



FPGA BASED PROCESSOR IMPLEMENTATION

This report is submitted as a partial fulfilment for the module

EN3030: Circuits and Systems Design

Department of Electronic and Telecommunication Engineering

University of Moratuwa

28th of July 2019

A.A.J.H.M.T.C Aluvihare 160019X

J.S.M Jayawardhana 160247T

H.U.K.J Viraj 160655R

H.H.K.B.D. Hettiarachchi 160209F

CONTENTS

1. Introduction	3
1.1.1. Processor	3
1.1.2. Specific Task and Solution	3
2. Processor Design.....	4-11
2.1.1. Instructions Set Architecture	4
2.1.2. Instructions	4
2.1.3. Registers	4
2.1.4. Datapath	5
2.1.5. Micro Instructions.....	6-10
2.1.6. State Diagram	11
3. Modules	12
3.1.1. Registers	13-14
3.1.2. ALU.....	14-15
3.1.3. Clock Divider	15
3.1.4. DRAM.....	16
3.1.5. IROM.....	17
4. Algorithm	18-21
5. Results Analysis and Verification	22-23
6. References	24
7. Appendix.....	24-79

1. Introduction

1.1. Processor

A 'processor' process data and output information. A generic processor contains 3 parts; control unit, arithmetic and logic unit (ALU) and the memory unit. Data is fed into the processor from memory (storage) or input device, processed in the ALU and fed back to the memory or through output devices.

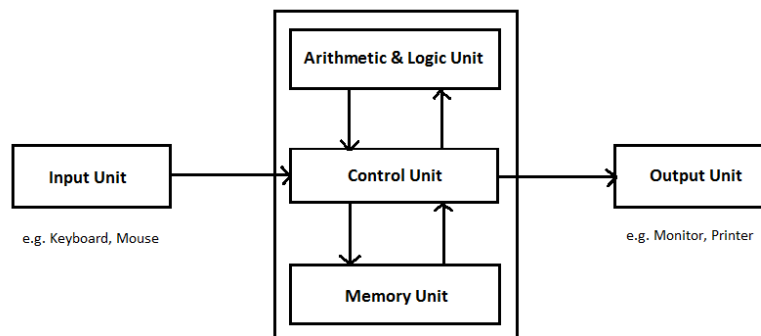


Figure 1.1 - Components of a Processor

A special purpose processor is a processor which is designed to do a specific task efficiently and cost effectively. These processors are optimized to do the relevant task and may or may not be able to be used in another task. Importance and the usefulness of the special purpose processors depend on their speed, power consumption, efficiency, economic and technical feasibility and the area encountered.

1.2. Specified Task and Solution

The task of the special purpose processor we designed is down-sampling an image by a factor of 2. A grayscale image of size 256X256 (65536 pixels) is down-sampled to get an image of size 128X128 (16384 pixels).

If we directly select and keep only the pixels with even index, the resulting image fulfils our requirement. This procedure can introduce very high frequency components to the image which distorts the data in the image in a harmful manner. Therefore, we should come up with an algorithm which can do this task while minimizing the distortion.

The algorithm is first developed and executed in Matlab and the appropriateness is observed. Then we designed the instruction set architecture of the processor such that the algorithm can be executed with a zero error. The processor was designed using Verilog Hardware Description Language using Quartus II Prime Lite Edition. Altera DE2-115 FPGA was used to deploy the design. The complete procedure includes 3 main parts.

1. Feeding the image into the processor
2. Processing the data
3. Getting the output from the processor

2. Processor Design

2.1 Instruction Set Architecture

2.1.1 Instructions

INSTRUCTION	OPCODE	OPERATION
NOP	00000001	No operation
CLAC	00000010	$AC \leftarrow 0, Z \leftarrow 1$
LDAC	00000011	$AC \leftarrow M[AR]$
STAC	00000100	$M[AR] \leftarrow AC$
MVACR	00000101	$R \leftarrow AC$
MVACR1	00000110	$R1 \leftarrow AC$
MVACR2	00000111	$R2 \leftarrow AC$
MVACR3	00001000	$R3 \leftarrow AC$
MVACR4	00001001	$R4 \leftarrow AC$
MVRAC	00001010	$AC \leftarrow R$
MVR1AC	00001011	$AC \leftarrow R1$
MVR2AC	00001100	$AC \leftarrow R2$
MVR3AC	00001101	$AC \leftarrow R3$
MVR4AC	00001110	$AC \leftarrow R4$
ADDN α	00001111 α	$AC \leftarrow AC + \alpha$, if $(AC + \alpha) = 0$, then $Z \leftarrow 1$, else $Z \leftarrow 0$
SUBN α	00010000 α	$AC \leftarrow AC - \alpha$, if $(AC - \alpha) = 0$, then $Z \leftarrow 1$, else $Z \leftarrow 0$
SFTL α	00010001 α	$AC \leftarrow AC \ll \alpha$
SFTR α	00010010 α	$AC \leftarrow AC \gg \alpha$
ADD	00010011	$AC \leftarrow AC + R$
SUB	00010100	$AC \leftarrow AC - R$
MVACAR	00010101	$AR \leftarrow AC$
END	00010110	End of the program
JMNZ β	00010111 β	If $(z=0)$, then go to β

2.1.2 Registers

REGISTER	SIZE(bit)	TASK
IR	8	Stores the instruction taken from IRAM
PC	16	Keeps the address of the next instruction to be executed
AR	16	Keeps the address of DRAM location of the next data to be fetched
AC	16	Memory access Direct access to ALU
R,R1,R2,R3,R4	16	General purpose registers

2.1.3 Datapath

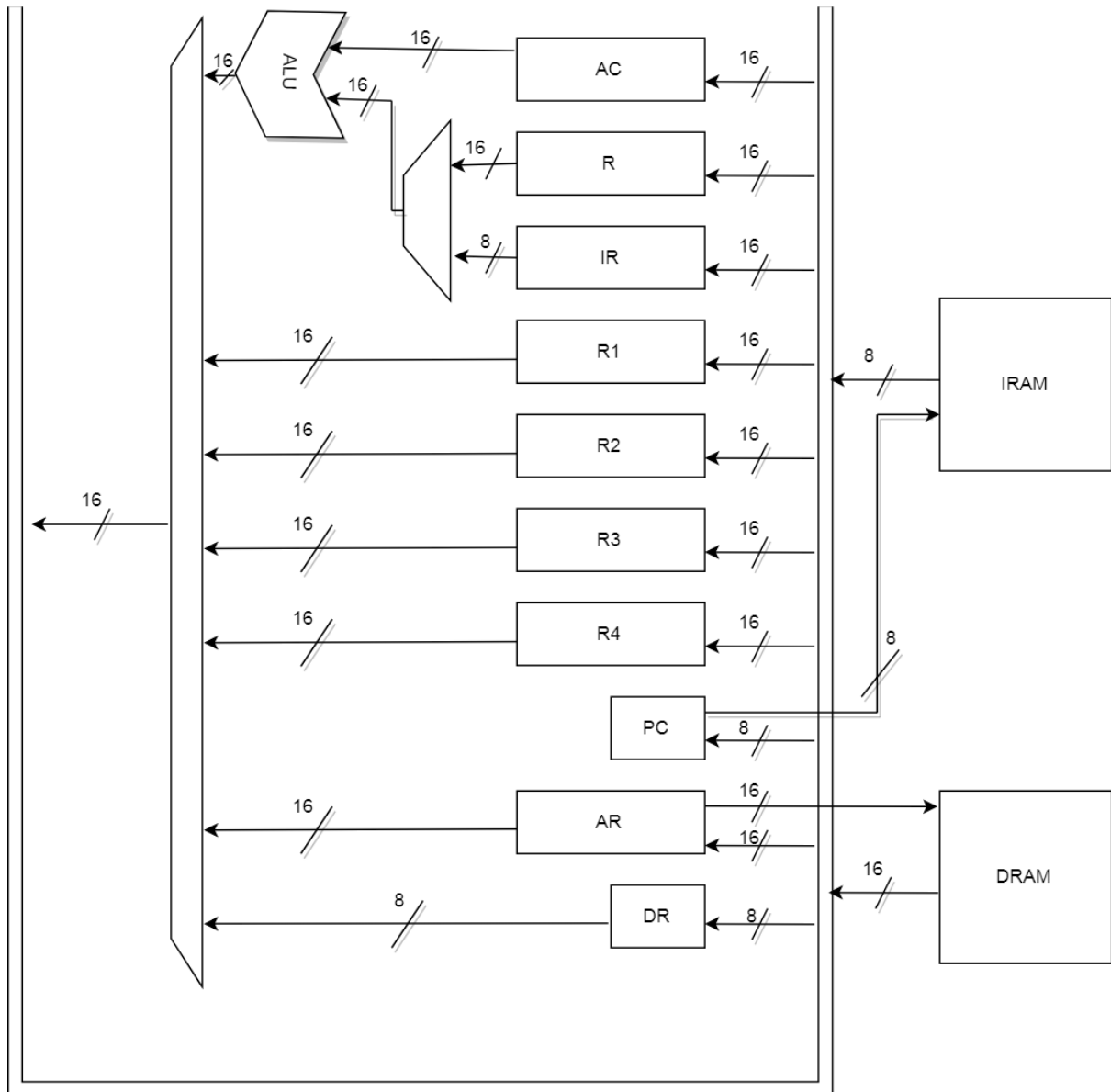


Figure 2.1 - Datapath of the processor

2.1.4 Micro Instructions

Micro instructions are the states of the processor. Instructions are fetched using 3 states as follows.

FETCH 1: IM READ

FETCH 2: $IR \leftarrow IM$

Then the instruction is decoded as follows.

FETCH3: if $(Z == 1 \ \& \ IR[5:0] == 6'd39)$ $PC \leftarrow PC + 1$

Else: $PC \leftarrow PC + 1, NS \leftarrow IR[5:0]$

After the decoding of the instruction, it is executed. The execution states depend on the instruction to be executed. Every micro instruction corresponding to each instruction is mentioned below.

Clearing the AC register to value 0.

CLAC : $AC \leftarrow 0$

Loading data from memory to the AC register. The AR register should contain the corresponding memory address for this operation.

LDAC 1: DM READ

LDAC 2: $DR \leftarrow DM$

LDAC 3:

LDAC 4: $AC \leftarrow DR$

Writing a memory location with the value of AC. . The AR register should contain the corresponding memory address for this operation.

STAC 1: $DR \leftarrow AC$

STAC 2: $DM \leftarrow DR$; DM WRITE

Loading the value of AC to R.

MVACR: $R \leftarrow AC$

Loading the value of AC to R1.

MVACR1: $R1 \leftarrow AC$

Loading the value of AC to R2.

MVACR2: $R2 \leftarrow AC$

Loading the value of AC to R3.

MVACR3: $R3 \leftarrow AC$

Loading the value of AC to R4.

MVACR4: $R4 \leftarrow AC$

Loading the value of R to AC.

MVRAC: $AC \leftarrow R$

Loading the value of R1 to AC.

MVRAC1: $AC \leftarrow R1$

Loading the value of R2 to AC.

MVRAC2: $AC \leftarrow R2$

Loading the value of R3 to AC.

MVRAC3: $AC \leftarrow R3$

Loading the value of R4 to AC.

MVRAC4: $AC \leftarrow R4$

Adding the next value in instruction memory to the AC register and loading the answer back to AC.

ADDN α

ADDN1: IM Read

ADDN2: $IR \leftarrow IM$

ADDN3: $ALUA \leftarrow AC, ALUA \leftarrow IR$

ADDN4: $AC \leftarrow AC + IR, PC \leftarrow PC + 1$

Subtracting the next value in instruction memory from the AC register and loading the answer back to AC.

SUBN α

SUBN1: IM Read

SUBN2: $IR \leftarrow IM$

SUBN3: $AC \leftarrow AC - IR$, $PC \leftarrow PC + 1$

Multiplying the value of the AC register by a factor of α by left shifting by α bits.

SFTL α

SFTL 1: IM Read

SFTL 2: $IR \leftarrow IM$

SFTL3: $AC \ll IR$, $PC \leftarrow PC + 1$

Dividing the value of the AC register by a factor of α by right shifting by α bits.

SFTR α

SFTR 1: IM Read

SFTR 2: $IR \leftarrow IM$

SFTR3: $AC \gg IR$, $PC \leftarrow PC + 1$

Adding the value of R to the AC register and loading the answer back to AC.

ADD

ADD1: $ALU \leftarrow AC + R$

ADD2: $AC \leftarrow ALU$

Subtracting the value of R from the AC register and loading the answer back to AC.

SUB

SUB1: $AC \leftarrow AC - R$

SUB2: $AC \leftarrow ALU$

Copying the value of AC to AR.

MVACAR

MVACAR1: $AR \leftarrow AC$

Branching of the instructions.

JMNZ β

If($Z=0$)

JMNZ1Y: IM Read

JMNZY2:IR←IM

JMNZY3:PC[7:0]←IR

If(Z=1)

JMNZN1:PC←PC+1

2.1.5 State Diagram

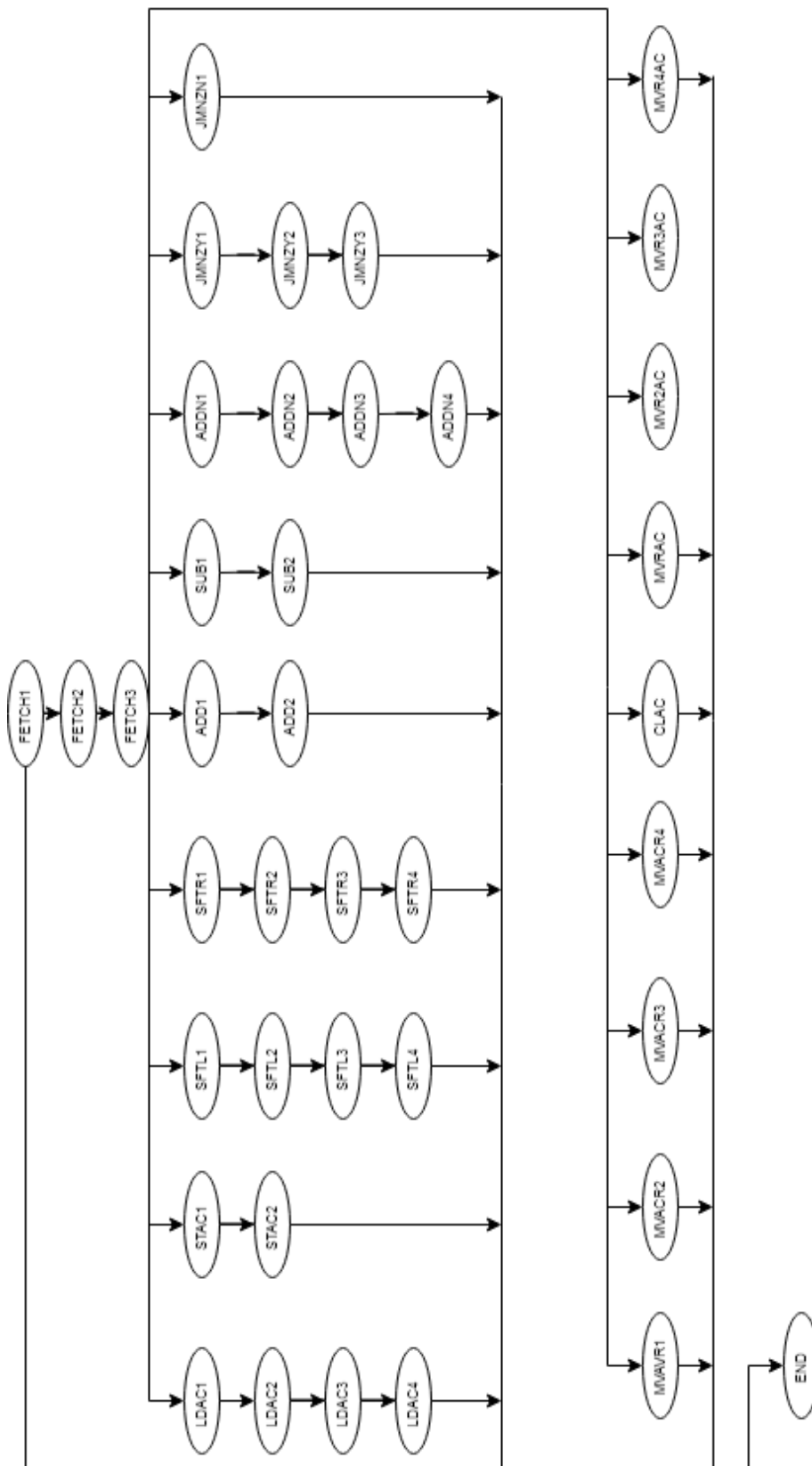


Figure 2.2 - State Diagram

3. Modules

In the processor module all the instances related to processing and controlling part are available. This contains a several modules. Memory modules, clock divider, etc. are not inside the processor module. Following modules are in the processor in our implementation.

1. Register-R
2. Register-R1
3. Register-R2
4. Register-R3
5. Register-R4
6. IR
7. PC
8. AC
9. AR
10. DR
11. ROM
12. MUX1
13. MUX2
14. DRAM
15. ALU
16. Cock Divider
17. State Machine

The processor module has four inputs as showing in the figure.

1. **Clock** (single bit) – The clock gives the timing using an oscillator inside the FPGA board for synchronization. For our processor implementation the clock speed of the built-in oscillator inside the FPGA is much higher. We use a clock divider in order to reduce the clock speed.
2. **Enable** (single bit) – Enable is also a single bit input which is used to enable the processing giving an input manually. With Enable it is easy to demonstrate because otherwise the starting point is not in our hand.

Our processor has six outputs as showing in the figure.

1. **AC** (16 bits) – AC in the processor is taken to the outside of the module to visualize for testing purposes as well as demonstration purposes.
2. **PC** (8 bits) – PC in the processor is also taken to the outside the module in order to fulfil above two requirements same as in AC.
3. **Finish** (single bit) – Finish bit is the indicator of the termination of processing done by the processor. When the processing is finished finish-bit is turned ON.

Inside the processor module there are several modules related to the processing and controlling. The processor module handles connectivity among those modules.

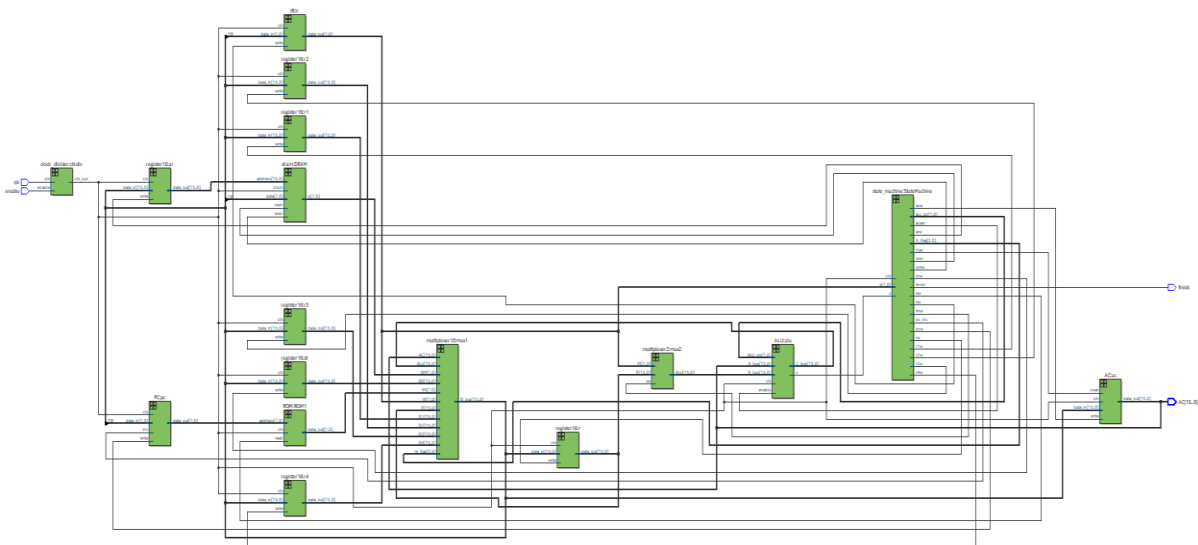


Figure 3.1 - RTL View

3.1 Registers

3.2.1. R, R1, R2, R3, R4

Registers are used to store temporary data during the process cycle. All are a 16-bits registers which are connected to the B-bus via a multiplexer. R becomes special with respect to R1, R2, R3 and R4 because R does not need to use B bus which is a long way to access the ALU. This register has no self-incremental operation, but it can easily access the ALU.

This module has single bit inputs Clock, Enable and 16-bits input Data-In as inputs. Enable is the bit which enables outputting data to the Data-out. As output of the modules, Data-out is the only one which is also 16-bits data.

3.2.3. PC

PC keeps the pointer to the instruction's address to be run next. It has four load-bit, increment-bit and clock with single bit and 16-bits Data-in from the instruction RAM to access the instructions. The address

of the instructions is 16 bits long, but they are not totally used because our ISA has a few instructions.

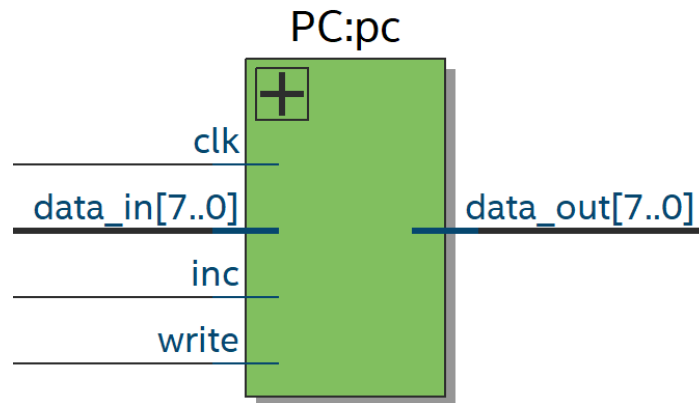


Figure 3.3 - PC

The output of the Data-out is 8 bit long because the instruction is 8 bits long. PC can self-increment without involving the ALU.

The PC is incremented at the positive edge of the clock.

3.2.4. IR

IR is an 8-bits register which keeps the Instruction during the process cycle. Just after every instruction runs the next instruction is fetched from IR. Basically IR works as a D flip-flop triggered at the positive edge of the clock. This has a data-in of 8-bits, clock and enable bits as well. The Data-input is connected to the A-Bus and the Data-out of 8-bits is also connected to the A-Bus through the MUX.

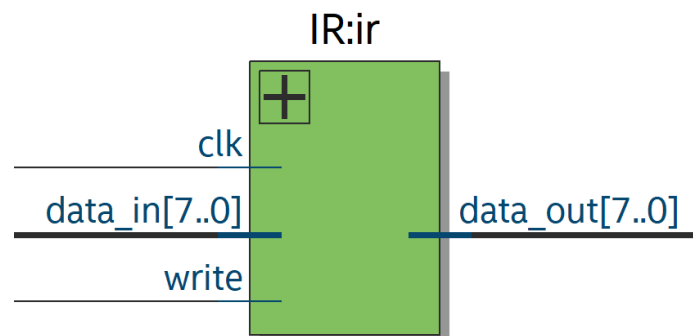


Figure 3.4 - IR

3.2.5. AC

AC is the only register directly connected to the ALU. This is 16-bits register which involve mostly when performing ALU operations. The input is connected to the A-Bus with 16-bits input. Single-bit load, increment-bit, clock and Data-in of 16-bits are the inputs for AC. The output is a Data-out of 16-bits

connecting to the ALU directly. When performing most of ALU operations the data is stored in AC. AC also can self-increment by 1 existing value in AC.

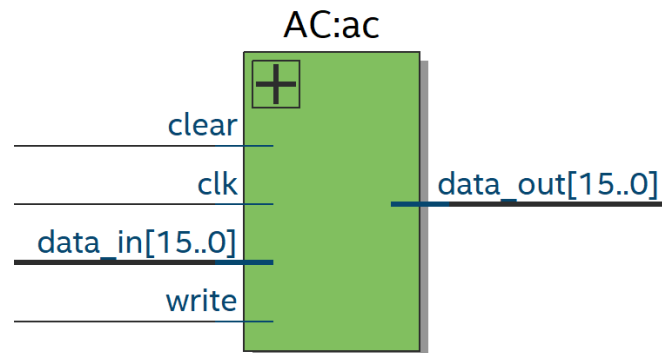


Figure 3.5 - AC

3.2 Arithmetic and Logic Unit (ALU)

All the arithmetic and logical operations are done through this module. It has 2 inputs which are A bus and B bus which are 16 bit each. A bus is the output of AC register. Which means AC is directly connected to the ALU. And there is only one output which is C bus. It is also a 16 bit bus. This output is directly connected to the input of the AC register. Therefore any output of an ALU operation can be written only to AC register.

In order to do an ALU operation following step can be taken,

1. Store one operand in register 'R'
2. Take the other operand into 'AC' (Accumulator)
3. Do the ALU operation

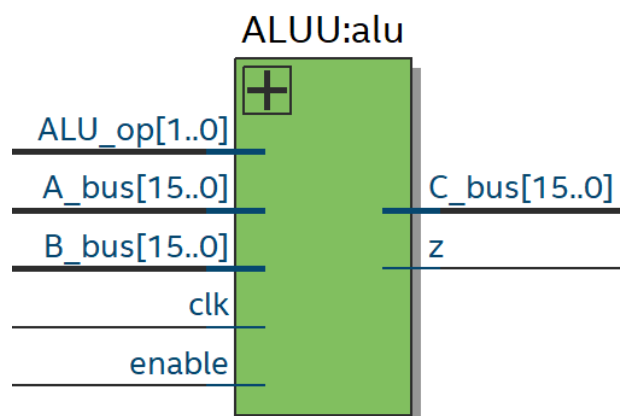


Figure 3.6 - ALU

Now the operand in the AC register will be the input ALU from the A bus and the value stored in R bus will be the input to the ALU from the B bus. Then the ALU operation will be done according to the ALU_op

flag in the module. It is a 2 bit flag and 4 operations can be done from this ALU module. These operations are decided by the control unit according to each instruction. Those operations are as follow,

Operation	ALU_op Flag	
ADD	00	$C \leftarrow A+B$
SUB	01	$C \leftarrow A-B$
SFTL	10	$C \leftarrow A \ll B$
SFTR	11	$C \leftarrow A \gg B$

In this module z flag is used to indicate whether the output of the ALU is zero or not. This was very useful when we implement JUMPNZ instruction in the control unit. Therefore before every JUMPNZ instruction we had to use SUB instruction which writes its output to the AC. We have set the z value high if the output value is zero($C=0$). Otherwise z is zero.

3.3 Clock Divider

This module is used to reduce the frequency of the original clock. We used an inbuilt 100MHz clock in the Spartan 6 FPGA development board. But in the processing cycle the processor has to read data from registers and calculate values through the ALU in between the positive and negative edges of the each clock cycle. So for 100MHz clock the time gap is 5ns which might not be sufficient to perform all these tasks. So, a clock divider module is implemented to widen this gap between positive and negative edges of the clock cycle. It takes the original clock as the input and returns a divided clock as output. All components of the processor use this divided clock as their input clock. So if we want to stop the processing we can do it by setting enable pin as 'Low'.

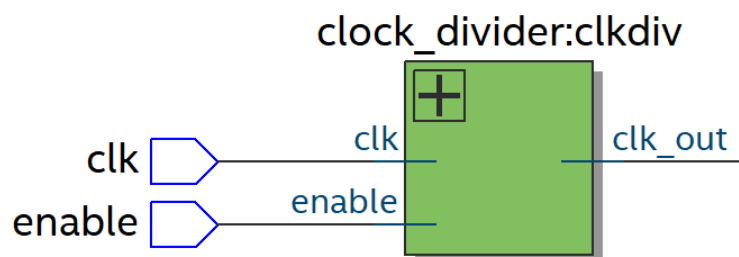


Figure 3.7 - Clock Divider

3.4 DRAM

This module is the main data memory which gives space to save image data for processing purposes. Our input image is a 256x256 image. We used matlab to generate a '.mif' file of the image which can be saved in the d-ram for further operations. This module has 65536 memory locations with a width of 8 bit to store the pixels of the image. Each pixel values which changes from 0 to 255 can be save in this 8 bit memory locations. This module has five input data paths and one output data path. "data" input is used to give input data and "address" path gives the address value of 16 bit which the data has to be read or write in the dram. "wren" and "rden" data inputs which have one bit, gives control signal for dram when control unit needs to write or read data. The "clock" gives the synchronizing signal to do operations in the dram. "q" used to give output data to the data bus when requested from the control unit.

In this module 65536 data units can be stored and hence we have used a 16 bit address to represent each data unit. Internal data storage of dram can be shown as following diagram.

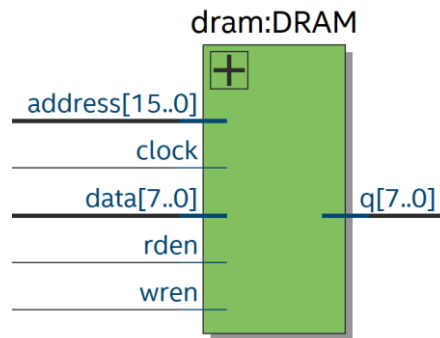


Figure 3.8 - DRAM

0x0000								
0x0001								
0xFFFF								

3.4 IROM

This module is used to store the instructions in the memory. All the instructions which are coded by assembly language is stored in this memory and when the processor needs instructions it fetches instructions to the IR. This module consists of 256 memory locations with a width of 8 bits. We have stored our assembly code for the algorithm of filtering and down sampling an image in this iram.

This module has three input data paths and one output path. The “address” data path gives a value of 8 bits which gives the address to read data from the module. ”read” input gives control signal for the iram when to read data from it and the clock is used for synchronizing operations. “data_out” is the data path which gives the output data to the data bus. We have used 126 instructions in our assembly code and they are stored in the memory locations of the iram.

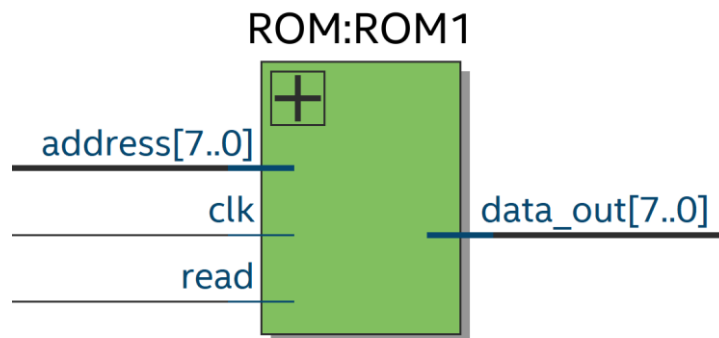


Figure 2.9 - IROM

4. Algorithm

Algorithm part is mainly consist of two main concepts which are filtering and down sampling. In our algorithm we have implement both these functions at the same time. So a Gaussian filter is used to filter the image pixel values and those filtered values are arranged according to a down sampling algorithm to obtain the filtered down sampled image. Algorithm was initially generated and tested using Matlab.

For the filtering part, 3x3 Gaussian kernel has been used and the weight distribution of the kernel is shown below.

1	2	1
2	4	2
1	2	1

After overlapping the center value of the kernel with a image pixel we calculate a weighted summation value and then by dividing that value by the weight of the kernel which is 16 we obtain a new filtered pixel value. In order to down sample the image in to half the size of the original image, we use the gaussian filter for one value per four pixel values from the image and stored that value in a new array to obtain the down sampled image.

Algorithm as a MATLAB code

```
DataMemory = zeros(1,65536);%linear data array for input image
outArray = zeros(127,127);%arra for downsampled image
for i = 1:256
    for j = 1:256
        DataMemory(256*(i-1)+j)= imGray(i,j);%generate the input data array
    end
end

for i = 2:2:254
    for j = 2:2:254
        PixelAddress = 256*(i-1)+j;%memory address of centre pixel of the
3*3 gaussian kernel
        NewPixel = 0;%correspodig pixel value of dowsapled image
        %Gaussia blurring and downsampling processes together
        NewPixel = NewPixel + DataMemory(PixelAddress)*4;
        NewPixel = NewPixel + DataMemory(PixelAddress-256)*2;
        NewPixel = NewPixel + DataMemory(PixelAddress-257)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress-255)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress+256)*2;
        NewPixel = NewPixel + DataMemory(PixelAddress+257)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress+255)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress+1)*2;
        NewPixel = NewPixel + DataMemory(PixelAddress-1)*2;
        NewPixel=NewPixel/16;
        outArray(i/2,j/2) = NewPixel;%saving pixel vales of dowsampled
image
    end
end
```

Assembly code written in to IRAM

Address	Instruction
0	CLAC
1	MVACR
2	MVACR1
3	MVACR2
4	MVACR3
5	MVACR4
6	MVR3AC
7	ADDN
8	2
9	MVACR3
10	CLAC
11	MVACR4
12	MVR4AC
13	ADDN
14	2
15	MVACR4
16	MVR3AC
17	SFTL
18	8
19	MVACR
20	MVR4AC
21	ADD
22	SUBN
23	255
24	SUBN
25	2
26	MVACR1
27	MVACAR
28	LDAC
29	SFTL
30	2
31	MVACR
32	MVR1AC
33	SUBN
34	255
35	SUBN
36	1
37	MVACAR
38	LDAC
39	SFTL
40	1

41	ADD
42	MVACR
43	MVR1AC
44	SUBN
45	255
46	SUBN
47	2
48	MVACAR
49	LDAC
50	ADD
51	MVACR
52	MVR1AC
53	SUBN
54	255
55	MVACAR
56	LDAC
57	ADD
58	MVACR
59	MVR1AC
60	ADDN
61	255
62	ADDN
63	1
64	MVACAR
65	LDAC
66	SFTL
67	1
68	ADD
69	MVACR
70	MVR1AC
71	ADDN
72	255
73	ADDN
74	2
75	MVACAR
76	LDAC
77	ADD
78	MVACR
79	MVR1AC
80	ADDN
81	255
82	MVACAR
83	LDAC

84	ADD
85	MVACR
86	MVR1AC
87	ADDN
88	1
89	MVACAR
90	LDAC
91	SFTL
92	1
93	ADD
94	MVACR
95	MVR1AC
96	SUBN
97	1
98	MVACAR
99	LDAC
100	SFTL
101	1
102	ADD
103	SFTR
104	4
105	MVACR
106	MVR2AC
107	MVACAR
108	MVRAC
109	STAC
110	MVR2AC
111	ADDN
112	1
113	MVACR2
114	MVR4AC
115	SUBN
116	254
117	JMNZ
118	12
119	MVR3AC
120	SUBN
121	254
122	JMNZ
123	6
124	END
125	NOP

5. Result analysing and verification

To analyze the result of FPGA, same image must be down sampled using MATLAB either. Refer the appendix for the relevant MATLAB code. By comparing two images, idea can be obtained about accuracy and efficiency of the processor. So analyzing error is very important to do some improvement processor. If output of the processor does not have much error, apparently both of down sampled images look same. Therefore, error calculation algorithm is needed to observe the error. Refer the appendix for the relevant MATLAB code.



Figure 5.1 - Original Image (256X256)

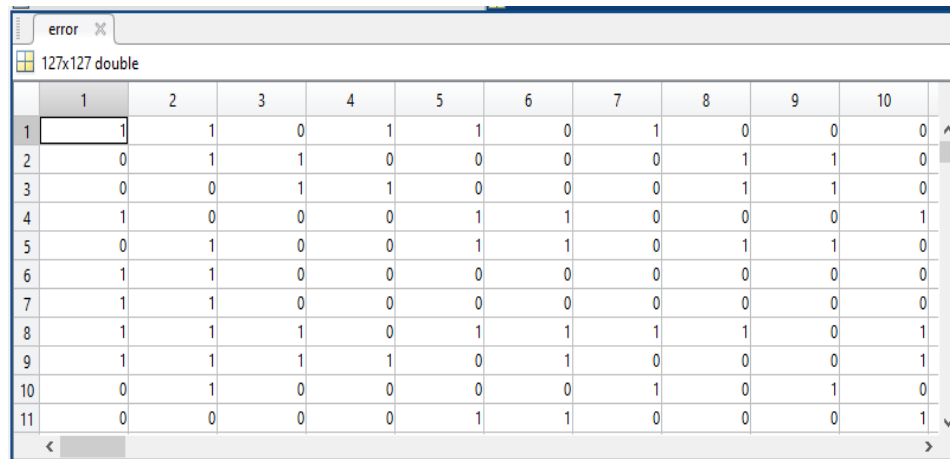


Figure 5.2 - Downsampled Image by Matlab



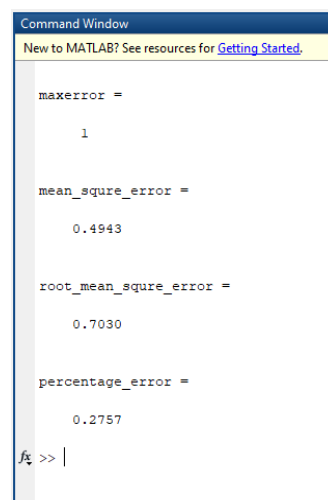
Figure 5.3 - Downsampled Image by Processor

In our error calculation, corresponding pixel value differences were only zeros and ones. Refer the figure 'pixel error array'. Therefore, the maximum pixel value error is one. According to that result, percentage error of the image is about 0.2757%. Since the percentage error is very low, our FPGA implemented processor is successful. That small error can be easily neglected.



	1	2	3	4	5	6	7	8	9	10
1	1	1	0	1	1	0	1	0	0	0
2	0	1	1	0	0	0	0	1	1	0
3	0	0	1	1	0	0	0	1	1	0
4	1	0	0	0	1	1	0	0	0	1
5	0	1	0	0	1	1	0	1	1	0
6	1	1	0	0	0	0	0	0	0	0
7	1	1	0	0	0	0	0	0	0	0
8	1	1	1	0	1	1	1	1	0	1
9	1	1	1	1	0	1	0	0	0	1
10	0	1	0	0	0	0	1	0	1	0
11	0	0	0	0	1	1	0	0	0	1

Figure 5.4 - Error Matrix



```

Command Window
New to MATLAB? See resources for Getting Started.

maxerror =

    1

mean_squre_error =

    0.4943

root_mean_squre_error =

    0.7030

percentage_error =

    0.2757

fx >> |
  
```

Figure 5.5 - Calculated Errors

The reason for the pixel value error is being ones and zeroes is simply due to 16-bit arithmetic operations of ALU. In MATLAB for the arithmetic operations, floating-point arithmetic is used. Therefore, the division operations give rounded approximation for the actual value. However, in our processor for the division it just right shifts 16 bit registers. Therefore, it always is rounded to floor value. If answer is approximated to floor value then error is zero and answer is approximated to ceiling value then error is one.

6. References

[1] Intel® FPGA University Program, University Program Material, Education Boards, and Laboratory Exercises

https://forums.intel.com/s/topic/0TO0P000000MWKHW44/intel-fpga-university-program?language=en_US

[2] Altera DE2-115 user manual.

https://www.intel.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf

[3] Altera My first FPGA design tutorial.

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tt/tt_my_first_fpga.pdf

[4]Peter M. Nyasulu, J Knight, "Introduction to Verilog".

https://www.cs.upc.edu/~jordicf/Teaching/secretsofhardware/VerilogIntroduction_Nyasulu.pdf

[5] FPGA Implementation of an 8-bit Simple Processor

<https://pdfs.semanticscholar.org/4780/a3533269d75e48816418859d25e46bfd27f7.pdf>

7. Appendix

7.1. MATLAB code for down sampling image.

```
%downsampling image using
imGray = rgb2gray(imread('lena.jpeg'));%read the image
DataMemory = zeros(1,65536);%linear data array for input image
outArray = zeros(127,127);%arra for downsampled image
for i = 1:256
    for j = 1:256
        DataMemory(256*(i-1)+j)= imGray(i,j);%generate the input data array
    end
end

for i = 2:2:254
    for j = 2:2:254
        PixelAddress = 256*(i-1)+j;%memory address of centre pixel of the
3*3 gaussian kernel
        NewPixel = 0;%correspodig pixel value of dowsapled image
        %Gaussia blurring and downsampling processes together
        NewPixel = NewPixel + DataMemory(PixelAddress)*4;
        NewPixel = NewPixel + DataMemory(PixelAddress-256)*2;
        NewPixel = NewPixel + DataMemory(PixelAddress-257)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress-255)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress+256)*2;
        NewPixel = NewPixel + DataMemory(PixelAddress+257)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress+255)*1;
        NewPixel = NewPixel + DataMemory(PixelAddress+1)*2;
        NewPixel = NewPixel + DataMemory(PixelAddress-1)*2;
        NewPixel=NewPixel/16;
        outArray(i/2,j/2) = NewPixel;%saving pixel vales of dowsampled
image
    end
end
im_matlab=uint8(outArray);
figure;
imshow(imGray);
title('Original Image');
figure;
imshow(im_matlab);
title('Image Downsampled by MATLAB');
imwrite(im_matlab,'lena_matlab.png');
```

7.2. Python code for write assembly code to Instruction memory

```
assem = {'FETCH':0, 'NOP':43, 'CLAC':13, 'LDAC':4, 'STAC':8,  
'MVACR':14, 'MVACR1':15, 'MVACR2':16, 'MVACR3':17, 'MVACR4':18,  
'MVRAC':19, 'MVR1AC':20, 'MVR2AC':21, 'MVR3AC':22, 'MVR4AC':23,  
'ADDN':24, 'SUBN':27, 'SFTL':30, 'SFTR':33, 'ADD':36, 'SUB':37,  
'MVACAR':38, 'END':44, 'JMNZ':39}  
  
f = open("Assembly.txt")  
file1 = open("ROM.txt", "w+")  
lines = f.readlines()  
for i in range(len(lines)):  
    line=lines[i]  
    #print(line)  
    if line[:-1] in assem:  
        s1="word[%s]=8'd%s;\n"%(i,assem[line[:-1]])  
        file1.write(s1)  
    else:  
        s2="word[%s]=8'd%s;\n"%(i,line[:-1])  
        file1.write(s2)  
file1.close()
```

7.3. MATLAB code for create mif file to write image data to data memory

```
%create mif file
src = imread('lena.jpeg');
gray = rgb2gray(src);
gray=gray';
[m,n] = size( gray ); %size of picture

N = m*n; %ram depth
word_len = 8;
data = reshape(gray, 1, N);% reshape picture's data

fid=fopen('lena.mif', 'w'); % open mif file
fprintf(fid, 'DEPTH=%d;\n', N);
fprintf(fid, 'WIDTH=%d;\n', word_len);

fprintf(fid, 'ADDRESS_RADIX = UNS;\n');
fprintf(fid, 'DATA_RADIX = HEX;\n');
fprintf(fid, 'CONTENT\t');
fprintf(fid, 'BEGIN\n');
for i = 0 : N-1
fprintf(fid, '\t%d\t:\t%x;\n',i, data(i+1));
end
fprintf(fid, 'END;\n'); % printf the end
fclose(fid); % close your file
```

7.4. MATLAB code for make the processed image using hex file generated by processor

```
%generating the image from hex filefrom fpga
im_length=127^2;
im_data=zeros(1,im_length);
fo=fopen('lena_fpga.hex','r');
for i=1:im_length
    line=fgetl(fo);
    data_hex=line(10:11);
    data_dec=(sscanf(data_hex,'%x'));
    im_data(i)=data_dec;
end
fclose(fo);
im_fpga=reshape(im_data,127,127);
im_fpga=uint8(im_fpga);
im_fpga = im_fpga';
figure;
imshow(im_fpga);
title('Image Downsampled by FPGA');
imwrite(im_fpga,'lena_fpga.png');
```

7.5. MATLAB code for calculating error between down sampled image by MATLAB & FPGA

```
%calculating errors
error=abs(double(im_fpga)-double(im_matlab));
maxerror=max(max(error))
sqError=error.^2;
mean_squire_error=sum(sum(sqError))/127^2
root_mean_squire_error=(mean_squire_error)^0.5
percentage_error=root_mean_squire_error*100/255
```

7.6. Verilog implementation of processor

Top module

```
module top_module(enable,finish,clk,AC);
    input enable,clk;
    output finish;
    output [15:0] AC;

    wire [15:0] Bus,ALU,MUX,R,R1,R2,R3,R4,DR,AR;
    wire [7:0] IM,DM,PC,IR;
    wire acw,clac,rw,r1w,r2w,r3w,r4w,irw,pcw,pc_inc,drw,arw,
        imr,dmr,mux,ALUen,dmw,zz,clkd;
    wire [1:0] alu_op;
    wire [3:0] b_flag;

    clock_divider clkdiv(clk,enable,clkd);

    multiplexer10 mux1(AC,ALU,R,R1,R2,R3,R4,IR,IM,DR,DM,b_flag,Bus);

    state_machine StateMachine(clkd,zz,IR,acw,clac,rw,r1w,r2w,r3w,r4w,irw,pcw,
        pc_inc,drw,arw,imr,dmw,dmr,b_flag,alu_op,mux,
        finish,ALUen);

    ALU alu(ALUen,clkd,AC,MUX,alu_op,zz,ALU);

    multiplexer2 mux2(R,IR,mux,MUX);

    PC pc(clkd,pc_inc,pcw,Bus[7:0],PC);

    AC ac(clkd,acw,Bus,AC,clac);

    IR ir(clkd,irw,Bus[7:0],IR);

    register16 r(clkd,rw,Bus,R);
```

```
register16 r1(clkd,r1w,Bus,R1);
register16 r2(clkd,r2w,Bus,R2);
register16 r3(clkd,r3w,Bus,R3);
register16 r4(clkd,r4w,Bus,R4);

register16 ar(clkd,arw,Bus,AR);
register16 dr(clkd,drw,Bus,DR);

ROM ROM1(clkd,imr,PC,IM);
Dram DRAM(.address(AR),.clock(clkd),.data(Bus),.rden(dmr),
           .wren(dmw),.q(DM));

endmodule
```

Clock Divider

```
module clock_divider(clk,enable,clk_out);
    input clk,enable;
    output reg clk_out=1'd0;
    reg [3:0] count=4'd0;
    always@(posedge clk)
        begin
            if (enable )
                begin
                    count=count+4'd1;
                    if (count==4'd10)
                        begin
                            clk_out=~clk_out;
                            count=4'd0;
                        end
                end
        end
end
```

```
endmodule
```

State Machine

```
module state_machine(clk,z,ir,acw,clac,rw,r1w,r2w,r3w,r4w,irw,pcw,pc_inc,
                    drw,arw,imr,dmw,dmr,b_flag,alu_op,mux,finish,aluen)
    ;

    input clk,z;
    input [7:0] ir;
    output reg acw,clac,rw,r1w,r2w,r3w,r4w,irw,pcw,pc_inc,drw,arw,
                imr,dmw,dmr,mux,finish=0,aluen=0;

    output reg [1:0] alu_op;
    output reg [3:0] b_flag;
    reg [5:0] PS=FETCH1;
    reg [5:0] NS=FETCH1;

    parameter
    FETCH1 =6'd1,
    FETCH2 =6'd2,
    FETCH3 =6'd3,

    LDAC1=6'd4,
    LDAC2=6'd5,
    LDAC3=6'd6,
    LDAC4=6'd7,

    STAC1=6'd8,
    STAC2=6'd9,

    CLAC=6'd13,

    MVACR =6'd14,
```

MVACR1 =6'd15,

MVACR2 =6'd16,

MVACR3 =6'd17,

MVACR4 =6'd18,

MVRAC =6'd19,

MVR1AC =6'd20,

MVR2AC =6'd21,

MVR3AC =6'd22,

MVR4AC =6'd23,

ADDN1 =6'd24,

ADDN2 =6'd25,

ADDN3 =6'd26,

ADDN4 =6'd45,

SUBN1 =6'd27,

SUBN2 =6'd28,

SUBN3 =6'd29,

SUBN4 =6'd46,

SFTL1 =6'd30,

SFTL2 =6'd31,

SFTL3 =6'd32,

SFTL4 =6'd47,

SFTR1 =6'd33,

SFTR2 =6'd34,

SFTR3 =6'd35,

SFTR4 =6'd48,

ADD1 =6'd36,

ADD2 =6'd49,

SUB1 =6'd37,

SUB2 =6'd50,

MVACAR =6'd38,

JMNZY1 =6'd39,

JMNZY2=6'd40,

JMNZY3 =6'd41,

JMNZN1 =6'd42,

NOP =6'd43,

END =6'd44;

always@(posedge clk) PS<=NS;

always@(negedge clk)

case (PS)

FETCH1:begin

```
    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=1;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=0;

    alu_op<=0;

    aluen<=0;

    finish<=0;

    NS<=FETCH2;

end

FETCH2:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;
```

```
irw<=1;  
pcw<=0;  
pc_inc<=0;  
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=4'd8;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH3;  
end
```

```
FETCH3:begin  
acw<=0;  
clac<=0;  
rw<=0;  
r1w<=0;  
r2w<=0;  
r3w<=0;  
r4w<=0;  
irw<=0;  
pcw<=0;  
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;
```

```
mux<=0;

b_flag<=0;

alu_op<=0;

aluen<=0;

finish<=0;

if (ir[5:0]==JMNZY1 && z==1)

    begin

        NS<=JMNZN1;

        pc_inc<=1;

    end

else if (ir[5:0]==6'd0) NS<=FETCH1;

else

    begin

        NS<=ir[5:0];

        pc_inc<=1;

    end

end
```

```
CLAC:begin

    acw<=0;

    clac<=1;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;
```

```
pc_inc<=0;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=0;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=FETCH1;

end
```

```
LDAC1:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=1;

    mux<=0;
```

```
b_flag<=0;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=LDAC2;  
end
```

```
LDAC2:begin  
    acw<=0;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=0;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;  
    NS<=LDAC3;  
end
```

```
LDAC3:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=1;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=4'd10;

    alu_op<=0;

    aluen<=0;

    finish<=0;

    NS<=LDAC4;

end
```

```
LDAC4:begin

    acw<=1;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;
```

```
r4w<=0;

irw<=0;

pcw<=0;

pc_inc<=0;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=4'd9;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=FETCH1;

end
```

```
STAC1:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=1;
```



```
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=0;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=STAC2;  
end
```

```
STAC2:begin  
    acw<=0;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=1;  
    dmr<=0;  
    mux<=0;  
    b_flag<=4'd9;  
    alu_op<=0;
```

```
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
MVACR:begin  
    acw<=0;  
    clac<=0;  
    rw<=1;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=0;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;  
    NS<=FETCH1;  
end
```

```
MVACR1:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=1;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=0;

    alu_op<=0;

    aluen<=0;

    finish<=0;

    NS<=FETCH1;

end
```

```
MVACR2:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=1;

    r3w<=0;
```

```
r4w<=0;  
irw<=0;  
pcw<=0;  
pc_inc<=0;  
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=0;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
MVACR3:begin  
acw<=0;  
clac<=0;  
rw<=0;  
r1w<=0;  
r2w<=0;  
r3w<=1;  
r4w<=0;  
irw<=0;  
pcw<=0;  
pc_inc<=0;  
drw<=0;  
arw<=0;  
imr<=0;
```

```
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=0;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
MVACR4:begin  
    acw<=0;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=1;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=0;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;
```

```
NS<=FETCH1;
```

```
end
```

```
MVRAC:begin
```

```
acw<=1;
```

```
clac<=0;
```

```
rw<=0;
```

```
r1w<=0;
```

```
r2w<=0;
```

```
r3w<=0;
```

```
r4w<=0;
```

```
irw<=0;
```

```
pcw<=0;
```

```
pc_inc<=0;
```

```
drw<=0;
```

```
arw<=0;
```

```
imr<=0;
```

```
dmw<=0;
```

```
dmr<=0;
```

```
mux<=0;
```

```
b_flag<=4'd2;
```

```
alu_op<=0;
```

```
aluen<=0;
```

```
finish<=0;
```

```
NS<=FETCH1;
```

```
end
```

```
MVR1AC:begin
```

```
acw<=1;
```

```
clac<=0;
```

```
rw<=0;
```

```
r1w<=0;  
r2w<=0;  
r3w<=0;  
r4w<=0;  
irw<=0;  
pcw<=0;  
pc_inc<=0;  
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=4'd3;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
MVR2AC:begin  
    acw<=1;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;
```

```
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=4'd4;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
MVR3AC:begin  
    acw<=1;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=4'd5;
```



```
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
MVR4AC:begin  
    acw<=1;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=4'd6;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;  
    NS<=FETCH1;  
end
```

```
ADDN1:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=1;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=0;

    alu_op<=0;

    aluen<=0;

    finish<=0;

    NS<=ADDN2;

end
```

```
ADDN2:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;
```

```
r4w<=0;

irw<=1;

pcw<=0;

pc_inc<=0;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=4'd8;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=ADDN3;

end
```

```
ADDN3:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=1;

    drw<=0;

    arw<=0;

    imr<=0;
```

```
dmw<=0;  
dmr<=0;  
mux<=1;  
b_flag<=0;  
alu_op<=0;  
aluen<=1;  
finish<=0;  
NS<=ADDN4;  
end
```

```
ADDN4:begin  
    acw<=1;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=4'd1;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;
```

```
NS<=FETCH1;
```

```
end
```

```
SUBN1:begin
```

```
    acw<=0;
```

```
    clac<=0;
```

```
    rw<=0;
```

```
    r1w<=0;
```

```
    r2w<=0;
```

```
    r3w<=0;
```

```
    r4w<=0;
```

```
    irw<=0;
```

```
    pcw<=0;
```

```
    pc_inc<=0;
```

```
    drw<=0;
```

```
    arw<=0;
```

```
    imr<=1;
```

```
    dmw<=0;
```

```
    dmr<=0;
```

```
    mux<=0;
```

```
    b_flag<=0;
```

```
    alu_op<=0;
```

```
    aluen<=0;
```

```
    finish<=0;
```

```
NS<=SUBN2;
```

```
end
```

```
SUBN2:begin
```

```
    acw<=0;
```

```
    clac<=0;
```

```
    rw<=0;
```

```
r1w<=0;

r2w<=0;

r3w<=0;

r4w<=0;

irw<=1;

pcw<=0;

pc_inc<=0;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=4'd8;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=SUBN3;

end
```

```
SUBN3:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=1;
```

```
drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=1;

b_flag<=0;

alu_op<=2'd1;

aluen<=1;

finish<=0;

NS<=SUBN4;

end
```

```
SUBN4:begin

    acw<=1;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=4'd1;
```

```
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
SFTL1:begin  
    acw<=0;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=1;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=0;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;  
    NS<=SFTL2;  
end
```

```
SFTL2:begin
```



```
acw<=0;

clac<=0;

rw<=0;

r1w<=0;

r2w<=0;

r3w<=0;

r4w<=0;

irw<=1;

pcw<=0;

pc_inc<=0;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=4'd8;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=SFTL3;

end
```

```
SFTL3:begin
```

```
acw<=0;

clac<=0;

rw<=0;

r1w<=0;

r2w<=0;

r3w<=0;

r4w<=0;
```

```
irw<=0;

pcw<=0;

pc_inc<=1;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=1;

b_flag<=0;

alu_op<=2'd2;

aluen<=1;

finish<=0;

NS<=SFTL4;

end
```

```
SFTL4:begin

    acw<=1;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;
```

```
dmr<=0;  
mux<=0;  
b_flag<=4'd1;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
SFTR1:begin  
    acw<=0;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=1;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=0;  
    alu_op<=0;  
    aluen<=0;  
    finish<=0;
```

```
NS<=SFTR2;
```

```
end
```

```
SFTR2:begin
```

```
    acw<=0;
```

```
    clac<=0;
```

```
    rw<=0;
```

```
    r1w<=0;
```

```
    r2w<=0;
```

```
    r3w<=0;
```

```
    r4w<=0;
```

```
    irw<=1;
```

```
    pcw<=0;
```

```
    pc_inc<=0;
```

```
    drw<=0;
```

```
    arw<=0;
```

```
    imr<=0;
```

```
    dmw<=0;
```

```
    dmr<=0;
```

```
    mux<=0;
```

```
    b_flag<=4'd8;
```

```
    alu_op<=0;
```

```
    aluen<=0;
```

```
    finish<=0;
```

```
NS<=SFTR3;
```

```
end
```

```
SFTR3:begin
```

```
    acw<=0;
```

```
    clac<=0;
```

```
    rw<=0;
```

```
r1w<=0;  
r2w<=0;  
r3w<=0;  
r4w<=0;  
irw<=0;  
pcw<=0;  
pc_inc<=1;  
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=1;  
b_flag<=0;  
alu_op<=2'd3;  
aluen<=1;  
finish<=0;  
NS<=SFTR4;  
end
```

```
SFTR4:begin  
    acw<=1;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;
```

```
drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=4'd1;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=FETCH1;

end
```

```
ADD1:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=0;
```

```
alu_op<=0;  
alu_en<=1;  
finish<=0;  
NS<=ADD2;  
end
```

```
ADD2:begin  
    acw<=1;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=0;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=4'd1;  
    alu_op<=0;  
    alu_en<=0;  
    finish<=0;  
    NS<=FETCH1;  
end
```

```
SUB1:begin
```

```
    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=0;

    alu_op<=2'd1;

    aluen<=1;

    finish<=0;

    NS<=SUB2;

    end

SUB2:begin

    acw<=1;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;
```



```
irw<=0;

pcw<=0;

pc_inc<=0;

drw<=0;

arw<=0;

imr<=0;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=4'd1;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=FETCH1;

end
```

```
MVACAR:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=1;
```

```
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=0;  
alu_op<=0;  
aluen<=0;  
finish<=0;  
NS<=FETCH1;  
end
```

```
JMNZY1:begin  
    acw<=0;  
    clac<=0;  
    rw<=0;  
    r1w<=0;  
    r2w<=0;  
    r3w<=0;  
    r4w<=0;  
    irw<=0;  
    pcw<=0;  
    pc_inc<=0;  
    drw<=0;  
    arw<=0;  
    imr<=1;  
    dmw<=0;  
    dmr<=0;  
    mux<=0;  
    b_flag<=0;  
    alu_op<=0;
```

```
    aluen<=0;

    finish<=0;

    NS<=JMNZY2;

    end

JMNZY2:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=1;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=4'd8;

    alu_op<=0;

    aluen<=0;

    finish<=0;

    NS<=JMNZY3;

    end

JMNZY3:begin

    acw<=0;
```

```
    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=1;

    pc_inc<=0;

    drw<=0;

    arw<=0;

    imr<=0;

    dmw<=0;

    dmr<=0;

    mux<=0;

    b_flag<=4'd7;

    alu_op<=0;

    aluen<=0;

    finish<=0;

    NS<=FETCH1;

end

//JMNZY4:NS<=FETCH1;

JMNZN1:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;
```

```
r3w<=0;

r4w<=0;

irw<=0;

pcw<=0;

pc_inc<=1;

drw<=0;

arw<=0;

imr<=1;

dmw<=0;

dmr<=0;

mux<=0;

b_flag<=0;

alu_op<=0;

aluen<=0;

finish<=0;

NS<=FETCH1;

end
```

```
END:begin

    acw<=0;

    clac<=0;

    rw<=0;

    r1w<=0;

    r2w<=0;

    r3w<=0;

    r4w<=0;

    irw<=0;

    pcw<=0;

    pc_inc<=0;

    drw<=0;

    arw<=0;
```

```
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=0;  
alu_op<=0;  
aluen<=0;  
finish<=1;  
NS<=FETCH1;  
end
```

```
NOP:begin  
acw<=0;  
clac<=0;  
rw<=0;  
r1w<=0;  
r2w<=0;  
r3w<=0;  
r4w<=0;  
irw<=0;  
pcw<=0;  
pc_inc<=0;  
drw<=0;  
arw<=0;  
imr<=0;  
dmw<=0;  
dmr<=0;  
mux<=0;  
b_flag<=0;  
alu_op<=0;  
aluen<=0;
```

```
        finish<=1;

        NS<=NOP;

    end

endcase

endmodule
```

ALU

```
module ALU(enable,clk,A_bus,B_bus,ALU_op,z,C_bus);

    input [15:0] A_bus,B_bus;

    input [1:0] ALU_op;

    input enable,clk;

    output reg z=1'd0;

    output reg [15:0] C_bus;


    always@(posedge clk)

    begin

        if (enable==1)

            begin

                case (ALU_op)

                    2'd0: C_bus<=A_bus+B_bus;//ADD

                    2'd1: begin

                                C_bus<=A_bus-B_bus;//SUBTRACT

                                z<=((A_bus-B_bus)==16'd0)? 1'd1 : 1'd0;

                            end

                    2'd2: C_bus<=A_bus<<B_bus;//SHIFT LEFT

                    2'd3: C_bus<=A_bus>>B_bus;//SHIFT RIGHT

                endcase

            end

        end

    end
```

```
        end

    end

endmodule
```

Instruction Memory

```
module ROM(clk,read,address,data_out);

    input clk,read;
    input [7:0] address;
    output reg [7:0] data_out;
    reg [7:0] word[255:0];

    initial
    begin
        word[0]=8'd13;
        word[1]=8'd14;
        word[2]=8'd15;
        word[3]=8'd16;
        word[4]=8'd17;
        word[5]=8'd18;
        word[6]=8'd22;
        word[7]=8'd24;
        word[8]=8'd2;
        word[9]=8'd17;
        word[10]=8'd13;
        word[11]=8'd18;
        word[12]=8'd23;
        word[13]=8'd24;
        word[14]=8'd2;
        word[15]=8'd18;
        word[16]=8'd22;
```



```
word[17]=8'd30;  
word[18]=8'd8;  
word[19]=8'd14;  
word[20]=8'd23;  
word[21]=8'd36;  
word[22]=8'd27;  
word[23]=8'd255;  
word[24]=8'd27;  
word[25]=8'd2;  
word[26]=8'd15;  
word[27]=8'd38;  
word[28]=8'd4;  
word[29]=8'd30;  
word[30]=8'd2;  
word[31]=8'd14;  
word[32]=8'd20;  
word[33]=8'd27;  
word[34]=8'd255;  
word[35]=8'd27;  
word[36]=8'd1;  
word[37]=8'd38;  
word[38]=8'd4;  
word[39]=8'd30;  
word[40]=8'd1;  
word[41]=8'd36;  
word[42]=8'd14;  
word[43]=8'd20;  
word[44]=8'd27;  
word[45]=8'd255;  
word[46]=8'd27;  
word[47]=8'd2;
```

```
word[48]=8'd38;  
word[49]=8'd4;  
word[50]=8'd36;  
word[51]=8'd14;  
word[52]=8'd20;  
word[53]=8'd27;  
word[54]=8'd255;  
word[55]=8'd38;  
word[56]=8'd4;  
word[57]=8'd36;  
word[58]=8'd14;  
word[59]=8'd20;  
word[60]=8'd24;  
word[61]=8'd255;  
word[62]=8'd24;  
word[63]=8'd1;  
word[64]=8'd38;  
word[65]=8'd4;  
word[66]=8'd30;  
word[67]=8'd1;  
word[68]=8'd36;  
word[69]=8'd14;  
word[70]=8'd20;  
word[71]=8'd24;  
word[72]=8'd255;  
word[73]=8'd24;  
word[74]=8'd2;  
word[75]=8'd38;  
word[76]=8'd4;  
word[77]=8'd36;  
word[78]=8'd14;
```

```
word[79]=8'd20;  
word[80]=8'd24;  
word[81]=8'd255;  
word[82]=8'd38;  
word[83]=8'd4;  
word[84]=8'd36;  
word[85]=8'd14;  
word[86]=8'd20;  
word[87]=8'd24;  
word[88]=8'd1;  
word[89]=8'd38;  
word[90]=8'd4;  
word[91]=8'd30;  
word[92]=8'd1;  
word[93]=8'd36;  
word[94]=8'd14;  
word[95]=8'd20;  
word[96]=8'd27;  
word[97]=8'd1;  
word[98]=8'd38;  
word[99]=8'd4;  
word[100]=8'd30;  
word[101]=8'd1;  
word[102]=8'd36;  
word[103]=8'd33;  
word[104]=8'd4;  
word[105]=8'd14;  
word[106]=8'd21;  
word[107]=8'd38;  
word[108]=8'd19;  
word[109]=8'd8;
```

```
word[110]=8'd21;
word[111]=8'd24;
word[112]=8'd1;
word[113]=8'd16;
word[114]=8'd23;
word[115]=8'd27;
word[116]=8'd254;
word[117]=8'd39;
word[118]=8'd12;
word[119]=8'd22;
word[120]=8'd27;
word[121]=8'd254;
word[122]=8'd39;
word[123]=8'd6;
word[124]=8'd44;
word[125]=8'd43;
end
always@(posedge clk) if(read) data_out<=word[address];
endmodule
```

Register 16bit

```
module register16(clk,write,data_in,data_out);
    input clk,write;
    input [15:0] data_in;
    output reg [15:0] data_out;
    always@(posedge clk)
        if(write) data_out<=data_in;
endmodule
```

Instruction Register

```
module IR(clk,write,data_in,data_out);  
    input clk, write;  
    input [7:0] data_in;  
    output reg [7:0] data_out;  
    always@(posedge clk)  
        if(write) data_out<=data_in;  
endmodule
```

Programme Counter

```
module PC(clk,inc,write,data_in,data_out);  
    input clk,inc,write;  
    input [7:0] data_in;  
    output reg [7:0] data_out=8'd0;  
    always@(posedge clk)  
        begin  
            if(write) data_out<=data_in;  
            else if (inc) data_out<=data_out+8'd1;  
        end  
endmodule
```

Multiplexer_2

```
module multiplexer2(R,IR,sw,ALU);  
    input [15:0] R;  
    input [7:0] IR;  
    input sw;  
    output reg[15:0] ALU;  
    always@(R or IR or sw)  
        begin  
            case (sw)  
                1'd0: ALU= R;  
            endcase  
        end  
endmodule
```

```
        1'd1:  ALU= {8'd0,IR};

    endcase

end

endmodule
```

Multiplexer_10

```
module multiplexer10 (AC,ALU,R,R1,R2,R3,R4,IR,IM,DR,DM,m_flag,B_bus);

    input [15:0] AC,ALU,R,R1,R2,R3,R4,DR;

    input [7:0] DM,IM,IR;

    input [3:0] m_flag;

    output reg[15:0] B_bus;

    always@(m_flag or AC or ALU or R or R1 or R2 or R3 or R4 or IR or IM or DR
    or DM)

begin

    case(m_flag)

        4'd0:  B_bus= AC;

        4'd1:  B_bus= ALU;

        4'd2:  B_bus= R;

        4'd3:  B_bus= R1;

        4'd4:  B_bus= R2;

        4'd5:  B_bus= R3;

        4'd6:  B_bus= R4;

        4'd7:  B_bus= {8'd0,IR};

        4'd8:  B_bus= {8'd0,IM};

        4'd9:  B_bus= DR;

        4'd10: B_bus= {8'd0,DM};[

    endcase

end

endmodule
```

