addressing the Problem
   2. Convolution
3. Instruction Set Architecture
   1. General Architecture of Processor
   2. Data Path
   3. Instruction Set
   4. Control Signals by Control Unit
   5. Control Signal by ALU Control Unit
   6. Assembly Code of the Image Convolution Processor
   7. Instruction Cycle
4. RTL Modules
5. Testing, Simulation, and Modifications
   1. Modifications to rectify errors
   2. VGA simulation output
6. Codes
   1. Python code
   2. Verilog code

# Introduction

### Problem Statement

The main objective of this project is to make a customized processor which is capable of taking 256X256 grayscale padded images, performing convolution operations on the image, and displaying the output image on a VGA monitor. Since it is a customized processor it will take console less hardware and will also be faster than the general-purpose processor.

### Overview of Processor Design

The primary objective of this project is to design a Microprocessor and a CPU (Central Processing Unit) which can perform image filtering on the given image and generate correct results. The coding was done using Verilog Hardware Description Language (HDL). Xilinx Vivado was used as an analysis and synthesis tool for the design of the processor and for the simulation of Verilog HDL designs. The implemented design was then tested on NEXUS 4 DDR artic 7 FPGA. The report includes the Microprocessor and CPU design, the codes that are used, the vivado simulation output and the VGA display output.

The basic arithmetic, logical, input/output (I/O) operations speciﬁed by the instructions need to be executed to carry out the instructions of a computer program. The electronic circuitry within a computer that performs the above-mentioned task is called the Central Processing Unit (CPU). The main two building blocks of a CPU are the Processor and the Control Unit (CU).

### Methodology

The following steps were followed when designing the processor.

1. Develop the algorithm for image convolution.
2. Writing *Python* High-Level language code that included *OpenCV*

and *NumPy* libraries for converting the jpeg/png image to .mem format.

1. Deﬁne the speciﬁcation of the processor such as the number of registers, memory requirements and bus size according to the algorithm.
2. Design the ISA including instruction building blocks, microinstructions, datapath and control lookup table.
3. Convert the algorithm to instructions that would be stored in IRAM.
4. Implement the modules in Verilog.
5. Testing and verifying the results.

# The Algorithm

This section explains the working principles of the algorithm which was implemented inside the designed custom processor, to perform the image processing on the given 256x256 grayscale (single-channel) 8-bit image. Since the image is an 8-bit single channel, the maximum and minimum values a pixel can take become 255 and 0 respectively.

### Addressing the Problem

### We can divide the problem into 3 stages.

### First we will use python to pad the input jpeg/png image and make it 268X258 size. After this each pixel will be converted into 8bit format and the pixel values will range form 0 to 255. These new pixel values will be stored in .mem file format and included into Verilog design.

### In the second stage using the hardware generated, the image will be convolved using the desired filter and the convolved image will be stored into a block ram.

### The stored image in block ram will then be displayed on the vga monitor.

### Convolution:-

### convolution is a process of applying the filter to an image, in image processing. in this design we have taken a 3X3 filter which is applied on a 258X258 image. First, we fetch 9 different image pixel data and and store them in the registers in the processor. Then we multiply them with the filter corresponding values and sum them together to get the convolved pixel value. This processor is repeated over the entire image to get the convolved image. This process can be understood by the figure below.

### 

# Instruction Set Architecture (ISA)

# The Instruction Set Architecture deﬁnes the structure used withinside the layout in depth. This segment covers the structure of the Central Processing Unit (CPU), datapath and the connectivity among numerous modules.

### General Architecture of Processor:-

In our design processor module is the top module which contains instruction fetch, instruction decode, instruction execute, two memory modules:- one as source memory and the other as destination memory, write back circuitry, and VGA top module.

* Input-output ports:-

1. Clock(clk) is a system clock provided by nexys 4 ddr which works at 100MHz frequency.
2. Reset(rst) signal is used for putting the whole design at initial halt state. All the stored information in register will be cleared and no operations will be performed until rst is high.
3. Load(ld) is used to initialize the filter values with the desired ones.
4. Filter :- this control signal allows us to choose the filter values to be applied. When load signal will be high, the filter values can be changed by applying the filter ctrl signal. After this a rst toggle signal is necessary so that the changes can be reflected on output.
5. O\_complete:- it is output signal indicating that the application of filter on image has been complete and the convolved imge is stored in destination memory.
6. o\_hsync,o\_vsync,o\_red,o\_green,o\_blue :- when the complete signal is high , it represents that the processing is complete and the image is stored in destination memory . from here the vga top module starts functioning and display the convolved image on the vga monitor. While performing these actions , these signals are generated.

* Sub modules :-

1. IF module:-

Here the instructions pointed to by the program counter are fetched from the instruction ram, and also the next value of PC is computed.

Every instruction is a 32-bit instruction and has a unique address associated with it which is pointed by the program counter.

For every clock cycle the program counter is incremented by 4 and sometimes branch or jump instruction is supposed to come then a decision is made to either select the branch/jump address or the next incremented PC value.

1. ID module:-

The already fetched instruction is decoded here into opcode, rs rt, rd , 16 bit immediate value, 26-bit immediate value. According to the appropriate opcode the decoded instruction is used to generate the alu function command, bus A and B, sign extended immediate values, and the rd signal.

1. EX module:-

In this step, the ALU is used to perform some calculations. The exact operation depends on the instruction that is already decoded. The ALU operates on operands that have been already made ready in the previous cycle. A, B, Imm, etc.

1. Memory:-

The only instructions that make use of this step are loads, and stores.

1. WB:-

In this step, the result is written back into the register file. The result may come from the ALU or the result may come from the memory system. The position of the destination register in the instruction word depends on the instruction.

1. VGA:-

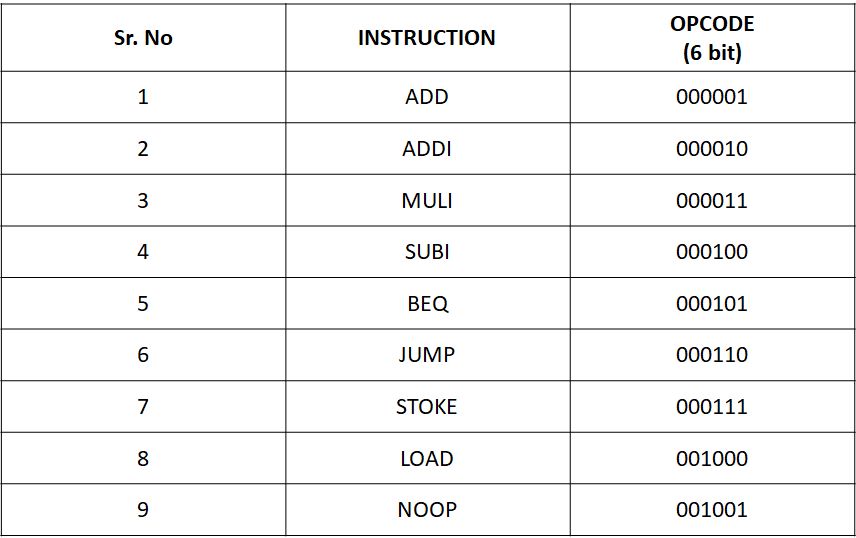
Upon completion of the filter application on the original image a complete signal is generated which will activate this module. It will be responsible to generate the appropriate signals of hsync and vsync and give appropriate RGB values.

### Data Path

### 

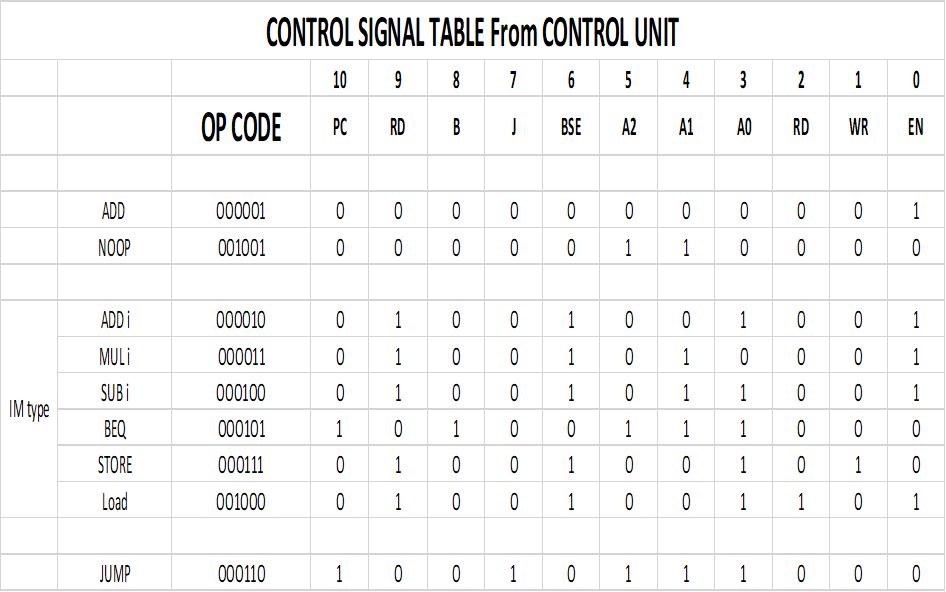
### Instruction Set

Instruction set contains the assembly code that we write for the processor. The following table represents all the instructions that we used to build the assembly code that we intend to use to perform the convolution.



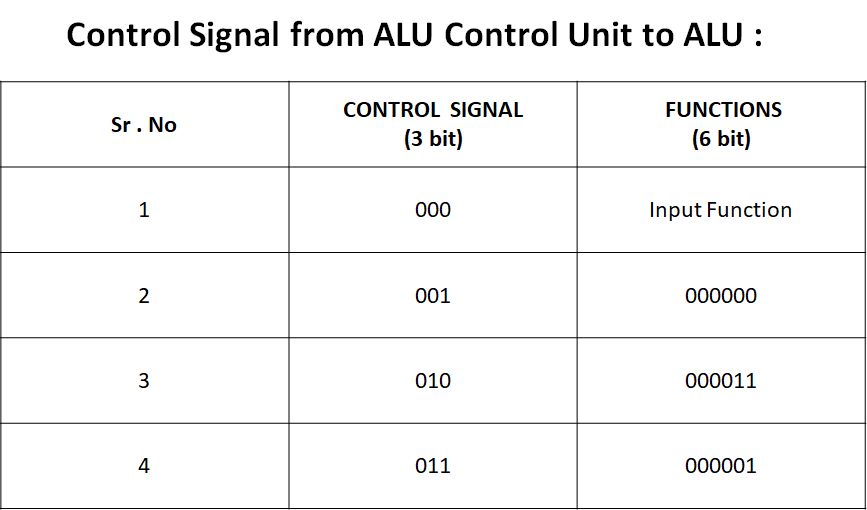
### Control signals by control unit

The control store contains all the control signals for each of the micro-instruction of each instruction in the ISA. The following table gives a great description of the LookUp Table that is implemented in the processor.

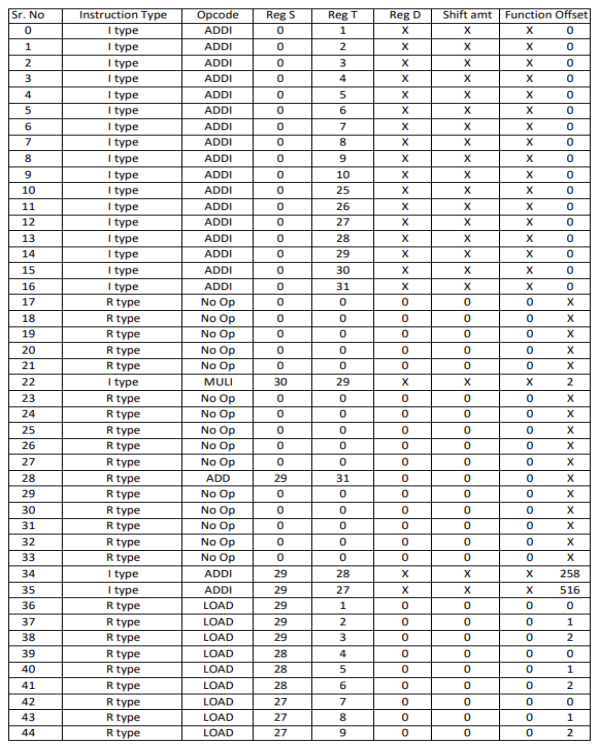


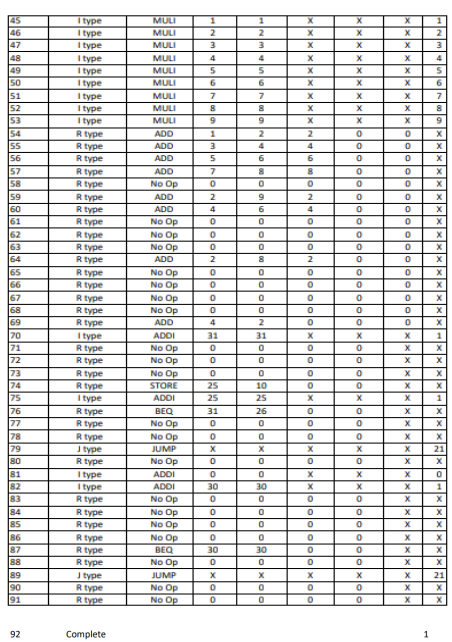
* 1. Control signal by alu control unit

Depending on the input control signals from the control unit the alu control unit generates control commands for the alu to function appropriately. These control signals are displayed below:-



* 1. Assembly Code of the image convolution process:-





### Instruction Cycle

### The designed processor works in 5 stages :-

### Instruction fetch stage:-

### The processor fetches the instruction from the memory address pointed to by the program counter the program counter is then incremented by 4 to point the next instruction.

### Instruction decode stage :-

### The processor decodes the instruction to determine what operation needs to be performed and which operands are involved the operands may be register memory location or immediate value.

### Execute stage :-

### The processor performs the arithmetic or logical operation specified by the instruction on the operands determined by the instruction decoder stage.

### Memory access stage :-

### The processor accesses memory to read or write data as required by the instruction. In the designed processor only 2 instructions which are load and store instructions can access the memory.

### Write-back stage:-

### The processor writes the result of the operation back to the appropriate register. The value might be fetched from the data memory or might be the output of the alu depending on the instruction.

# RTL Modules

* + 1. IF module:-

this module is responsible to generate instructions and choose the next PC. To perform these actions it is having submodules which are :-

1. p\_c
2. IRAM
3. Inc\_pc
4. If\_mux

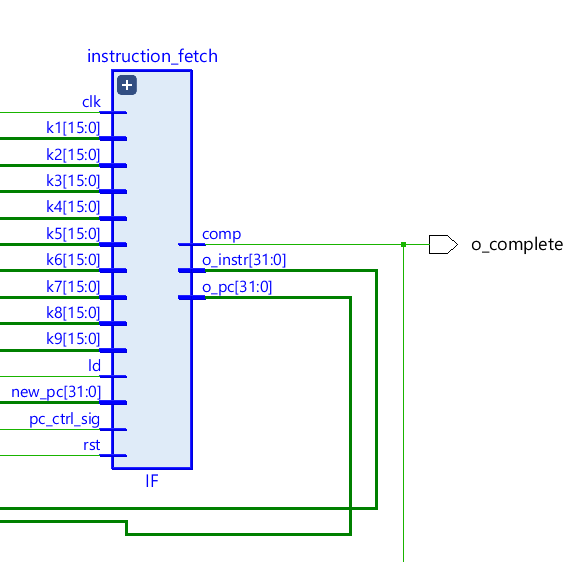
Until the rst is high the PC value is initialized to 0 else at every posedge of clk the PC value is taken as the output of the if\_mux module.

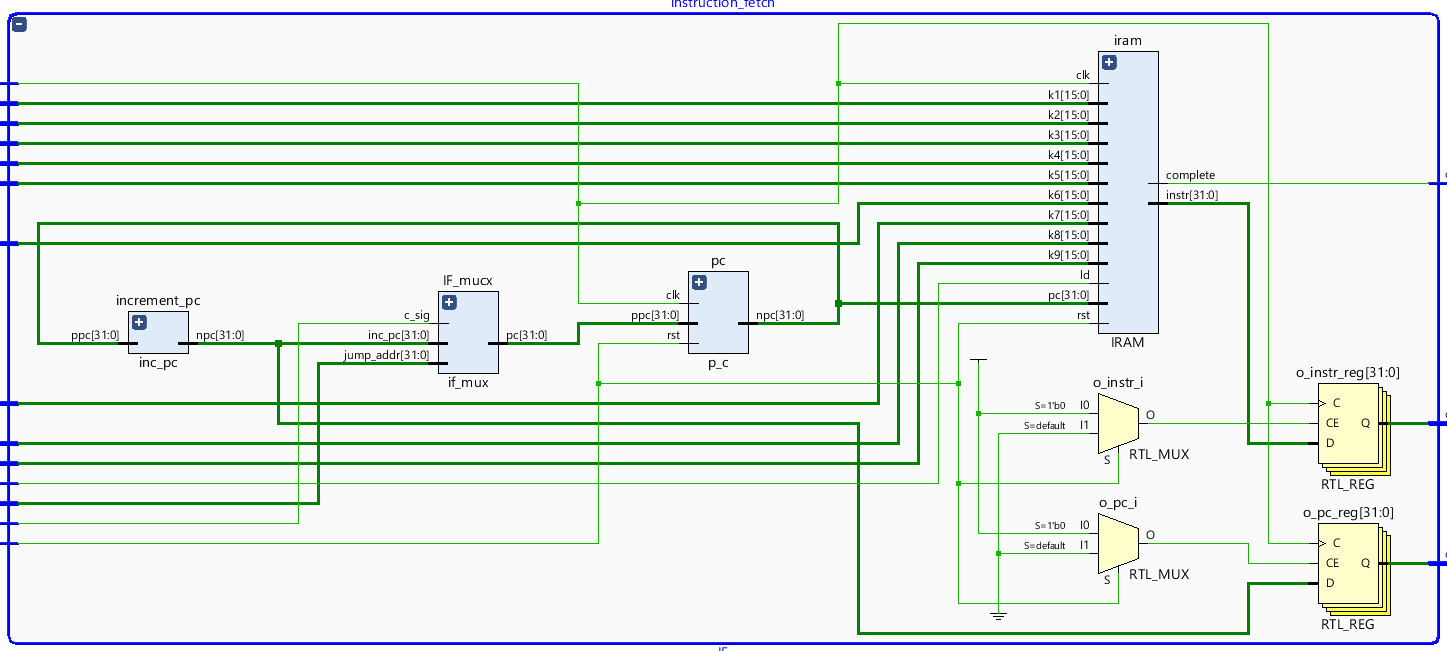
If\_mux module gets its input as the incremented PC or the branch/ jump address from the instruction decode stage. The 11th bit of the control signal generated by the control unit is the select line for the multiplexer inside the if\_mux module. If it is 0 then the PC+4 value is taken else the branch/jump address is taken.

Inc\_pc module increments the PC by 4 and passes the output to if\_mux module and the instruction decode stage.

The PC of the present clock is given input to the IRAM module which will provide a 32-bit instruction. IRAM module first divides the PC value by 4 and then by using a case block the respective instruction will be generated. These instructions are binary format of the assembly code which will be guiding the processor to take proper steps to perform the operation on the image.

* IF module schematic:-





* + 1. ID module:-

Previously generated PC values and instructions are decoded here. Slicing of the instruction is done in the below-described way:-

1. Instr[31:26] = opcode
2. Instr[25:21] = source register address
3. Instr[20:16] = source/destination register address
4. Instr[15:11] = destination register address
5. Instr[10:6] = shiftamount.
6. Instr[5:0] = function to be performed by alu
7. Instr[15:0] = 16 bit immediate value
8. Instr[25:0] = 26 bit immediate value

Submodules:-

1. Control unit
2. Regbank
3. Sign\_extend

The opcode is given input to the count unit which will generate the control signals for controlling the multiplexers and different modules in the design. The table below shows the control signals generated for the respective opcode.

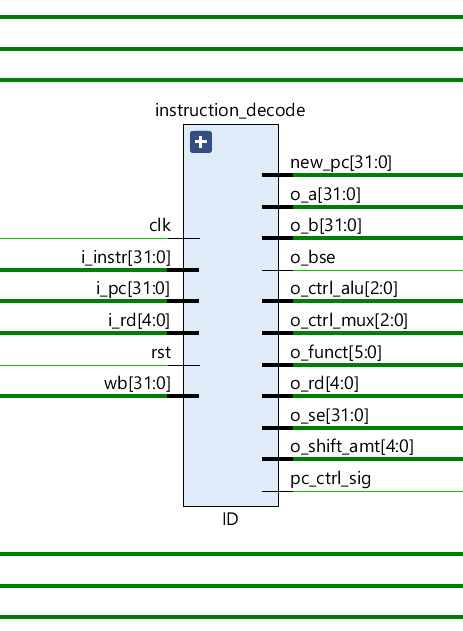
Rs and rt addresses are directly given as input to the register bank to fetch the values stored in the respective registers. These values are stored in buses A and B. For branch instruction, the busses values are compared and the compared signal along with the 6th bit of the control unit output is used to choose between the PC + 4 value of the PC + sign extended 16bit immediate value, and the result is stored in npc\_branch register. Meanwhile, the jump address is generated by taking 4 MSB bits of PC , 26-bit immediate value, and 2bit 0 as LSB.

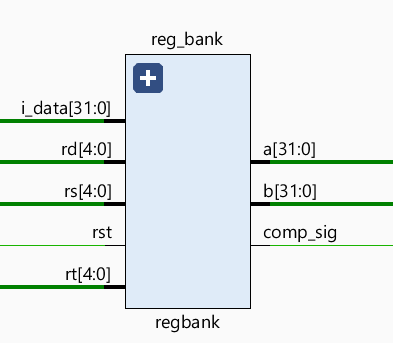
Using the 7th bit of control signal as the select line the npc\_branch or jump address is selected. This signal is then given to the if\_mux module as discussed in the IF module.

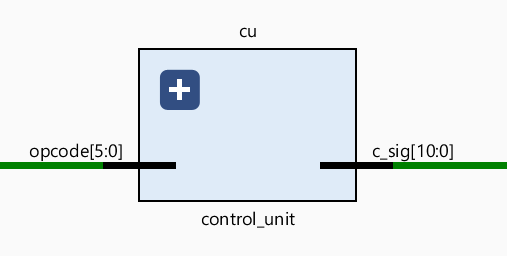
The 9th bit of the control signal is used to choose the select the address of rt or rd.

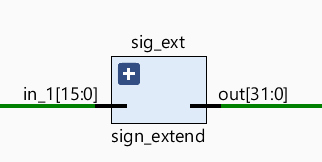
The ID module generates control signals, A, B, se and rd values as output for the next module.

Schematic of ID module and its sub module:-









* + 1. Execute module:-

The previously generated output is used by this module to do arithmetic and logic operations on the input data.

Submodules:-

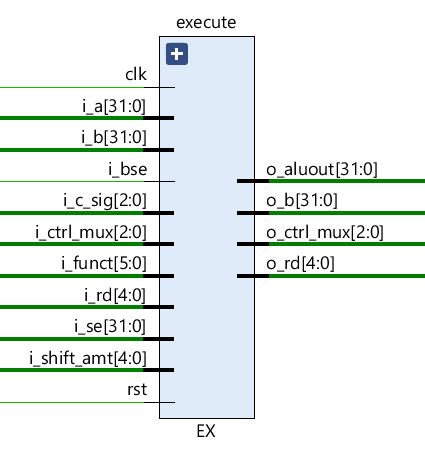
1. Ctrl\_alu
2. Alu

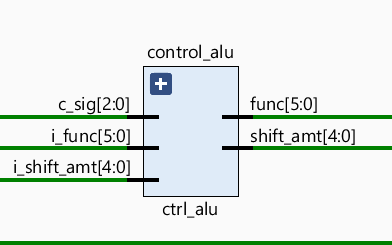
The ctrl\_alu module gets function and shift amount signals generated by the previous stage. Using these it generates appropriate commands for the ALU to perform the appropriate function on the data. The commands generated according to the input function and shift amount are given in the table below.

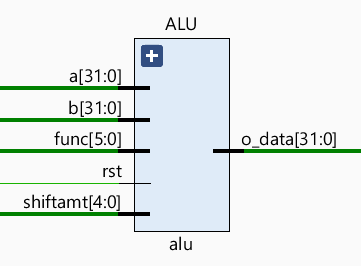
The ALU modules receive the busses A and B, and commands from the ctrl\_alu module. According to the commands, it performs arithmetic and logical actions on the data and generates alu\_out as the output signal.

This module forwards control signals, ALU output, bus B data, and destination register address.

Schematic of execute module and its sub module:-







* + 1. Memory stage:-

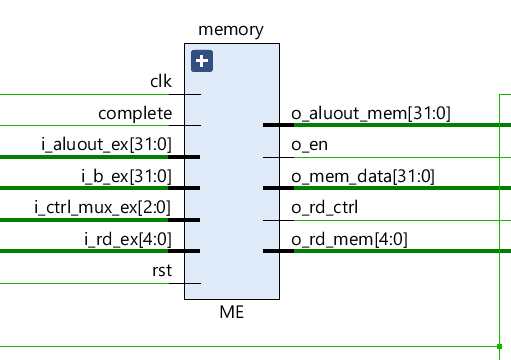
This module has two memory units which are block ram. the source memory size is 66564 for storing 258X258 images. The destination memory size is 65536 to store 256X256 images. The source memory is initialized using the command $readmemb(“dog.mem”,memory). This command is used to initialize the memory using a .mem file which has stored its data in binary format. For this, a .mem file was created using Python to pad the image with 2 rows and a column of 0 then convert each pixel to a 8-bit value.

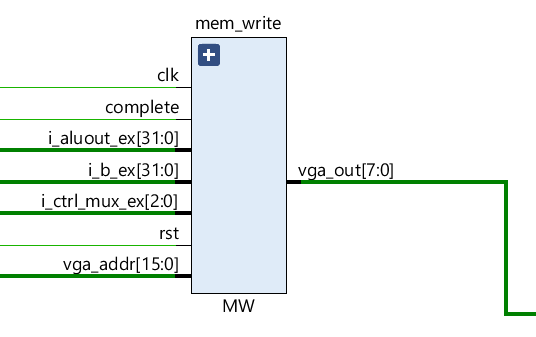
In the program, the pixel values were fetched form this module and after performing the convolution operation the convolved pixel is stored in the destination memory. When the complete signal is high, the vga module provides an input address to the destination memory and fetches the pixel to be displayed.

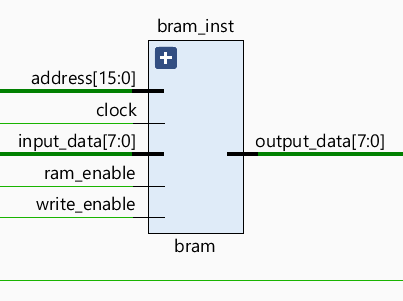
The control signals received from the previous stage are used for reading or writing into the memory. If 3rd bit is high then the data is read and if 2nd bit is high the data is written in the memory.

This module generates mem\_data\_out and forwards control signal, aluout and destination register address to the next stage.

Schematic of memory module , mem\_write and bram module:-

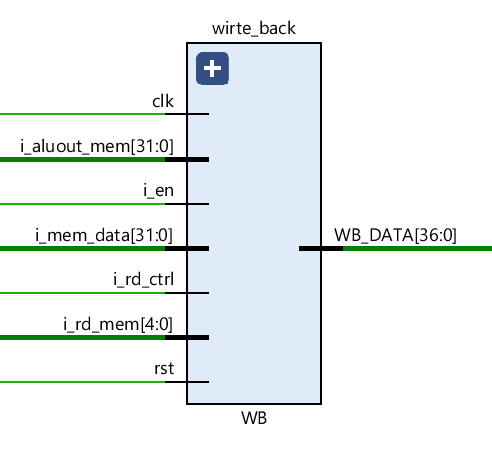






* + 1. Write back module:-

This module is used to select either data\_mem or aluout is to be passed or not. If the 2nd bit is high that means the present instruction is read instruction then the data\_mem is selected else aluout is selected. The data will be written back only if the 0th bit is high.

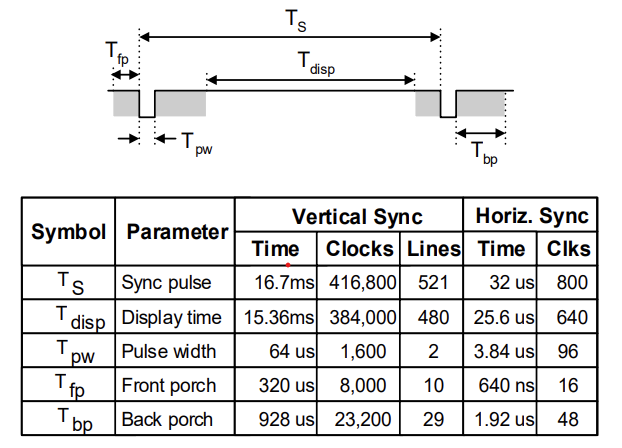


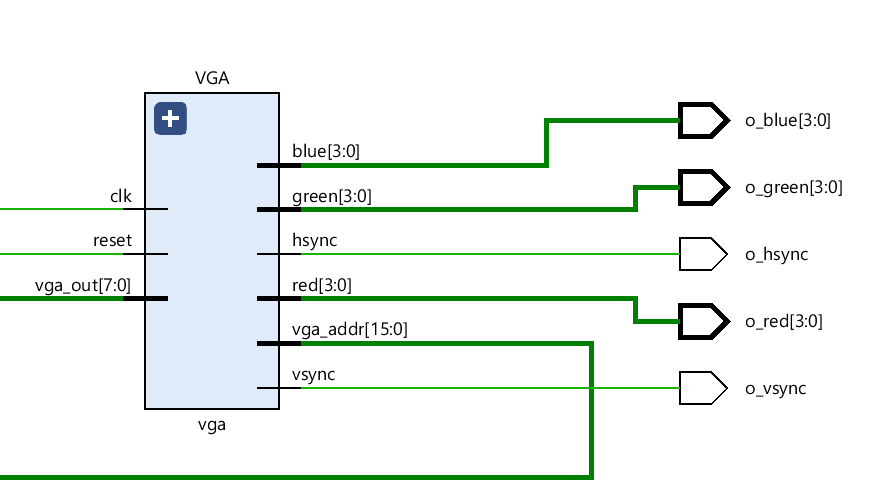
* + 1. VGA module :-

VGA module is used to display the convolved image which is stored in block ram. the VGA module works on 25 megahertz of the clock signal, VGA module provides a 16-bit address fetches the 8-bit data stored on the respective address of the block ram, and display it. VGA module is having red, blue, green, hsync and vsync as the output signals red blue, green signals are all 4 bit signals ,and hsync and vsync signals are oone-bit signal. to display agrayscalee image on the VGA monitor varied to give same pixel value to red, blue and green.

Horizontal Sync (hsync): This signal indicates the start of each line on the display. It is typically generated using a counter that increments on each clock cycle until it reaches a preset value. Once the counter reaches the preset value, it resets and generates a pulse to indicate the start of a new line. The hsync pulse width can be adjusted to set the horizontal resolution of the display.

Vertical Sync (vsync): This signal indicates the start of each frame on the display. It is typically generated using a counter that increments on each hsync pulse until it reaches a preset value. Once the counter reaches the preset value, it resets and generates a pulse to indicate the start of a new frame. The vsync pulse width can be adjusted to set the vertical resolution of the display.





# Testing, Simulation, and Modiﬁcations

### Modifications to rectify errors:-

### Module limit exceeds:-

### Vivado has a limit for trainable parameters in one module. In our processor initially, we store both the original image (258X258) and the convolved output image (256X256) in the same module which exceeded the limit. Due to this, we had to separate the memory module into write\_mem and memory module.

### Longer than expected synthesis time:-

### After splitting the memories, the design was taking several hours to synthesize, the reason might be because Vivado consumed RAM at a particular place, and in order to store the rest of the data it started making d flip-flops to store the data. To rectify this we used block ram.

### Multiple driver error:-

### In our first design, we kept the register bank as RAM. That design was working perfectly in simulation but was not getting implemented because in one clock cycle writing at one address and reading from another address. As both things were happening at the same time there was multiple driver error detected. In order to fix this multiple driver error we defined separate registers instead of RAM for the register bank and instantiated these resistors in the top module of the register bank.

### Poor placement of clock pin:-

### As we had given multiple reset signals in our design, the Vivado misunderstood the reset signal as a clock signal. In order to rectify this error we made the register zero default value 0 and used it to initialize other registers.

### PC getting randomly initialized on board:-

### While simulating the design on board, as the reset signal was getting low the PC was expected to start from 0 but instead of that it was getting randomly initialized. So in order to initialize the pc by 0 we used a counter in the IRAM and made it count until 4. As soon as the counter reached 4 we passed a jump instruction to make PC to 0.

* 1. VGA simulation output :

### 

### 

### 

### 

### 

### 

### FUTURE WORK :

### In this project we processed grayscale images, in the future, we would like to do work on colored images.

### The image which was used for input was stored in .mem file. in the future we would like to take image capture fromhe camera and display it on a monitor after performing filtration on it.

### References:-

### <https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual>

### <https://www.youtube.com/watch?v=UqdIBD4pJIU&t=1251s>

### <https://www.youtube.com/watch?v=nSgN93Zu-1g&list=PL1C2GgOjAF-JSlKHoSDOYVoF0lyCd76DD>

### Codes:-

### Python code:-

### Code to read an image file, pad it and convert into .mem forat :-

### import cv2 as cv

### import numpy as np

### import matplotlib.pyplot as plt

### # read and display the image

### file = 'peppers.png'

### image = cv.imread(file, cv.IMREAD\_GRAYSCALE)

### fig, ax = plt.subplots()

### ax.imshow(image, cmap='gray', vmin = 0, vmax = 255)

### plt.show()

### # write the binary representation of the image pixels to a txt file to replace data in Ram.v

### ram\_index = 0# we will be storing the image starting from the address 20 in our Data RAM

### with open(file.split('.')[0] + '.mem', 'w') as imgfile:

### for row in range(image.shape[0]+2):

### for col in range(image.shape[1]+2):

### # print(image[row, col])

### if row == 0:

### imgfile.write("{:08b}\n".format(0))

### ram\_index += 1

### elif row == 257:

### imgfile.write("{:08b}\n".format(0))

### ram\_index += 1

### elif col == 0:

### imgfile.write("{:08b}\n".format(0))

### ram\_index += 1

### elif col == 257:

### imgfile.write("{:08b}\n".format(0))

### ram\_index += 1

### else :

### imgfile.write("{:08b}\n".format(image[row-1, col-1]))

### ram\_index += 1

### Code to read the convolved image and plot it :-

### # read the text file output from the vivado and reconstruct the image

### import numpy as np

### import matplotlib.pyplot as plt

### dsimagesize = 256 # size of the downsampled image

### dsimage\_vivado = np.zeros((dsimagesize, dsimagesize))

### with open('output.txt', 'r') as imgfile:

### lines = imgfile.readlines()

### line = 0

### for row in range(dsimagesize):

### for col in range(dsimagesize):

### pixel\_val = int(lines[line].split('\n')[0][-8:], 2)

### # print(pixel\_val)

### dsimage\_vivado[row,col] = pixel\_val

### line += 1

### # visualise the image

### fig, ax = plt.subplots()

### ax.imshow(dsimage\_vivado, cmap='gray', vmin = 0, vmax = 255)

### plt.savefig("vivado.png")

### plt.show()

### Verilog code :-

### Processor top module :-

### `timescale 1ns / 1ps

### module processor(clk,rst,ld,filter,o\_hsync,o\_vsync,o\_red,o\_green,o\_blue,o\_complete);

### input clk,rst,ld;

### input [2:0] filter;

### output reg o\_hsync,o\_vsync;

### output reg [3:0] o\_red,o\_green,o\_blue;

### output o\_complete;

### reg [15:0] k1,k2,k3,k4,k5,k6,k7,k8,k9;

### wire pc\_ctrl\_sig;

### wire [31:0] new\_pc,o\_pc,o\_instr;

### wire [36:0] WB\_DATA;

### wire [2:0] o\_ctrl\_alu;

### wire o\_bse,o\_rd\_ctrl,o\_en;

### wire [5:0] o\_funct;

### wire [4:0] o\_shift\_amt;

### wire [31:0] o\_a,o\_b,o\_b\_ex,o\_se,o\_aluout\_ex,o\_mem\_data,o\_aluout\_mem;

### wire [4:0] o\_rd,o\_rd\_ex,o\_rd\_mem;

### wire [2:0] o\_ctrl\_mux,o\_ctrl\_mux\_ex;

### wire complete;

### assign o\_complete = complete;

### wire hsync,vsync;

### wire [3:0] red,green,blue;

### wire [15:0]addr;

### always @(\*) begin

### if(ld == 1) begin

### if(filter == 1) begin

### k1 = 16'b1000000000000010;

### k2 = 16'b1000000000000001;

### k3 = 16'b0000000000000000;

### k4 = 16'b1000000000000001;

### k5 = 16'b0000000000000001;

### k6 = 16'b0000000000000001;

### k7 = 16'b0000000000000000;

### k8 = 16'b0000000000000001;

### k9 = 16'b0000000000000010;

### end

### end

### end

### always @(\*) begin

### o\_hsync = hsync;

### o\_vsync = vsync;

### o\_red = red;

### o\_green = green;

### o\_blue = blue;

### end

### wire [15:0] vga\_addr;

### wire [7:0] vga\_out;

### assign vga\_12 = vga\_out;

### IF instruction\_fetch(clk,rst,ld,k1,k2,k3,k4,k5,k6,k7,k8,k9,new\_pc,pc\_ctrl\_sig,o\_pc,o\_instr,complete);

### ID instruction\_decode(clk,rst,WB\_DATA[36:32],WB\_DATA[31:0],o\_pc,o\_instr,o\_ctrl\_alu,o\_bse,o\_funct,o\_shift\_amt,o\_a,o\_b,o\_se,o\_rd,o\_ctrl\_mux,new\_pc,pc\_ctrl\_sig);

### // clk,rst,i\_c\_sig,i\_ctrl\_alu,i\_bse,i\_funct,i\_shift\_amt,i\_a,i\_b,i\_se,i\_rd,i\_ctrl\_mux,o\_ctrl\_mux,o\_aluout,o\_b, o\_rd

### EX execute(clk,rst,o\_ctrl\_alu,o\_bse,o\_funct,o\_shift\_amt,o\_a,o\_b,o\_se,o\_rd,o\_ctrl\_mux,o\_ctrl\_mux\_ex,o\_aluout\_ex,o\_b\_ex,o\_rd\_ex);

### ME memory (clk,rst,o\_ctrl\_mux\_ex,o\_aluout\_ex,o\_b\_ex,o\_rd\_ex,complete,o\_rd\_ctrl,o\_en,o\_mem\_data,o\_aluout\_mem,o\_rd\_mem);

### MW mem\_write(clk,rst,o\_ctrl\_mux\_ex,o\_aluout\_ex,o\_b\_ex,complete,vga\_addr,vga\_out);

### vga VGA(clk,~complete,red,green,blue,hsync,vsync,vga\_addr,vga\_out);

### WB wirte\_back(clk,rst,o\_rd\_ctrl,o\_en,o\_mem\_data,o\_aluout\_mem,o\_rd\_mem,WB\_DATA);

### endmodule

### IF module :-

### `timescale 1ns / 1ps

### module IF(clk,rst,ld,k1,k2,k3,k4,k5,k6,k7,k8,k9,new\_pc,pc\_ctrl\_sig,o\_pc,o\_instr,comp);

### input clk,rst,ld;

### input [15:0] k1,k2,k3,k4,k5,k6,k7,k8,k9;

### input [31:0] new\_pc;

### input pc\_ctrl\_sig;

### output reg [31:0] o\_pc,o\_instr;

### output comp;

### wire [31:0] ppc,npc,incre\_pc;

### wire [31:0] instr;

### wire complete;

### assign comp = complete;

### p\_c pc(clk,rst,ppc,npc);

### IRAM iram(clk,rst,ld,k1,k2,k3,k4,k5,k6,k7,k8,k9,npc,instr,complete);

### inc\_pc increment\_pc(npc,incre\_pc);

### if\_mux IF\_mucx(pc\_ctrl\_sig,incre\_pc,new\_pc,ppc);

### always @(posedge clk) begin

### if(rst == 0) begin

### o\_pc <= incre\_pc;

### o\_instr <= instr;

### end

### end

### endmodule

### p\_c module :-

### `timescale 1ns / 1ps

### // 1. programme counter module to pass new programme counter to the IRAM

### module p\_c(clk,rst,ppc,npc);

### input clk,rst;

### input [31:0] ppc;

### output reg [31:0] npc;

### always @(posedge clk) begin

### if(rst)

### npc = 0;

### else

### npc = ppc;

### end

### endmodule

### IRAM code :-

### `timescale 1ns / 1ps

### // 2. contains the instructions , updates the value when pc value changes

### module IRAM(clk,rst,ld,k1,k2,k3,k4,k5,k6,k7,k8,k9,pc,instr,complete);

### input clk,rst,ld;

### input [15:0] k1,k2,k3,k4,k5,k6,k7,k8,k9;

### input [31:0] pc;

### output reg [31:0] instr;

### output reg complete;

### reg [15:0] mul\_1,mul\_2,mul\_3,mul\_4,mul\_5,mul\_6,mul\_7,mul\_8,mul\_9;

### reg [6:0]counter;

### reg flag;

### localparam AAD = 6'b000001,

### AADI = 6'b000010,

### MULI = 6'b000011,

### SUBI = 6'b000100,

### BEQ = 6'b000101,

### JUMP = 6'b000110,

### STORE = 6'b000111,

### LOAD = 6'b001000,

### NOOP = 6'b001001;

### localparam REG\_0 = 5'b00000,

### REG\_1 = 5'b00001,

### REG\_2 = 5'b00010,

### REG\_3 = 5'b00011,

### REG\_4 = 5'b00100,

### REG\_5 = 5'b00101,

### REG\_6 = 5'b00110,

### REG\_7 = 5'b00111,

### REG\_8 = 5'b01000,

### REG\_9 = 5'b01001,

### REG\_10 = 5'b01010,

### REG\_11 = 5'b01011,

### REG\_12 = 5'b01100,

### REG\_13 = 5'b01101,

### REG\_14 = 5'b01110,

### REG\_15 = 5'b01111,

### REG\_16 = 5'b10000,

### REG\_17 = 5'b10001,

### REG\_18 = 5'b10010,

### REG\_19 = 5'b10011,

### REG\_20 = 5'b10100,

### REG\_21 = 5'b10101,

### REG\_22 = 5'b10110,

### REG\_23 = 5'b10111,

### REG\_24 = 5'b11000,

### REG\_25 = 5'b11001,

### REG\_26 = 5'b11010,

### REG\_27 = 5'b11011,

### REG\_28 = 5'b11100,

### REG\_29 = 5'b11101,

### REG\_30 = 5'b11110,

### REG\_31 = 5'b11111;

### always @(posedge clk) begin

### if(rst) begin

### counter = 0;

### end

### else begin

### counter = counter + 1;

### end

### end

### always @(\*) begin

### if(ld == 1) begin

### mul\_1 = k1;

### mul\_2 = k2;

### mul\_3 = k3;

### mul\_4 = k4;

### mul\_5 = k5;

### mul\_6 = k6;

### mul\_7 = k7;

### mul\_8 = k8;

### mul\_9 = k9;

### complete = 0;

### flag = 0;

### end

### else begin

### if(counter == 4 && flag == 0) begin

### instr = {JUMP , 26'b00\_0000\_0000\_0000\_0000\_0000\_0000}; // make pc 0

### #10 flag = 1;

### end

### else begin

### case (pc/4)

### //0 : inst=32'b000000\_00001\_00010\_00011\_00000\_000000;

### 0 : instr = {AADI,REG\_0,REG\_1,16'b0000\_0000\_0000\_0000}; //reg\_1 <- reg\_0 + 0;

### 1 : instr = {AADI,REG\_0,REG\_2,16'b0000\_0000\_0000\_0000}; //reg\_2 <- reg\_0 + 0;

### 2 : instr = {AADI,REG\_0,REG\_3,16'b0000\_0000\_0000\_0000}; //reg\_3 <- reg\_0 + 0;

### 3 : instr = {AADI,REG\_0,REG\_4,16'b0000\_0000\_0000\_0000}; //reg\_4 <- reg\_0 + 0;

### 4 : instr = {AADI,REG\_0,REG\_5,16'b0000\_0000\_0000\_0000}; //reg\_5 <- reg\_0 + 0;

### 5 : instr = {AADI,REG\_0,REG\_6,16'b0000\_0000\_0000\_0000}; //reg\_6 <- reg\_0 + 0;

### 6 : instr = {AADI,REG\_0,REG\_7,16'b0000\_0000\_0000\_0000}; //reg\_7 <- reg\_0 + 0;

### 7 : instr = {AADI,REG\_0,REG\_8,16'b0000\_0000\_0000\_0000}; //reg\_8 <- reg\_0 + 0;

### 8 : instr = {AADI,REG\_0,REG\_9,16'b0000\_0000\_0000\_0000}; //reg\_9 <- reg\_0 + 0;

### 9 : instr = {AADI,REG\_0,REG\_10,16'b0000\_0000\_0000\_0000}; //reg\_10 <- reg\_0 + 0;

### 10 : instr = {AADI,REG\_0,REG\_25,16'b0000\_0000\_0000\_0000}; //reg\_25 <- reg\_0 + 0;

### 11 : instr = {AADI,REG\_0,REG\_26,16'b0000\_0001\_0000\_0000}; //reg\_26 <- reg\_0 + 256;

### 12 : instr = {AADI,REG\_0,REG\_27,16'b0000\_0000\_0000\_0000}; //reg\_27 <- reg\_0 + 0;

### 13 : instr = {AADI,REG\_0,REG\_28,16'b0000\_0000\_0000\_0000}; //reg\_28 <- reg\_0 + 0;

### 14 : instr = {AADI,REG\_0,REG\_29,16'b0000\_0000\_0000\_0000}; //reg\_29 <- reg\_0 + 0;

### 15 : instr = {AADI,REG\_0,REG\_30,16'b0000\_0000\_0000\_0000}; //reg\_30 <- reg\_0 + 0;

### 16 : instr = {AADI,REG\_0,REG\_31,16'b0000\_0000\_0000\_0000}; //reg\_31 <- reg\_0 + 0;

### 17 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // main:noop

### 18 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // main:noop

### 19 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // main:noop

### 20 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // main:noop

### 21 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // main:noop

### 22 : instr = {MULI,REG\_30,REG\_29,16'b0000\_0001\_0000\_0010}; // reg\_29 <- reg\_30\*258;

### 23 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 24 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 25 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 26 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 27 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 28 : instr = {AAD ,REG\_29,REG\_31,REG\_29,5'b00000,6'b000000}; //reg\_29 <- reg\_29 + reg\_31;

### 29 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 30 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 31 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 32 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 33 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 34 : instr = {AADI,REG\_29,REG\_28,16'b0000\_0001\_0000\_0010}; //reg\_28 <- reg\_29 + 258;

### 35 : instr = {AADI,REG\_29,REG\_27,16'b0000\_0010\_0000\_0100}; //reg\_27 <- reg\_29 + 516;

### 36 : instr = {LOAD,REG\_29,REG\_1,16'b0000\_0000\_0000\_0000}; //reg\_1 <- mem[reg\_29 + 0];

### 37 : instr = {LOAD,REG\_29,REG\_2,16'b0000\_0000\_0000\_0001}; //reg\_2 <- mem[reg\_29 + 1];

### 38 : instr = {LOAD,REG\_29,REG\_3,16'b0000\_0000\_0000\_0010}; //reg\_3 <- mem[reg\_29 + 2];

### 39 : instr = {LOAD,REG\_28,REG\_4,16'b0000\_0000\_0000\_0000}; //reg\_4 <- mem[reg\_28 + 0];

### 40 : instr = {LOAD,REG\_28,REG\_5,16'b0000\_0000\_0000\_0001}; //reg\_5 <- mem[reg\_28 + 1];

### 41 : instr = {LOAD,REG\_28,REG\_6,16'b0000\_0000\_0000\_0010}; //reg\_6 <- mem[reg\_28 + 2];

### 42 : instr = {LOAD,REG\_27,REG\_7,16'b0000\_0000\_0000\_0000}; //reg\_7 <- mem[reg\_27 + 0];

### 43 : instr = {LOAD,REG\_27,REG\_8,16'b0000\_0000\_0000\_0001}; //reg\_8 <- mem[reg\_27 + 1];

### 44 : instr = {LOAD,REG\_27,REG\_9,16'b0000\_0000\_0000\_0010}; //reg\_9 <- mem[reg\_27 + 2];

### 45 : instr = {MULI,REG\_1 ,REG\_1,mul\_1}; //reg\_1 <- reg\_1 \* k1;

### 46 : instr = {MULI,REG\_2 ,REG\_2,mul\_2}; //reg\_2 <- reg\_2 \* k2;

### 47 : instr = {MULI,REG\_3 ,REG\_3,mul\_3}; //reg\_3 <- reg\_3 \* k3;

### 48 : instr = {MULI,REG\_4 ,REG\_4,mul\_4}; //reg\_4 <- reg\_4 \* k4;

### 49 : instr = {MULI,REG\_5 ,REG\_5,mul\_5}; //reg\_5 <- reg\_5 \* k5;

### 50 : instr = {MULI,REG\_6 ,REG\_6,mul\_6}; //reg\_6 <- reg\_6 \* k6;

### 51 : instr = {MULI,REG\_7 ,REG\_7,mul\_7}; //reg\_7 <- reg\_7 \* k7;

### 52 : instr = {MULI,REG\_8 ,REG\_8,mul\_8}; //reg\_8 <- reg\_8 \* k8;

### 53 : instr = {MULI,REG\_9 ,REG\_9,mul\_9}; //reg\_9 <- reg\_9 \* k9;

### 54 : instr = {AAD ,REG\_1 ,REG\_2 ,REG\_2 ,5'b00000,6'b000000}; //reg\_2 <- reg\_1 + reg\_2;

### 55 : instr = {AAD ,REG\_3 ,REG\_4 ,REG\_4 ,5'b00000,6'b000000}; //reg\_4 <- reg\_3 + reg\_4;

### 56 : instr = {AAD ,REG\_5 ,REG\_6 ,REG\_6 ,5'b00000,6'b000000}; //reg\_6 <- reg\_5 + reg\_6;

### 57 : instr = {AAD ,REG\_7 ,REG\_8 ,REG\_8 ,5'b00000,6'b000000}; //reg\_8 <- reg\_7 + reg\_8;

### 58 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 59 : instr = {AAD ,REG\_2 ,REG\_9 ,REG\_2 ,5'b00000,6'b000000}; //reg\_2 <- reg\_2 + reg\_9;

### 60 : instr = {AAD ,REG\_4 ,REG\_6 ,REG\_4 ,5'b00000,6'b000000}; //reg\_4 <- reg\_4 + reg\_6;

### 61 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 62 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 63 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 64 : instr = {AAD ,REG\_2 ,REG\_8 ,REG\_2 ,5'b00000,6'b000000}; //reg\_2 <- reg\_2 + reg\_8;

### 65 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 66 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 67 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 68 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 69 : instr = {AAD ,REG\_4 ,REG\_2 ,REG\_10 ,5'b00000,6'b000000}; //reg\_2 <- reg\_2 + reg\_4;

### 70 : instr = {AADI ,REG\_31 ,REG\_31,16'b0000\_0000\_0000\_0001}; //reg\_31 <- reg\_31 + 1;

### 71 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 72 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 73 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 74 : instr = {STORE,REG\_25,REG\_10 ,16'b0000\_0000\_0000\_0000}; // mem[reg\_25] <- reg\_2;

### 75 : instr = {AADI ,REG\_25 ,REG\_25,16'b0000\_0000\_0000\_0001}; //reg\_25 <- reg\_25 + 1;

### 76 : instr = {BEQ ,REG\_31 ,REG\_26,16'b0000\_0000\_0000\_0011}; //reg\_31 == reg\_26;

### 77 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 78 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 79 : instr = {JUMP , 26'b00\_0000\_0000\_0000\_0000\_0001\_0101}; // jump to main

### 80 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 81 : instr = {AADI ,REG\_0 ,REG\_31 ,16'b0000\_0000\_0000\_0000}; //reg\_31 <- reg\_0 + 0;

### 82 : instr = {AADI ,REG\_30 ,REG\_30,16'b0000\_0000\_0000\_0001}; //reg\_30 <- reg\_30 + 1;

### 83 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 84 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 85 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 86 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 87 : instr = {BEQ ,REG\_30 ,REG\_26,16'b0000\_0000\_0000\_0011}; // reg\_30 == reg\_26;

### 88 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 89 : instr = {JUMP , 26'b00\_0000\_0000\_0000\_0000\_0001\_0101}; // jump to main

### 90 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 91 : instr = {NOOP,REG\_0 ,REG\_0 ,16'b0000\_0000\_0000\_0000}; // noop

### 92 : complete = 1;

### endcase

### end

### end

### end

### endmodule

### inc\_pc :-

### `timescale 1ns / 1ps

### //3. increment programme counter by 4

### module inc\_pc(ppc,npc);

### input [31:0] ppc;

### output [31:0] npc;

### assign npc = ppc + 4;

### endmodule

### if\_mux :-

### `timescale 1ns / 1ps

### module if\_mux(c\_sig,inc\_pc,jump\_addr,pc);

### input c\_sig;

### input [31:0] inc\_pc;

### input [31:0] jump\_addr;

### output reg [31:0] pc;

### always @(\*) begin

### case(c\_sig)

### 0 : pc = inc\_pc;

### 1 : pc = jump\_addr;

### default : pc = inc\_pc;

### endcase

### end

### endmodule

### instruction decode :-

### `timescale 1ns / 1ps

### module ID(clk,rst,i\_rd,wb,i\_pc,i\_instr,o\_ctrl\_alu,o\_bse,o\_funct,o\_shift\_amt,o\_a,o\_b,o\_se,o\_rd,o\_ctrl\_mux,new\_pc,pc\_ctrl\_sig);

### input clk,rst;

### input [4:0] i\_rd;

### input [31:0] wb;

### input [31:0] i\_pc,i\_instr;

### output reg [2:0] o\_ctrl\_alu;

### output reg o\_bse;

### output reg [5:0] o\_funct;

### output reg [4:0] o\_shift\_amt;

### output reg [31:0] o\_a,o\_b,o\_se;

### output reg [4:0] o\_rd;

### output reg [2:0] o\_ctrl\_mux;

### output [31:0] new\_pc;

### output pc\_ctrl\_sig;

### wire [10:0] c\_sig;

### wire [31:0] a,b,se,npc\_se,npc\_branch;

### wire [4:0] rd;

### // {pc,rd,b,j,bse,a2\_a1\_a0,rd,wr,en}

### control\_unit cu(i\_instr[31:26],c\_sig);

### regbank reg\_bank(rst,i\_instr[25:21],i\_instr[20:16],i\_rd,wb,a,b,comp\_sig);

### sign\_extend sig\_ext(i\_instr[15:0],se);

### assign rd = c\_sig[9] ? i\_instr[20:16] : i\_instr[15:11];

### branch\_npc\_add bna({i\_instr[15],{14{1'b0}},i\_instr[14:0],2'b00},i\_pc[31:0],npc\_se);

### assign npc\_branch = (c\_sig[8]&comp\_sig) ? npc\_se : i\_pc[31:0];

### assign new\_pc = c\_sig[7] ? {i\_pc[31:28],i\_instr[25:0],2'b00} : npc\_branch;

### assign pc\_ctrl\_sig = c\_sig[10];

### // a2\_a1\_a0, bse, funct,shiftamt,a,b,se,rd, mem->reg, reg->mem, en;

### always @(posedge clk) begin

### if(rst == 0) begin

### o\_ctrl\_alu = c\_sig[5:3];

### o\_bse = c\_sig[6];

### o\_funct = i\_instr[5:0];

### o\_shift\_amt = i\_instr[10:6];

### o\_a = a;

### o\_b = b;

### o\_se = se;

### o\_rd = rd;

### o\_ctrl\_mux = {c\_sig[2],c\_sig[1],c\_sig[0]};

### end

### end

### endmodule

### control unit :-

### `timescale 1ns / 1ps

### module control\_unit(opcode,c\_sig);

### input [5:0] opcode;

### output reg [10:0]c\_sig;

### // {pc,rd,b,j,bse,a2,a1,a0,rd,wr,en}

### always @(opcode) begin

### case(opcode)

### 1 : c\_sig = 11'b0\_0\_0\_0\_0\_000\_0\_0\_1; // r-type

### 2 : c\_sig = 11'b0\_1\_0\_0\_1\_001\_0\_0\_1; // addi

### 3 : c\_sig = 11'b0\_1\_0\_0\_1\_010\_0\_0\_1; // muli

### 4 : c\_sig = 11'b0\_1\_0\_0\_1\_011\_0\_0\_1; // subi

### 5 : c\_sig = 11'b1\_0\_1\_0\_0\_111\_0\_0\_0; // beq

### 6 : c\_sig = 11'b1\_0\_0\_1\_0\_111\_0\_0\_0; // jump

### 7 : c\_sig = 11'b0\_1\_0\_0\_1\_001\_0\_1\_0; // store (reg -> mem)

### 8 : c\_sig = 11'b0\_1\_0\_0\_1\_001\_1\_0\_1; // load (mem -> reg)

### 9 : c\_sig = 11'b0\_0\_0\_0\_0\_111\_0\_0\_0; // NOOP

### endcase

### end

### endmodule

### regbank :-

### `timescale 1ns / 1ps

### // 6. this module contains register bank.

### module regbank(rst,rs,rt,rd,i\_data,a,b,comp\_sig);

### input rst;

### input [4:0] rs,rt,rd;

### input [31:0] i\_data;

### output reg [31:0] a,b;

### output reg comp\_sig;

### wire [31:0] a\_00,b\_00;

### wire [31:0] a\_01,b\_01;

### wire [31:0] a\_02,b\_02;

### wire [31:0] a\_03,b\_03;

### wire [31:0] a\_04,b\_04;

### wire [31:0] a\_05,b\_05;

### wire [31:0] a\_06,b\_06;

### wire [31:0] a\_07,b\_07;

### wire [31:0] a\_08,b\_08;

### wire [31:0] a\_09,b\_09;

### wire [31:0] a\_10,b\_10;

### wire [31:0] a\_25,b\_25;

### wire [31:0] a\_26,b\_26;

### wire [31:0] a\_27,b\_27;

### wire [31:0] a\_28,b\_28;

### wire [31:0] a\_29,b\_29;

### wire [31:0] a\_30,b\_30;

### wire [31:0] a\_31,b\_31;

### reg [31:0] i\_data\_00;

### reg [31:0] i\_data\_01;

### reg [31:0] i\_data\_02;

### reg [31:0] i\_data\_03;

### reg [31:0] i\_data\_04;

### reg [31:0] i\_data\_05;

### reg [31:0] i\_data\_06;

### reg [31:0] i\_data\_07;

### reg [31:0] i\_data\_08;

### reg [31:0] i\_data\_09;

### reg [31:0] i\_data\_10;

### reg [31:0] i\_data\_25;

### reg [31:0] i\_data\_26;

### reg [31:0] i\_data\_27;

### reg [31:0] i\_data\_28;

### reg [31:0] i\_data\_29;

### reg [31:0] i\_data\_30;

### reg [31:0] i\_data\_31;

### register\_00 R00(rst,rs,rt,rd,i\_data\_00,a\_00,b\_00);

### register\_01 R01(rst,rs,rt,rd,i\_data\_01,a\_01,b\_01);

### register\_02 R02(rst,rs,rt,rd,i\_data\_02,a\_02,b\_02);

### register\_03 R03(rst,rs,rt,rd,i\_data\_03,a\_03,b\_03);

### register\_04 R04(rst,rs,rt,rd,i\_data\_04,a\_04,b\_04);

### register\_05 R05(rst,rs,rt,rd,i\_data\_05,a\_05,b\_05);

### register\_06 R06(rst,rs,rt,rd,i\_data\_06,a\_06,b\_06);

### register\_07 R07(rst,rs,rt,rd,i\_data\_07,a\_07,b\_07);

### register\_08 R08(rst,rs,rt,rd,i\_data\_08,a\_08,b\_08);

### register\_09 R09(rst,rs,rt,rd,i\_data\_09,a\_09,b\_09);

### register\_10 R10(rst,rs,rt,rd,i\_data\_10,a\_10,b\_10);

### register\_25 R25(rst,rs,rt,rd,i\_data\_25,a\_25,b\_25);

### register\_26 R26(rst,rs,rt,rd,i\_data\_26,a\_26,b\_26);

### register\_27 R27(rst,rs,rt,rd,i\_data\_27,a\_27,b\_27);

### register\_28 R28(rst,rs,rt,rd,i\_data\_28,a\_28,b\_28);

### register\_29 R29(rst,rs,rt,rd,i\_data\_29,a\_29,b\_29);

### register\_30 R30(rst,rs,rt,rd,i\_data\_30,a\_30,b\_30);

### register\_31 R31(rst,rs,rt,rd,i\_data\_31,a\_31,b\_31);

### always @(\*) begin

### if(a == b)

### comp\_sig = 1;

### else

### comp\_sig = 0;

### end

### always @(\*) begin

### case(rs)

### 0 : a = a\_00;

### 1 : a = a\_01;

### 2 : a = a\_02;

### 3 : a = a\_03;

### 4 : a = a\_04;

### 5 : a = a\_05;

### 6 : a = a\_06;

### 7 : a = a\_07;

### 8 : a = a\_08;

### 9 : a = a\_09;

### 10 : a = a\_10;

### 25 : a = a\_25;

### 26 : a = a\_26;

### 27 : a = a\_27;

### 28 : a = a\_28;

### 29 : a = a\_29;

### 30 : a = a\_30;

### 31 : a = a\_31;

### endcase

### 

### case(rt)

### 0 : b = b\_00;

### 1 : b = b\_01;

### 2 : b = b\_02;

### 3 : b = b\_03;

### 4 : b = b\_04;

### 5 : b = b\_05;

### 6 : b = b\_06;

### 7 : b = b\_07;

### 8 : b = b\_08;

### 9 : b = b\_09;

### 10 : b = b\_10;

### 25 : b = b\_25;

### 26 : b = b\_26;

### 27 : b = b\_27;

### 28 : b = b\_28;

### 29 : b = b\_29;

### 30 : b = b\_30;

### 31 : b = b\_31;

### endcase

### end

### always @(rd or i\_data) begin

### case(rd)

### 0 : i\_data\_00 = i\_data;

### 1 : i\_data\_01 = i\_data;

### 2 : i\_data\_02 = i\_data;

### 3 : i\_data\_03 = i\_data;

### 4 : i\_data\_04 = i\_data;

### 5 : i\_data\_05 = i\_data;

### 6 : i\_data\_06 = i\_data;

### 7 : i\_data\_07 = i\_data;

### 8 : i\_data\_08 = i\_data;

### 9 : i\_data\_09 = i\_data;

### 10 : i\_data\_10 = i\_data;

### 25 : i\_data\_25 = i\_data;

### 26 : i\_data\_26 = i\_data;

### 27 : i\_data\_27 = i\_data;

### 28 : i\_data\_28 = i\_data;

### 29 : i\_data\_29 = i\_data;

### 30 : i\_data\_30 = i\_data;

### 31 : i\_data\_31 = i\_data;

### endcase

### end

### endmodule

### register :-

### `timescale 1ns / 1ps

### module register\_00(rst,rs,rt,rd,i\_data,out\_data\_a,out\_data\_b);

### input rst;

### input [4:0]rs,rt,rd;

### input [31:0] i\_data;

### output reg [31:0] out\_data\_a;

### output reg [31:0] out\_data\_b;

### reg [4:0] reg\_num;

### reg [31:0] reg\_data;

### always @(\*) begin

### if(rst) begin

### reg\_num = 0;

### end

### else begin

### if(reg\_num == rs)

### out\_data\_a <= 0;

### if(reg\_num == rt)

### out\_data\_b <= 0;

### if(reg\_num == rd)

### reg\_data <= i\_data;

### end

### 

### end

### endmodule

### sign\_ext:-

### `timescale 1ns / 1ps

### // 7. this module is used for extending the sign.

### module sign\_extend(in\_1,out);

### input [15:0] in\_1;

### output reg [31:0] out;

### // in[31:0] = in[15] 16'b0 in[14:0];

### wire s\_bit = in\_1[15];

### always @(\*) begin

### out = {s\_bit,{16{1'b0}},in\_1[14:0]};

### end

### endmodule

### bna :-

### `timescale 1ns / 1ps

### module branch\_npc\_add(in\_1,in\_2,out);

### input [31:0] in\_1,in\_2;

### output reg [31:0] out;

### always @(\*) begin

### if(in\_1[31] == in\_2[31]) begin

### out = in\_1[30:0] + in\_2[30:0];

### out[31] = in\_1[31];

### end

### else begin

### if(in\_1[30:0] > in\_2[30:0]) begin

### out[30:0] = in\_1[30:0] - in\_2[30:0];

### out[31] = in\_1[31];

### end

### else begin

### out[30:0] = in\_2[30:0] - in\_1[30:0];

### out[31] = in\_2[31];

### end

### end

### end

### endmodule

### execute :-

### `timescale 1ns / 1ps

### module EX(clk,rst,i\_c\_sig,i\_bse,i\_funct,i\_shift\_amt,i\_a,i\_b,i\_se,i\_rd,i\_ctrl\_mux,o\_ctrl\_mux,o\_aluout,o\_b,o\_rd);

### input clk,rst;

### input [2:0] i\_c\_sig;

### input i\_bse;

### input [5:0] i\_funct;

### input [4:0] i\_shift\_amt;

### input [31:0] i\_a,i\_b,i\_se;

### input [4:0] i\_rd;

### input [2:0] i\_ctrl\_mux;

### output reg [2:0] o\_ctrl\_mux;

### output reg [31:0] o\_aluout,o\_b;

### output reg [4:0] o\_rd;

### // a2\_a1\_a0, bse, funct,shiftamt,a,b,se,rd,mem->reg, reg->mem, en;

### // {pc,rd,b,j,bse,a2,a1,a0,rd,wr,en}

### wire [5:0] func;

### wire [4:0] shift\_amt;

### wire [31:0] b\_se,aluout;

### ctrl\_alu control\_alu(i\_c\_sig,i\_funct,i\_shift\_amt,func,shift\_amt);

### alu ALU(rst,func,shift\_amt,i\_a,b\_se,aluout);

### assign b\_se = i\_bse ? i\_se : i\_b;

### always @(posedge clk) begin

### if(rst == 0) begin

### o\_ctrl\_mux = i\_ctrl\_mux;

### o\_aluout = aluout;

### o\_b = i\_b;

### o\_rd = i\_rd;

### end

### end

### endmodule

### control unit :-

### `timescale 1ns / 1ps

### module ctrl\_alu(c\_sig,i\_func,i\_shift\_amt,func,shift\_amt);

### input [2:0]c\_sig;

### input [5:0] i\_func;

### input [4:0] i\_shift\_amt;

### output reg [5:0] func;

### output reg [4:0] shift\_amt;

### always @(\*) begin

### case(c\_sig)

### // r-type

### 0 : begin

### func = i\_func;

### shift\_amt = i\_shift\_amt;

### end

### // addi

### 1 : begin

### func = 6'b0;

### shift\_amt = i\_shift\_amt;

### end

### // muli

### 2 : begin

### func = 6'b000011;

### shift\_amt = i\_shift\_amt;

### end

### // subi

### 3 : begin

### func = 6'b000001;

### shift\_amt = i\_shift\_amt;

### end

### 

### endcase

### end

### endmodule

### alu :-

### `timescale 1ns / 1ps

### module alu(rst,func,shiftamt,a,b,o\_data);

### input rst;

### input [5:0] func;

### input [4:0] shiftamt;

### input [31:0] a,b;

### output reg [31:0] o\_data;

### always @(\*) begin

### if(rst) begin

### o\_data = 0;

### end

### else begin

### case(func)

### 0 : begin

### if(a[31] == b[31]) begin

### o\_data = a[30:0] + b[30:0];

### o\_data[31] = a[31];

### end

### else begin

### if(a[30:0] > b[30:0]) begin

### o\_data[30:0] = a[30:0] - b[30:0];

### o\_data[31] = a[31];

### end

### else begin

### o\_data[30:0] = b[30:0] - a[30:0];

### o\_data[31] = b[31];

### end

### end

### end

### 1 : o\_data = a - b;

### 2 : o\_data = a / b;

### 3 : begin

### o\_data[31] = a[31]^b[31];

### o\_data[30:0]= a[30:0] \* b[30:0];

### end

### 4 : o\_data = b >> shiftamt;

### 5 : o\_data = b << shiftamt;

### 6 : o\_data = a >> shiftamt;

### 7 : o\_data = a << shiftamt;

### endcase

### end

### end

### endmodule

### memory :-

### module ME(clk,rst,i\_ctrl\_mux\_ex,i\_aluout\_ex,i\_b\_ex,i\_rd\_ex,complete,o\_rd\_ctrl,o\_en,o\_mem\_data,o\_aluout\_mem,o\_rd\_mem);

### input clk,rst;

### input [2:0] i\_ctrl\_mux\_ex;

### input [31:0] i\_aluout\_ex,i\_b\_ex;

### input [4:0] i\_rd\_ex;

### input complete;

### output reg o\_rd\_ctrl,o\_en;

### output reg [31:0] o\_mem\_data,o\_aluout\_mem;

### output reg [4:0] o\_rd\_mem;

### reg [31:0]mem\_data;

### (\* RAM\_STYLE="BLOCK" \*)

### reg [7:0] memory [66563:0];

### initial

### $readmemb("peppers.mem", memory);

### always @(\*) begin

### if(i\_ctrl\_mux\_ex[2:1] == 2) begin

### mem\_data = {24'b0,memory[i\_aluout\_ex]};

### end

### 

### end

### always @(posedge clk) begin

### if(rst == 0) begin

### // mem->reg, en, data\_mem , aluout, rd;

### o\_rd\_ctrl = i\_ctrl\_mux\_ex[2];

### o\_en = i\_ctrl\_mux\_ex[0];

### o\_mem\_data = mem\_data;

### o\_aluout\_mem = i\_aluout\_ex;

### o\_rd\_mem = i\_rd\_ex;

### end

### end

### endmodule

### mem\_write :-

### `timescale 1ns / 1ps

### module MW(clk,rst,i\_ctrl\_mux\_ex,i\_aluout\_ex,i\_b\_ex,complete,vga\_addr,vga\_out);

### input clk,rst;

### input [2:0] i\_ctrl\_mux\_ex;

### input [31:0] i\_aluout\_ex,i\_b\_ex;

### input complete;

### input [15:0] vga\_addr;

### output reg [7:0] vga\_out;

### reg WE;

### reg write;

### reg [7:0]in\_data;

### reg [7:0] inp\_dat;

### wire [7:0]out\_data;

### reg [15:0] addr;

### always @(\*) begin

### if(complete == 1) begin

### write = 0;

### addr = vga\_addr;

### vga\_out = out\_data;

### end

### else begin

### if(i\_ctrl\_mux\_ex[2:1] == 1) begin;

### write = 1;

### addr = i\_aluout\_ex[15:0];

### inp\_dat = in\_data;

### end

### else begin

### write = 0;

### addr = i\_aluout\_ex[15:0];

### inp\_dat = in\_data;

### end

### end

### end

### bram

### #(

### .RAM\_WIDTH (8 ),

### .RAM\_ADDR\_BITS (16 ),

### .INIT\_START\_ADDR(0 ),

### .INIT\_END\_ADDR (65535 )

### )

### bram\_inst

### (

### .clock (clk ),

### .ram\_enable (1 ),

### .write\_enable (write ),

### .address (addr ),

### .input\_data (inp\_dat ),

### .output\_data (out\_data )

### );

### always @(\*) begin

### if(i\_ctrl\_mux\_ex[2:1] == 1) begin

### if(i\_b\_ex[31] == 1)

### in\_data = 8'b00000000;

### else if(i\_b\_ex >= 255)

### in\_data = 8'b11111111;

### else

### in\_data = i\_b\_ex[7:0];

### end

### end

### endmodule

### bram :-

### module bram

### #(

### parameter RAM\_WIDTH = 32,

### parameter RAM\_ADDR\_BITS = 9,

### parameter INIT\_START\_ADDR = 0,

### parameter INIT\_END\_ADDR = 10

### )

### (

### input clock,

### input ram\_enable,

### input write\_enable,

### input [15:0] address,

### input [RAM\_WIDTH-1:0] input\_data,

### output reg [RAM\_WIDTH-1:0] output\_data

### );

### 

### (\* RAM\_STYLE="BLOCK" \*)

### reg [RAM\_WIDTH-1:0] ram\_name [(2\*\*RAM\_ADDR\_BITS)-1:0];

### always @(posedge clock)

### if (ram\_enable) begin

### if (write\_enable)

### ram\_name[address] <= input\_data;

### output\_data <= ram\_name[address];

### end

### 

### reg [15:0] index;

### integer f;

### always@(posedge clock)

### begin

### if ( ram\_name[65535] == 0 )

### begin

### index <= index+1;

### $fwrite(f,"%b\n",ram\_name[index] ); //NOT DRAM < DMEM

### 

### end

### if(index == 65535)

### begin

### $fclose(f);

### $finish;

### end

### end

### initial begin

### f = $fopen("output.txt","wb");

### index = 0;

### end

### 

### endmodule

### 

### VGA :-

### `timescale 1ns / 1ps

### module vga(

### input clk,

### input reset,

### 

### output reg [3:0] red,

### output reg [3:0] green,

### output reg [3:0] blue,

### output reg hsync,

### output reg vsync,

### output [15:0]vga\_addr,

### input [7:0] vga\_out

### 

### );

### parameter hpixels = 12'd800;

### parameter vlines = 12'd521;

### parameter hbp = 12'd144;

### parameter hfp = 12'd784;

### parameter vbp = 12'd31;

### parameter vfp = 12'd511;

### parameter W = 256;

### parameter H = 256;

### wire [10:0] C1, R1;

### reg [11:0] hc\_new;

### reg [9:0] hc;

### reg [9:0] vc;

### reg vidon;

### reg spriteon;

### reg vsenable;

### always @ (posedge clk)

### if (reset || vsenable) hc\_new <= 0;

### else hc\_new <= hc\_new + 1;

### always@\* hc = hc\_new[11:2];

### 

### always@\* vsenable = hc == hpixels - 1;

### 

### always @\* hsync = hc > (127);

### always @ (posedge clk)

### if (reset || vc\_clr) vc <= 0;

### else if (vsenable) vc <= vc + 1;

### assign vc\_clr = vsenable & (vc == vlines - 1);

### always@\* vsync = vc > 2;

### always @(\*) vidon = ((hc < hfp) && (hc > hbp) && (vc < vfp) && (vc > vbp));

### assign C1 = 11'd100;

### assign R1 = 11'd100;

### always @(\*) spriteon = ((hc >= C1 + hbp) && (hc < C1 + hbp + W) && (vc >= R1 + vbp) && (vc < R1 + vbp + H));

### always @(\*)

### begin

### red = 15;

### green = 15;

### blue = 15;

### if ((spriteon == 1) && (vidon == 1))

### begin

### red = vga\_out[7:4];

### green = vga\_out[7:4];

### blue = vga\_out[7:4];

### end

### end

### reg [17:0] addr\_cnt;

### always@(posedge clk)

### if(reset)

### addr\_cnt <= 0;

### else if((spriteon == 1) && (vidon == 1))

### addr\_cnt <= addr\_cnt + 1;

### assign vga\_addr = addr\_cnt[17:2];

### endmodule

### write back :-

### `timescale 1ns / 1ps

### module WB(clk,rst,i\_rd\_ctrl,i\_en,i\_mem\_data,i\_aluout\_mem,i\_rd\_mem,WB\_DATA);

### input clk,rst;

### input i\_rd\_ctrl,i\_en;

### input [31:0] i\_mem\_data,i\_aluout\_mem;

### input [4:0] i\_rd\_mem;

### output reg [36:0] WB\_DATA;

### reg [31:0] data;

### always @(\*) begin

### data = i\_rd\_ctrl ? i\_mem\_data : i\_aluout\_mem;

### end

### always @(\*) begin

### if(rst == 0 & i\_en == 1) begin

### WB\_DATA = {i\_rd\_mem,data};

### end

### end

### endmodule