Ms. Embedded Systems - 2019
UPPSALA UNIVERSITY
Digital Electronics Design with VHDL

16-BIT MICROPROCESSOR

DIANESWARR SHANMUGA SUNDARAM

UPPSALA
UNIVERSITET

29-04-2020

Contents

1. ABSTRACT

The project task is to design a 16- bit Microprocessor kernel which fetch, decode and execute Add, Store, Load, Jump, Jneg and NOP instruction and to show software simulation and hardware implementation using DE1-SoC.

2. INTRODUCTION

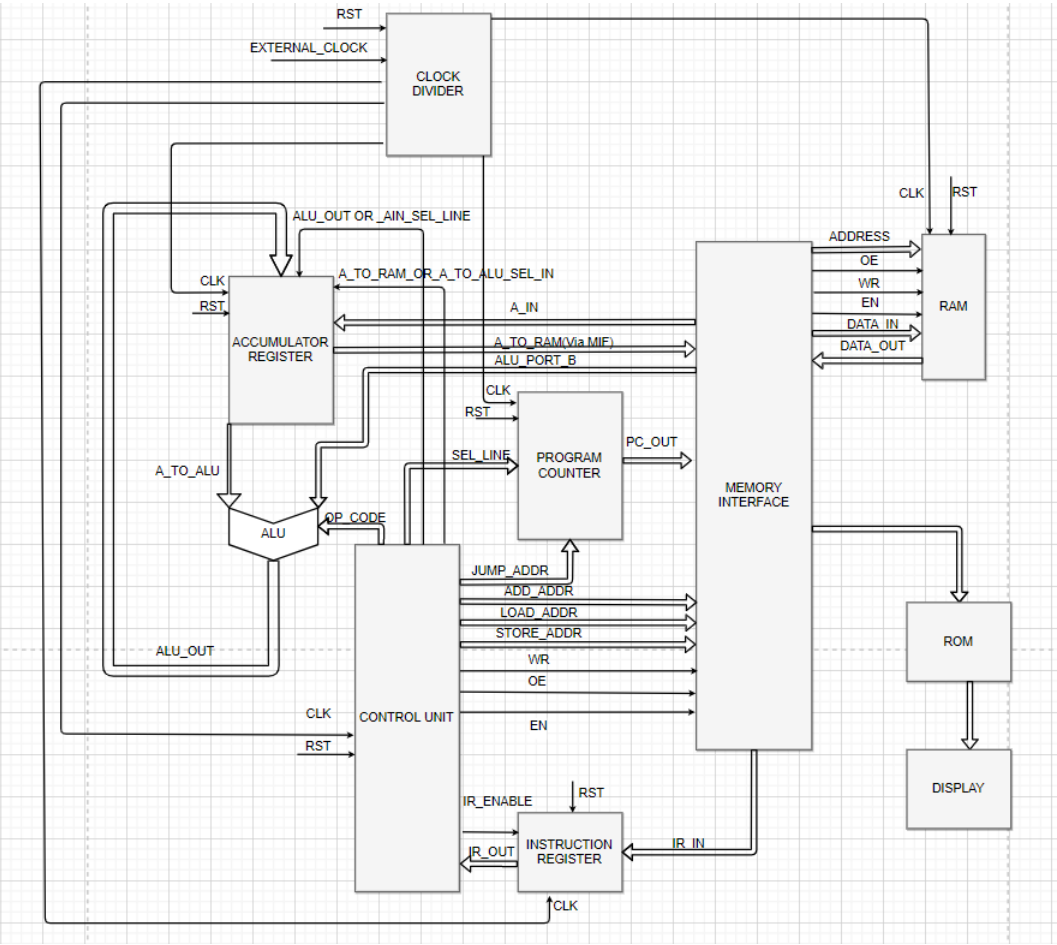

Figure 2.1: **Principle Drawing of 16 bit Micro processor**

A basic microprocessor will have program counter register (PC), instruction register(IR), Accumulator (for storing intermediate result),ALU,MDR and MAR but in this project we exclude MDR and we build an Memory interface(MIF), which is a input- output port, makes connection with the external Ram memory and hexdecimal display to display output. The length of data is 16-bit, length of instruction is 16-bit, instruction consists of 8-bit operand and 8-bit address.

3. PROJECT DESCRIPTION

The process which is happening in the 16-bit microprocessor is explained below.

The fetch, decode and execution cycle is executed inside the microprocessor. The instructions are stored in ram memory, during fetch cycle the instruction is read from the ram and it is send to Instruction register(IR) through Memory Interface(MIF) and program counter is incremented and after that the instruction which of 16-bit, which consists of 8-bit operand and 8-bit address are decoded in the decode cycle in the control unit. After the decode cycle, it moves to execution cycle where add, store, load, jump, jneg or nop operation are performed. Afer the execution cycle, it moves to fetch cycle and the cycle repeats.

When entering load cycle, the data is read from the ram and stored in the accumulator.

When entering add cycle, the data is read from the ram and move to the Alu port B, then add is performed and the result is stored in accumulator.

When entering store cycle, the data from the accumulator is send to the ram through MIF interface.

When entering jump cycle, after the decoding is done the 8-bit address is fed directly to the program counter from the control unit and the program counter jumps to the jump address

Before entering Jneg cycle, it checks for overflow and not zero, if that is true then it enter the Jneg state else it will enter fetch1 state, after entering into the jneg state, 8-bit address is fed directly to the program counter from the control unit and the program counter jumps to the jneg address

When entering NOP state, 8-bit address is fed directly to the program counter from the control unit and stays in that address, so that we could see the resulted output.

		Op-Code
ADD address	**AC <= AC + Content of the memory address**	**00**
STORE address	**Content of the memory<=AC address**	**01**
LOAD address	**AC <= Content of the memory address**	**02**

JUMP address	**PC <= Address**	**03**
JNEG address	**IF AC <0 then PC <= address**	**04**

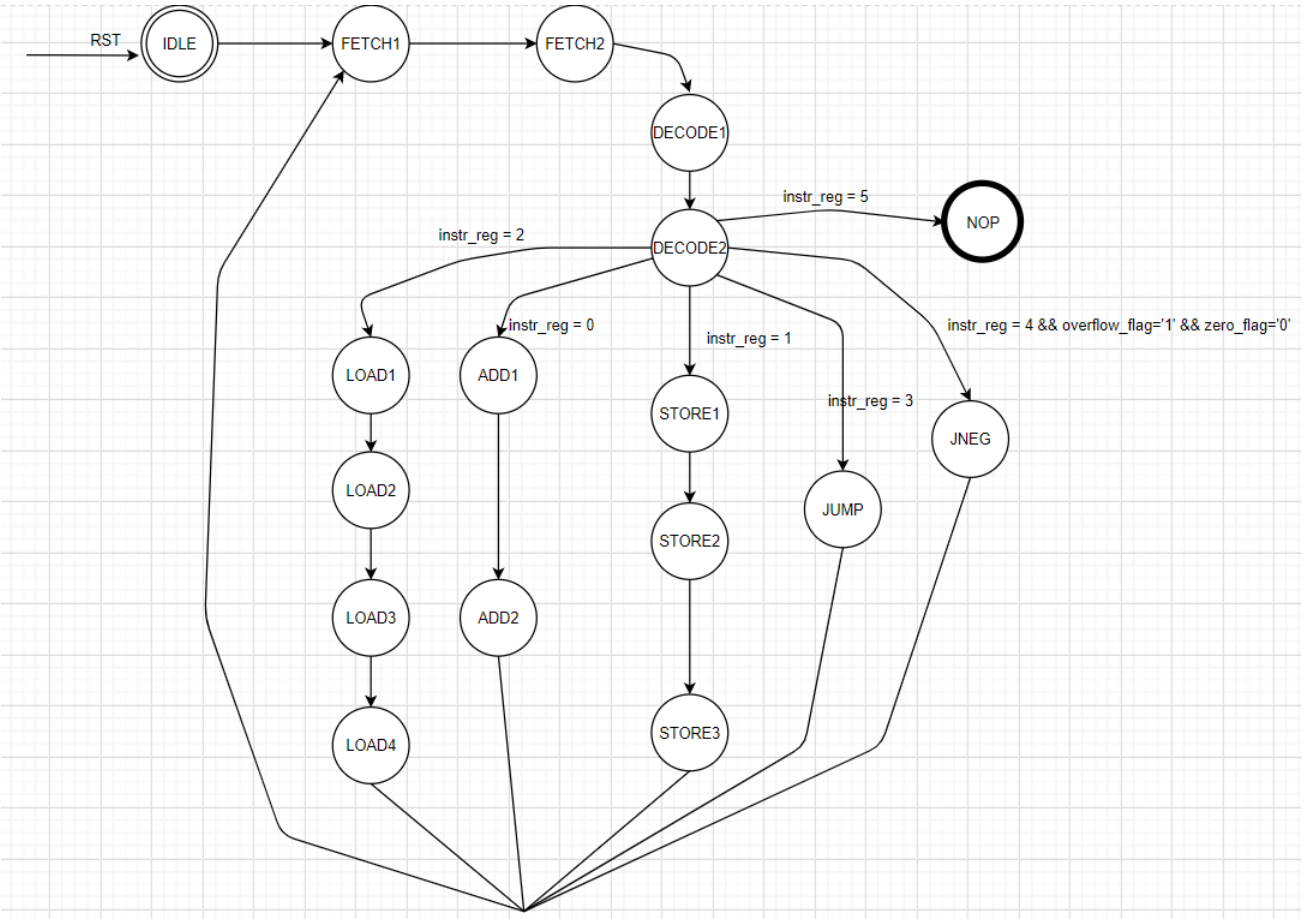

Figure 3.1: **STATE MACHINE**

NOP address	**No Operation**	**05**

I have implemented using 2 cycles of fetch, 4 cycles of load, 2 cycles of add, 3 cycles of store, 1 cycle jump, 1 cycle jneg and 1 cycle NOP(no operation). The state machine is shown in figure 3.1.

4. 16-BIT MICROPROCESSOR ARCHITECTURE IMPLEMENTATION

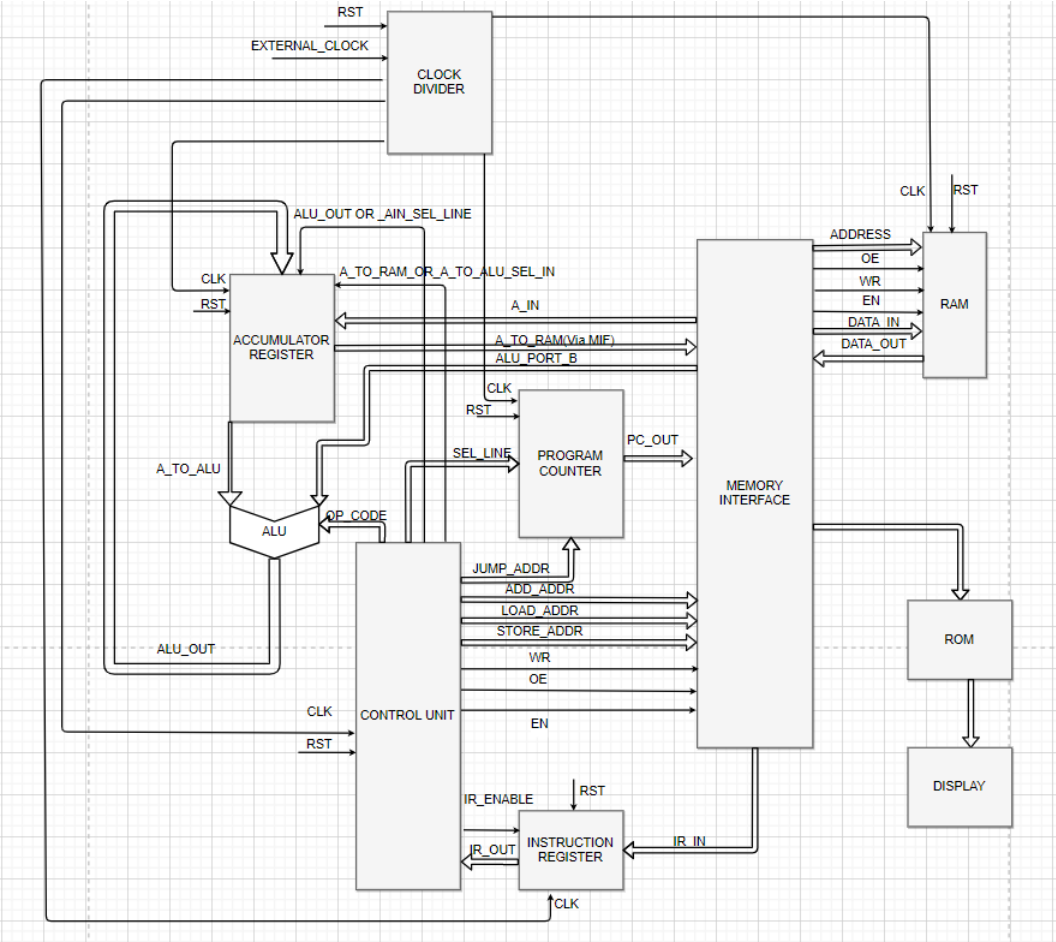

Figure 4.1: **Micro-processor internal architecture**

The figure 4.1 shows the internal architecture of the 16-bit Microprocessor and connection with external ram and rom. The instructions and data are stored in the Ram. The instruction stored in the ram is extracted via MIF interface and send to the instruction register, from there it is passed to the control unit, at the decode cycle control unit decodes the instruction, the

instruction (16-bit) is decoded into opcode (8-bit) and the address (8-bit), depending on the opcode the data is fetched from the address (8-bit) and load, store, add,jump,jneg and nop (No operation) is processed. When program counter reaches the nop instruction, it halts in the address encoded in the nop instructions and data stored at that address is displayed.

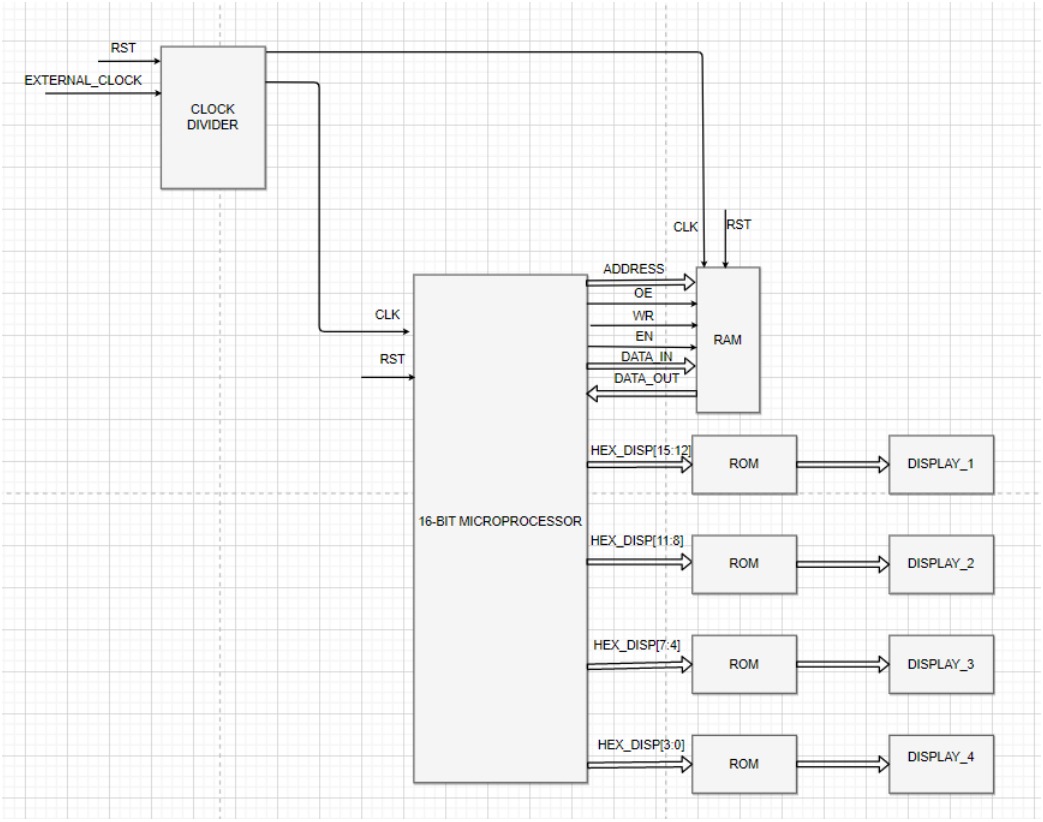

Figure 4.2: **External architecture**

The figure 4.2 shows the external architecture of the 16-bit Microprocessor. The microprocessor is connected with ram and rom. The microprocessor and ram runs at the one fourth frequency of the system clock, this is achieved using clock divider or prescaler. The data from rom triggers the hexadecimal display.

5. IMPLEMENTATION AND SIMULATION

NOTE: I have tested and simulated all components in the same file, thats the reason why you would see program counter name (please ignore it).

5.1 RAM

RAM			
External signals	Port	Width (In bits)	Description
Clock	Input	1	Clock signal (12.5MHz)
Reset	Input	1	Global Reset signal from the press button
WriteEn	Input	1	Write enable signal
OE	Input	1	Read enable signal
Enable	Input	1	Enable signal (to turn on the ram)
DataIn	Input	16	16-bit data from the microprocessor
Address	Input	8	Address in
DataOut	Output	16	16-bit data from the ram to processor

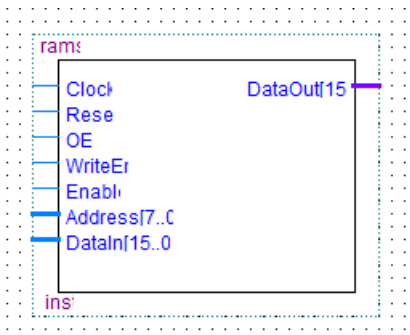

Figure 5.1: **RAM BSF**

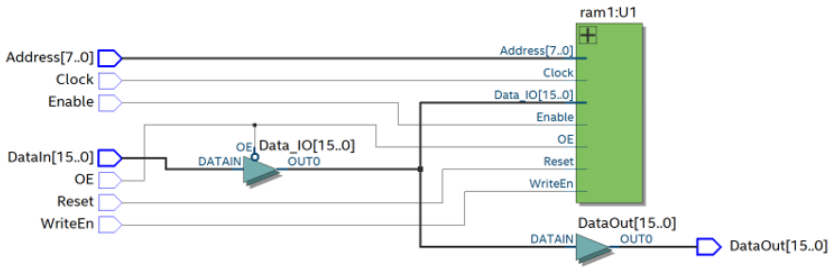

Figure 5.2: **RAM RTL**

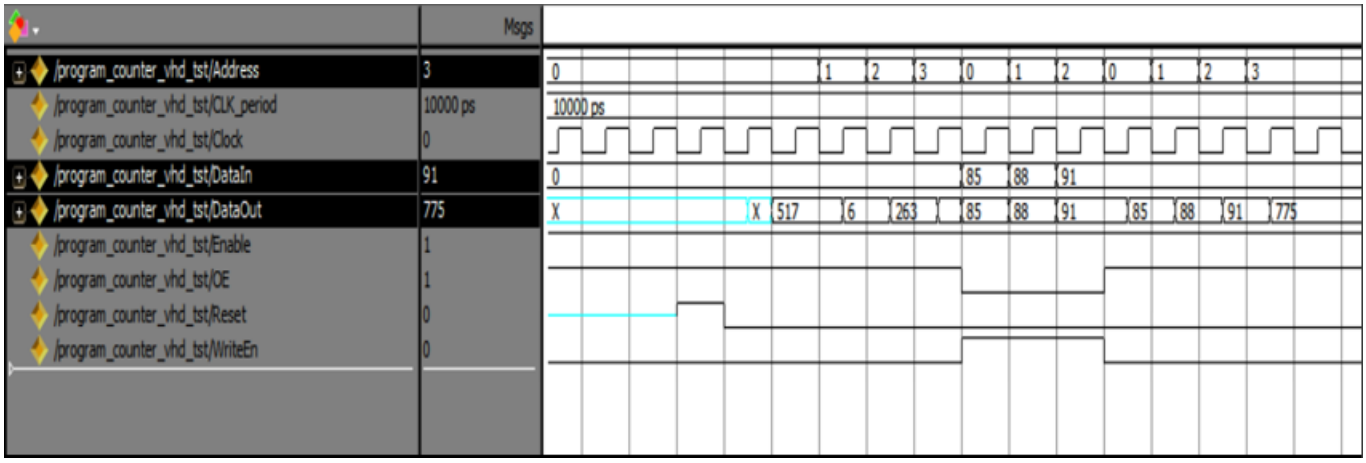

Figure 5.3: **RAM SIMULATION**

The Ram contains the instructions and data. Whenever the program counter is pointed to the specific address the instruction is fetched from that address and depending upon the instructions the data is read from or written to ram.

5.2 INSTRUCTION REGISTER

INSTRUCTION REGISTER			
External signals	Port	Width (In bits)	Description
Clock	Input	1	Clock signal (12.5MHz)
Reset	Input	1	Global Reset signal from the press button
IR_enable	Input	1	Enable signal
IR_INstruc	Input	16	16 bit instruction input
IR_out	Output	16	16-bit instruction output

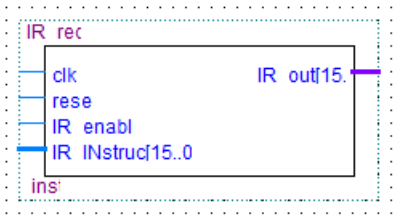

Figure 5.4: IR REG BSF

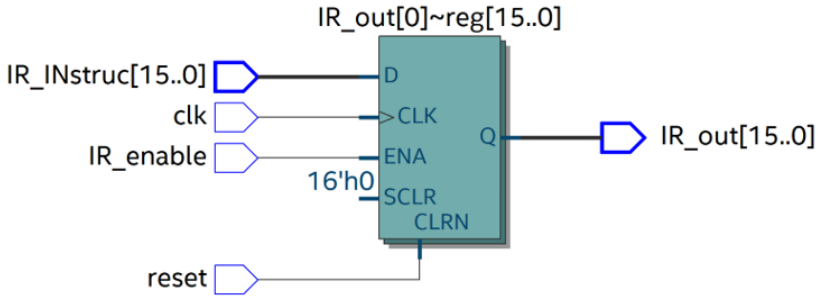

Figure 5.5: IR REG RTL

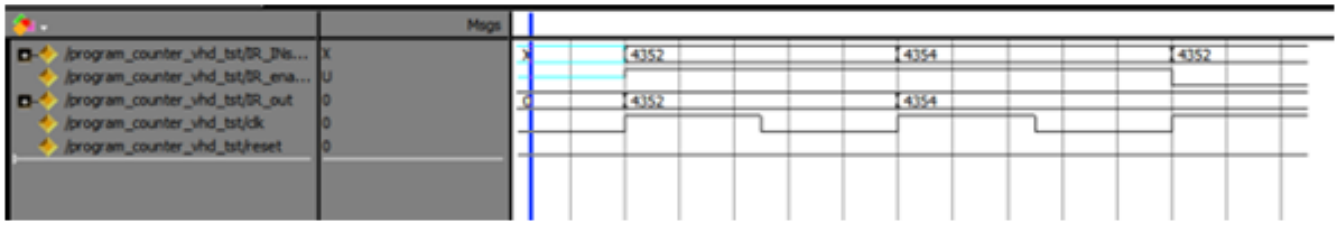

Figure 5.6: IR REG SIMULATION

When the program counter points to the specific address, the instruction from that address is sent from the ram to instruction register. At the fetch2 cycle an enable signal is sent from the control unit to the instruction register,always at positive edge of the clock, if instruction register is enabled then the instruction is passed to the control unit.

5.3 ACCUMULATOR REGISTER

ACCUMULATOR REGISTER			
External signals	Port	Width (In bits)	Description
Clock	Input	1	Clock signal (12.5MHz)
Reset	Input	1	Global Reset signal from the press button
A_enable	Input	1	Enable signal
A_in1	Input	16	16 bit data from ram
ALUto_a_or_a_to_ALU	Input	1	1-bit Selection line
ALU_out	Input	16	16-bit data input from the ALU
A_to_ram	Output	16	16 bit data output from A register to Ram
A_to_ALU	Output	16	16 bit data output from A register to ALU

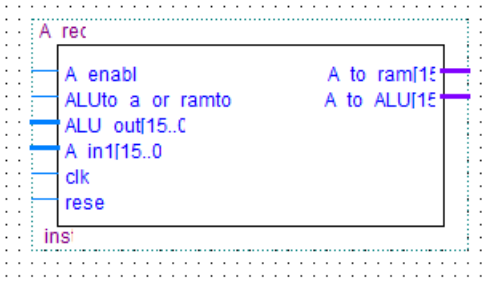

Figure 5.7: A REG BSF

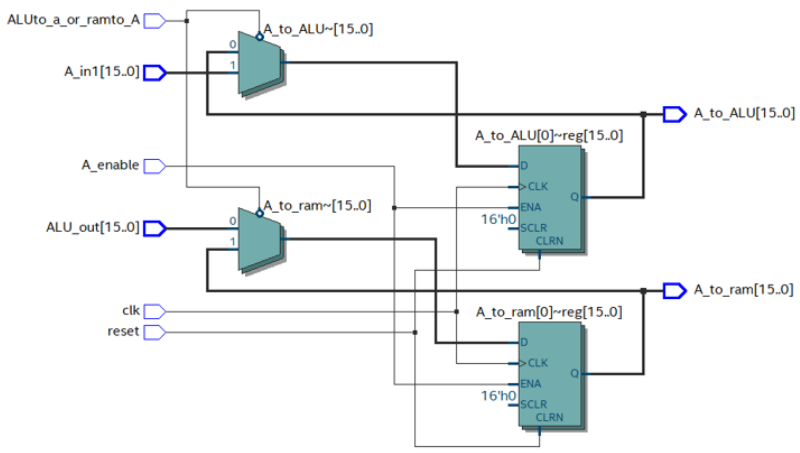

Figure 5.8: A REG RTL

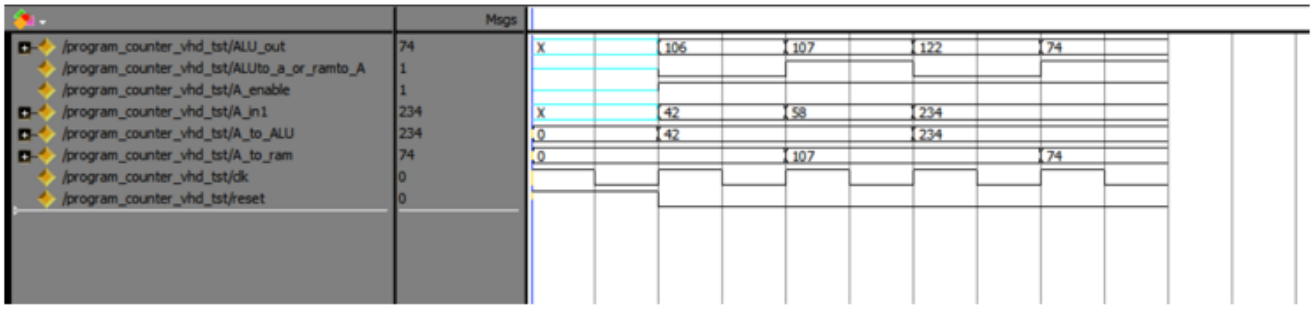

Figure 5.9: A REG SIMULATION

It is a 16-bit data register. The intermediate results are stored in this registers. The Arithmetic logic unit computes the result and the result is stored in the accumulator register.

5.4 ARITHMETIC LOGIC UNIT

ARITHMETIC LOGIC UNIT			
External signals	Port	Width (In bits)	Description
Clock	Input	1	Clock signal (12.5MHz)
Reset	Input	1	Global Reset signal from the press button
a_in	Input	16	16 bit input data from A register
b_in	Input	16	16 bit input data from RAM
op_code	Input	3	selection line to select operations
ALU_out	Output	16	16-bit data output to the A register
Zero	Output	1	Zero flag
Overflow	Output	1	Overflow flag

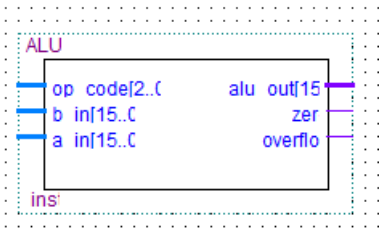

Figure 5.10: ALU BSF

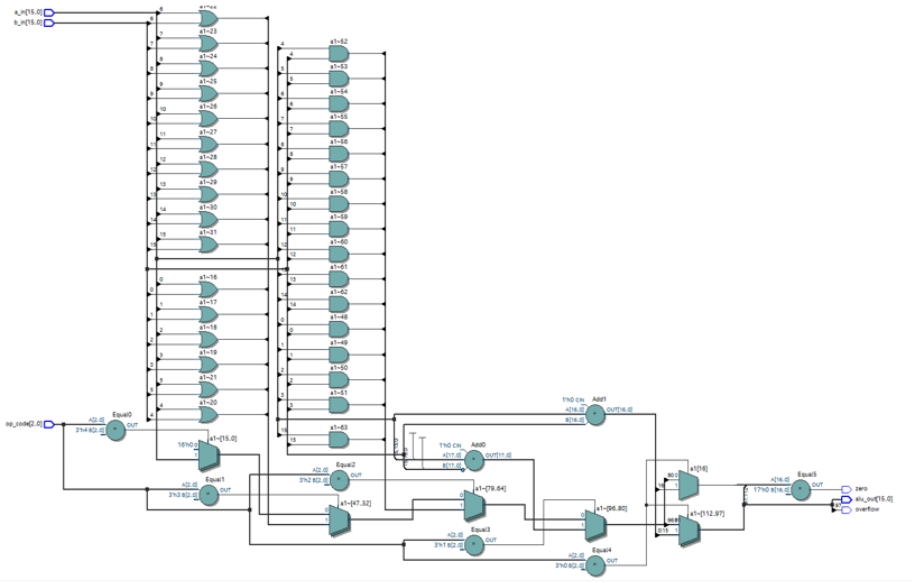

Figure 5.11: ALU RTL

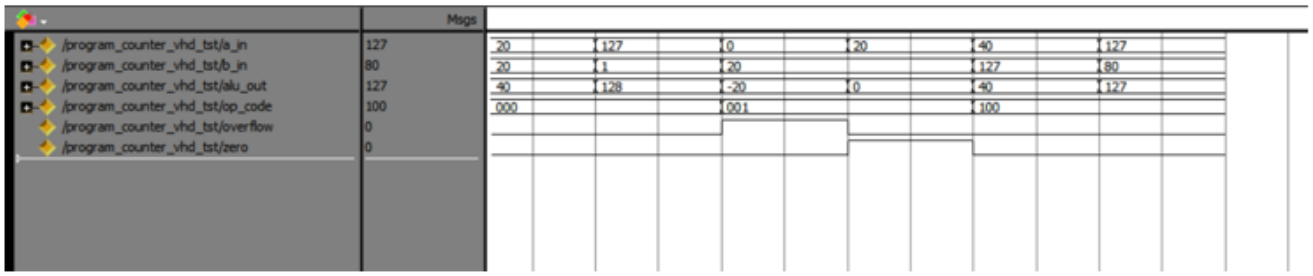

Figure 5.12: ALU SIMULATION

It is the unit where the calculation is computed and it detects the overflow and zero, the type of computation is decided by select signal from the control unit.

5.5 PROGRAM COUNTER

PROGRAM COUNTER			
External signals	Port	Width (In bits)	Description
clk	Input	1	Clock signal (12.5MHz)
reset	Input	1	Global Reset signal from the press button
Jump_to_address	Input	8	8 bit address from control unit
op_code	Input	2	selection line to select operations
pc_out	Output	8	8 bit address

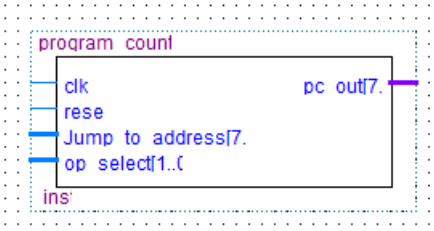

Figure 5.13: PROGRAM COUNTER BSF

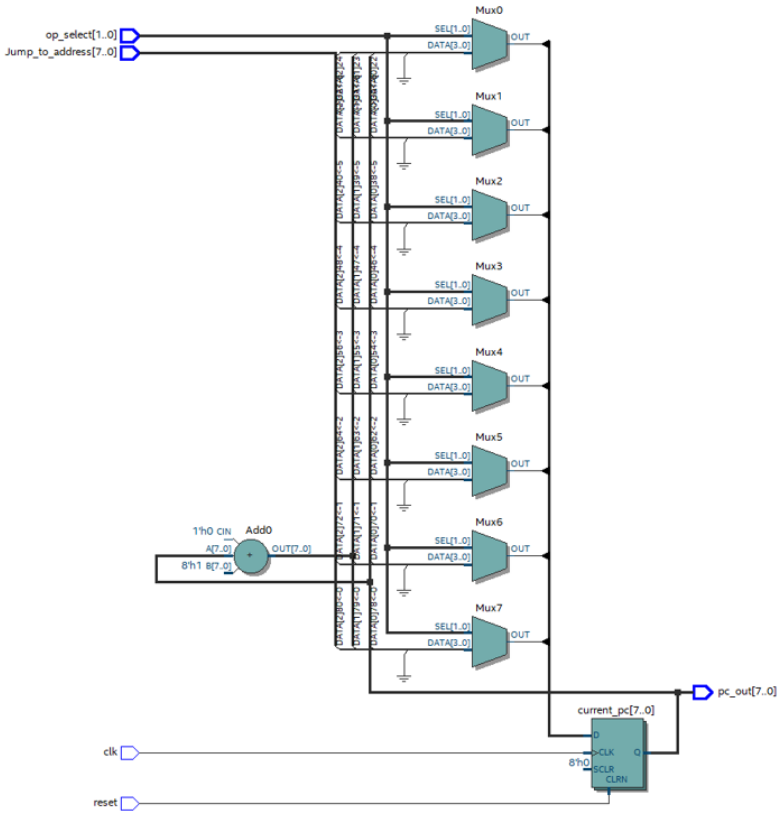

Figure 5.14: PROGRAM COUNTER RTL

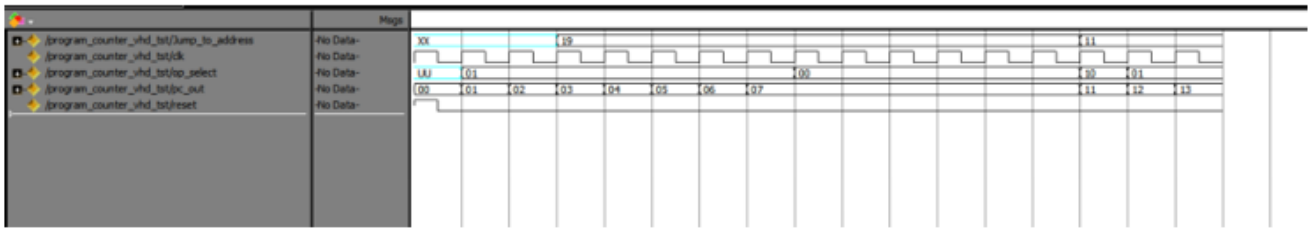

Figure 5.15: PROGRAM COUNTER SIMULATION

When the program counter points to the specific address, the instruction stored in that address is sent from the ram to instruction register. As each instruction gets fetched, the program counter increases its stored value by 1. When Jump,Jneg and nop instruction is performed program counter jumps to the specified address.

5.6 MEMORY INTERFACE(I/O)

MEMORY INTERFACE			
External signals	Port	Width (In bits)	Description
A_to_MIF_data	Input	16	16-bit data from accumulator
sel_pc_str_ld_add_nop	Input	2	Select line to select load, store,add,nop or program counter address
pc_address	Input	8	8 bit address from program counter
OE	Input	1	Read enable signal
wr	Input	1	Write enable signal
en	Input	1	Enable signal to enable the ram
add_address	Input	8	8- bit add address
load_store_address	Input	8	8-bit load address
Nop_address	Input	8	8 bit no operation address
data_out	Input	16	16 bit data or instruction from the ram
A_in	Input	16	16 bit data from the ram to A register
IR_in	Output	16	16 bit instruction from ram to IR register
ALU_portb_in	Output	16	16 bit data from ram to port b of ALU
MIF_to_mem	Output	16	16 bit data to the ram
OE1	Output	1	Read signal
wr1	Output	1	Write signal
en1	Output	1	Enable signal
Address	Output	8	8 bit address
hexa_disp	Output	16	16 bit data to be displayed

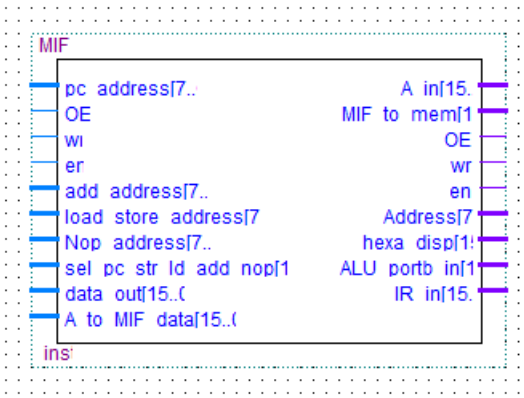

Figure 5.16: MEMORY INTERFACE BSF

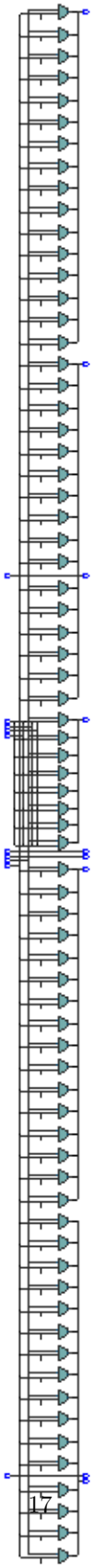

Figure 5.17: MEMORY INTERFACE RTL

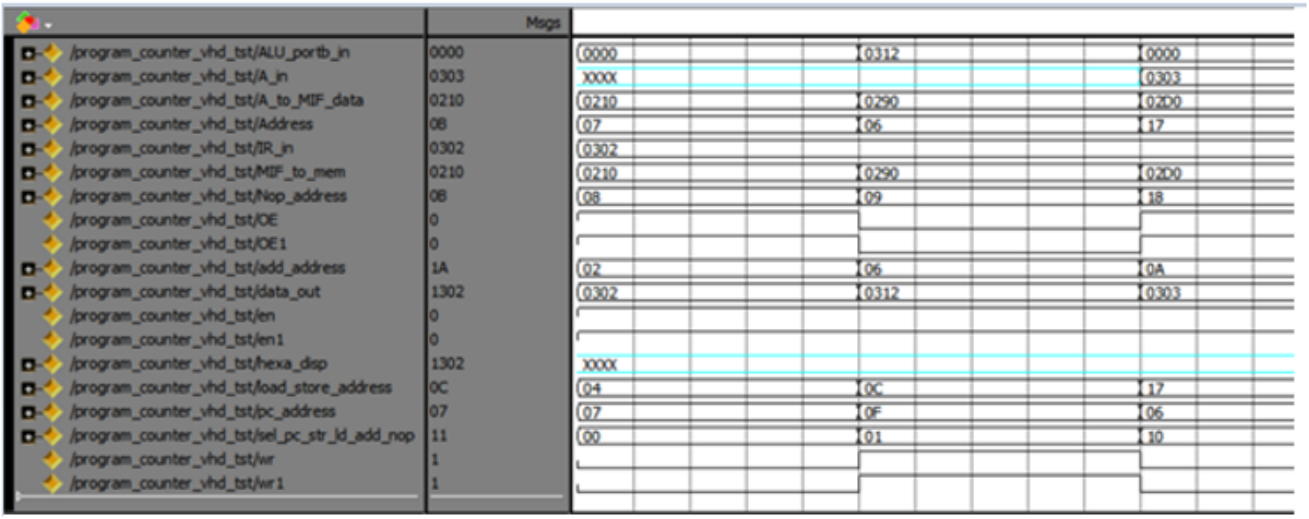

Figure 5.18: MEMORY INTERFACE SIMULATION

This component connects the processor and the external ram device. It acts like a bus, where signals are carried in and out. This consists of Multiplexer and Demultiplexer. The address is sent to the memory interface from the units from the processor to ram and the data is fetched from that address and sent to the respective units in the processor.

5.7 CONTROL UNIT

CONTROL UNIT			
External signals	Port	Width (In bits)	Description
clk	Input	1	Clock signal (12.5MHz)
reset	Input	1	Global Reset signal from the press button
instr_reg	Input	16	16 - bit instruction from the IR register
zero_flag	Input	1	Zero flag
overflow_flag	Input	1	Overflow flag
re	Output	1	Read signal
wr	Output	1	Write signal
select_line_of_pc	Output	2	Program counter select line
sel_pc_str_ld_add_nop	Output	2	Select line to select load, store,add,nop or program counter address
en	Output	1	Enable signal to enable the ram
add_address1	Output	8	8- bit add address
load_store_address1	Output	8	8-bit load address
Nop_address1	Output	8	8 bit no operation address
IR_enable1	Output	1	IR enable signal
A_enable1	Output	1	A register enable signal
ALU_or_ram_to_a	Output	1	Select line of A register
op_select_ALU	Output	3	Select line to choose arithmetic operation
jump_add_pc	Output	8	8 bit address sent to program counter

The control unit is the brain of the processor. It controls the bus signals and the components. This consists of 17 states which are idle, fetch1, fetch2, decode1, decode2, load1,load2,load3,load4,add1,add2, store1,store2, store3,jneg,jump and nop states.

After reset, It starts from the idle state then it enters the fetch state. In fetch stage, the instruction stored in the ram is fetched, from ram it is passed to IR register, then it is sent to the control unit .

At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode and 8-bit address. According to the opcode load,add,store,jump,jneg and nop is performed.

At load stage, 8-bit address extracted from the 16-bit instruction is sent to the MIF and the data from that address is extracted and stored in the accumulator register.

At add stage, 8-bit address extracted from the 16-bit instruction is sent to the MIF and the data from that address is extracted and sent to the ALU port B and data stored in the accumulator is sent to the ALU port A, then the add operation is performed and after the computation the result is stored in the accumulator.

At store stage, the data from the accumulator is sent to MIF and that data is stored in the ram at the 8-bit address extracted from the 16-bit instruction.

At Jump stage, 8-bit address extracted from the 16-bit instruction is sent to the Program counter, then it jumps to the that specific address.

At Jneg stage, first it checks for negative value in the accumulator. If so, then 8-bit address extracted from the 16-bit instruction is sent to the Program counter, then it jumps to the that specific address.

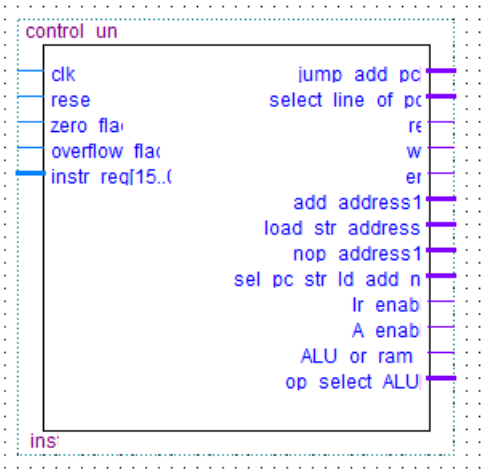

Figure 5.19: CONTROL UNIT BSF

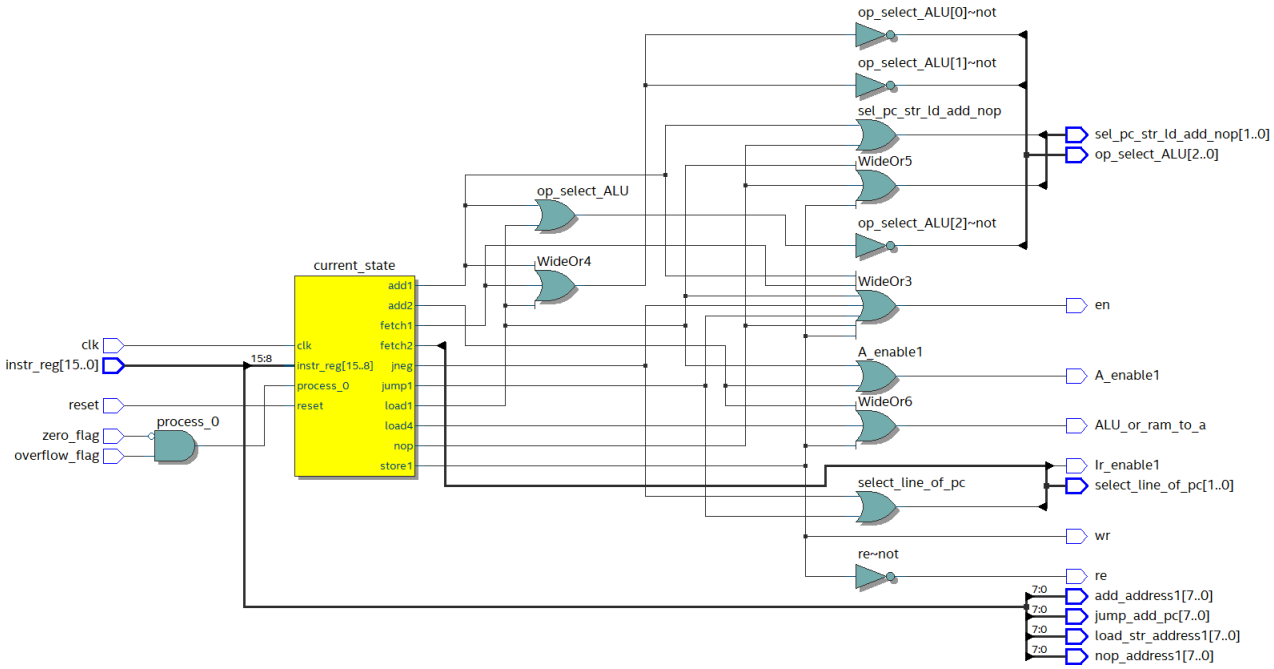

Figure 5.20: CONTROL UNIT RTL

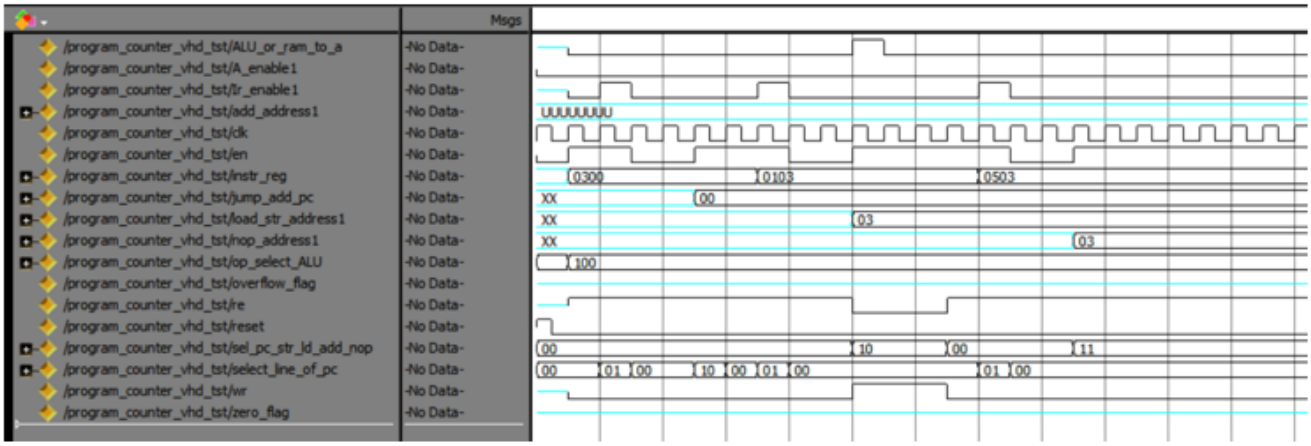

Figure 5.21: CONTROL UNIT SIMULATION

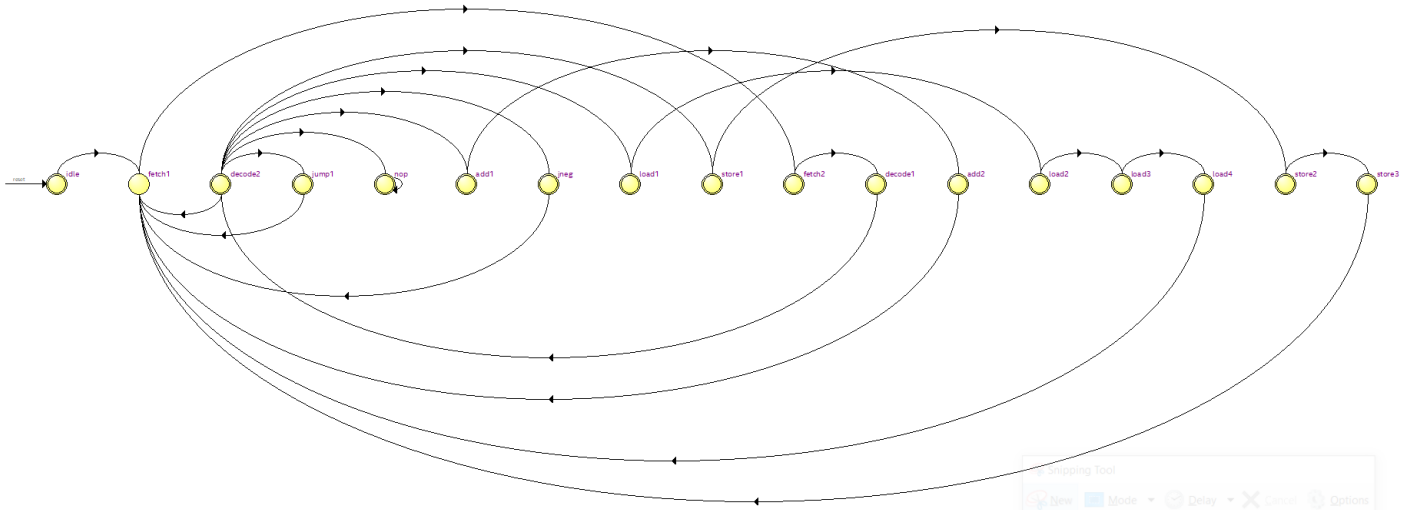

Figure 5.22: State machine simulation

Control Signal	reset	idle	fetch1	fetch2	decode1	decode2	load1	load2	load3	load4	add1	add2	store1	store2	store3	jneg	jump	nop
re	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1
wr	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
select_line_of_pc	00	00	00	01	00	00	00	00	00	00	00	00	00	00	00	10	10	00
sel_pc_str_ld_add_nop	00	00	00	00	00	00	10	10	10	10	01	01	10	10	10	00	00	11
en	0	0	1	1	0	0	1	1	1	1	1	0	1	1	1	1	1	1
Ir_enable1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A_enable1	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	0	0	0
ALU_or_ram_to_a	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0
op_select_ALU	111	111	100	100	100	100	000	000	000	000	000	000	100	100	100	100	100	100

Figure 5.23: Instruction table

At Nop stage, 8-bit address extracted from the 16-bit instruction is sent to the MIF, then it jumps to the that specific address and it stays there unless reset is pressed.

5.8 ROM (for display)

ROM			
External signals	Port	Width (In bits)	Description
addr	Input	4	4-bit data
q	Output	7	7 bit segment data

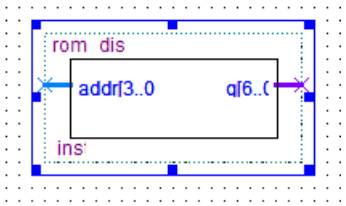

Figure 5.24: ROM BSF

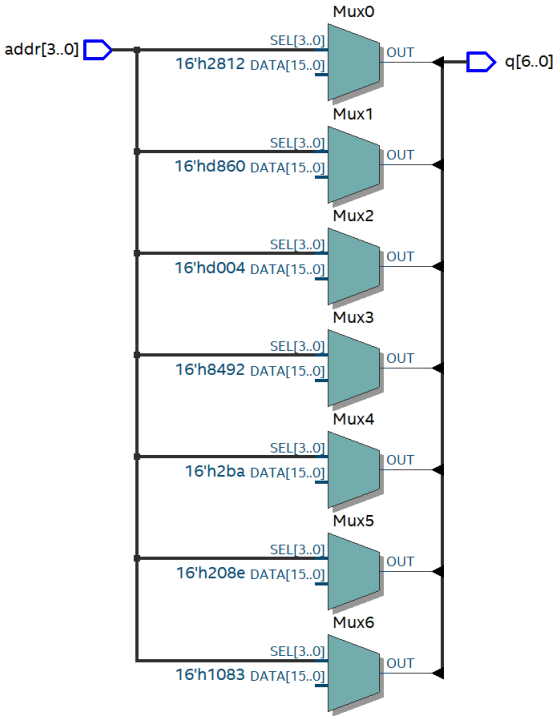

Figure 5.25: ROM RTL

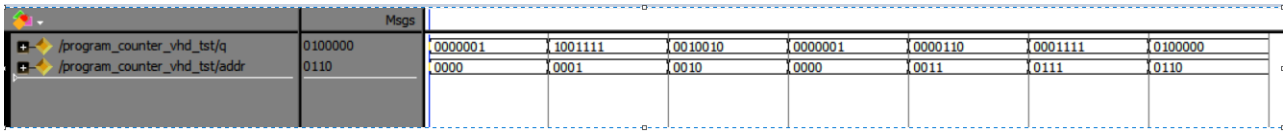

Figure 5.26: ROM SIMULATION

The data stored in the rom is permanent, it can't be changed. This component takes a signal as a std_logic_vector(3 downto 0) as input and returns with a size of 7 bits where each bit represents a segment. The value '0' shows that a segment will be on and '1' shows that a segment is off. Output values between 10 and 15 is shown as A-F

5.9 CLOCK DIVIDER

ROM			
External signals	Port	Width (In bits)	Description
clk	Input	1	External clk of 50 MHz
reset	Input	1	Reset signal
clk_out_div_4	Output	1	Output clock of 12.5 MHz for 16 bit microprocessor and ram

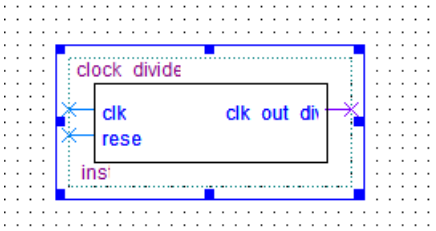

Figure 5.27: CLOCK DIVIDER BSF

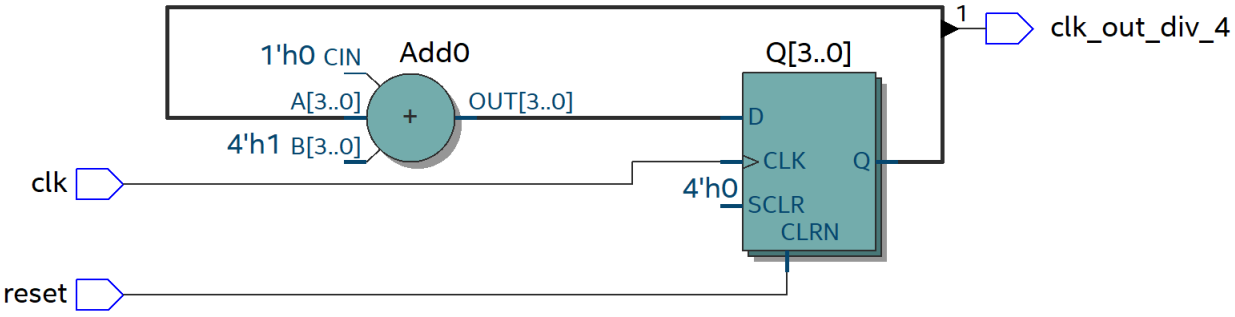

Figure 5.28: CLOCK DIVIDER RTL

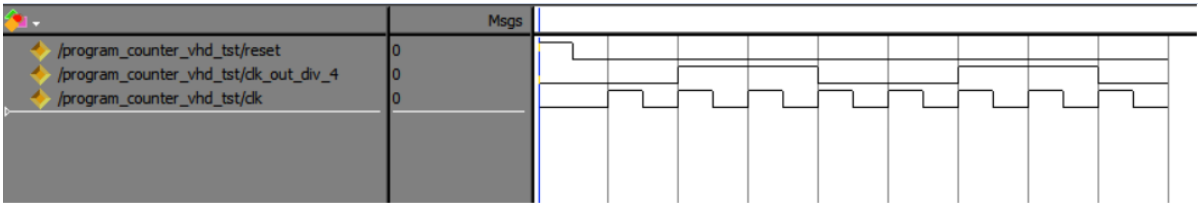

Figure 5.29: CLOCK DIVIDER SIMULATION

The clock divider is also called a frequency divider. The external clock of de1-soc is 50 MHZ. The clock divider is used to reduce the frequency by the 1/4th of the external clock frequecy which is 12.5 Mhz.

5.10 MICRO-PROCESSOR DESIGN BDF

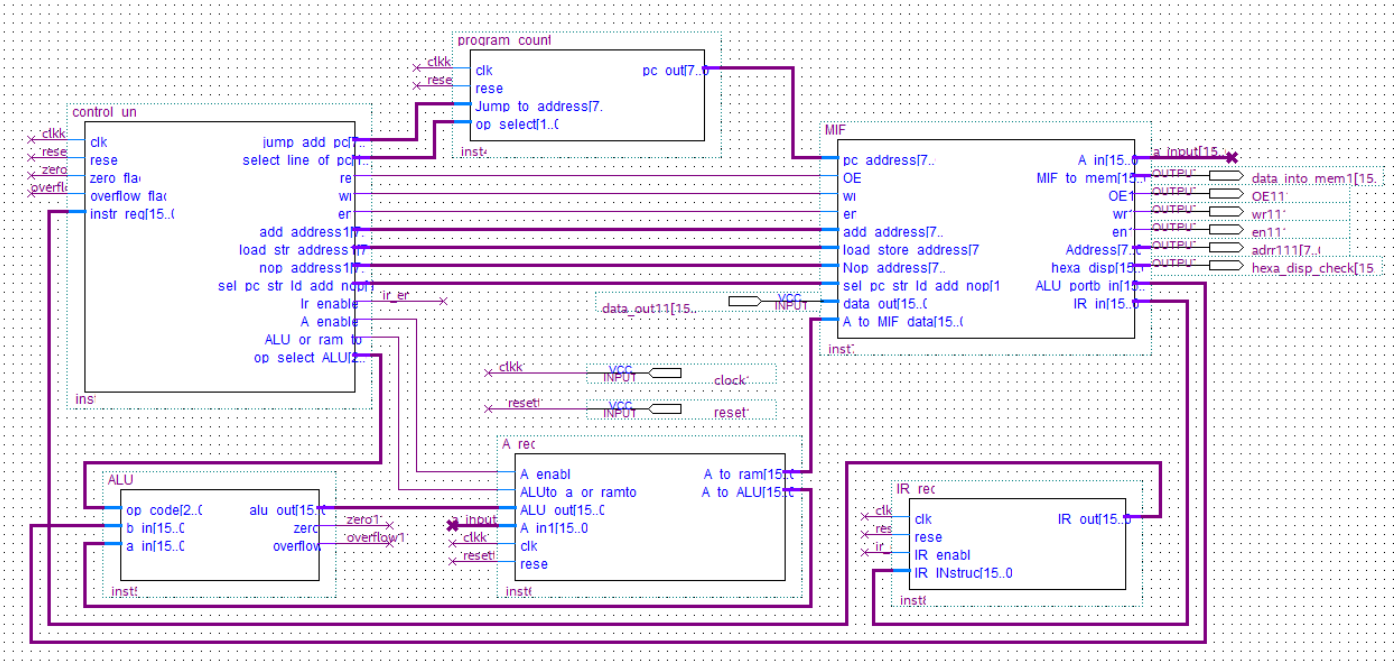

Figure 5.30: MICRO-PROCESSOR DESIGN BDF

5.11 COMPLETE DESIGN RTL

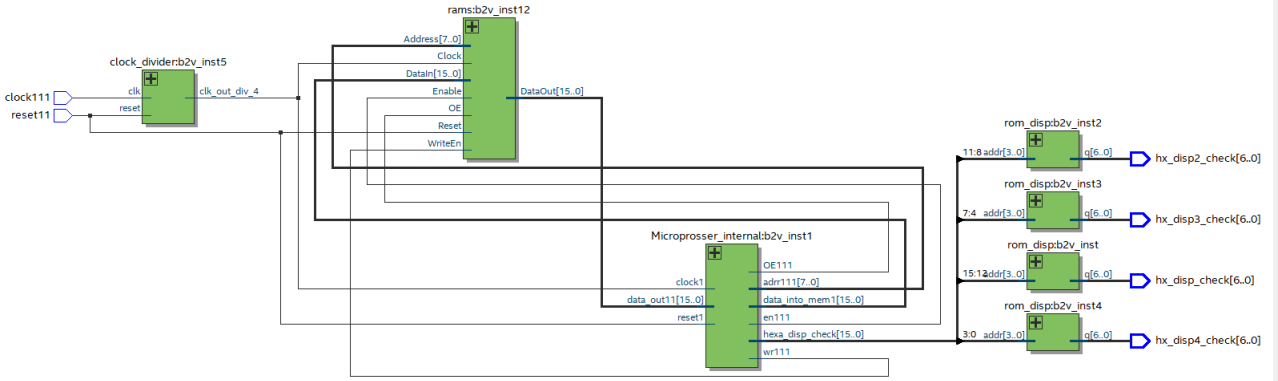

Figure 5.31: FULL DESIGN RTL

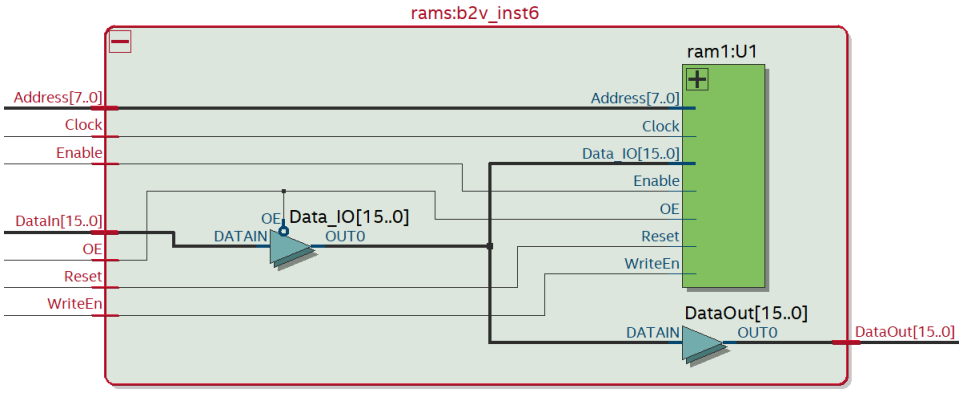

Figure 5.32: INTERNAL RAM RTL DESIGN

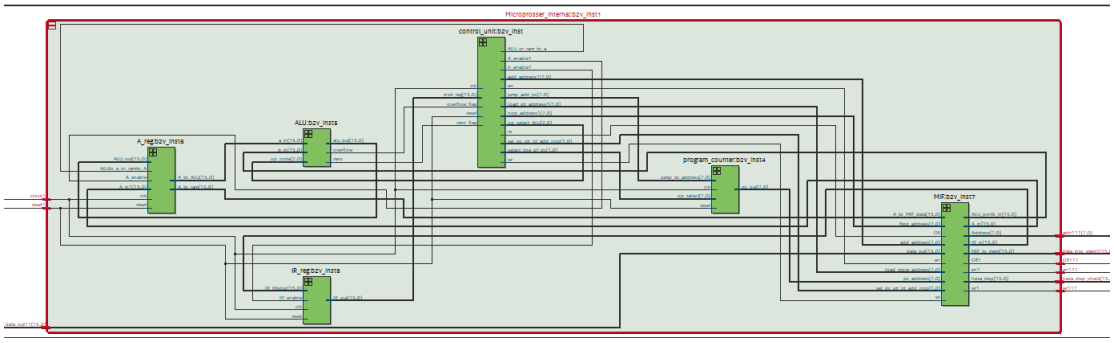

Figure 5.33: INTERNAL MICROPROCESSOR RTL DESIGN

5.12 COMPLETE DESIGN BDF

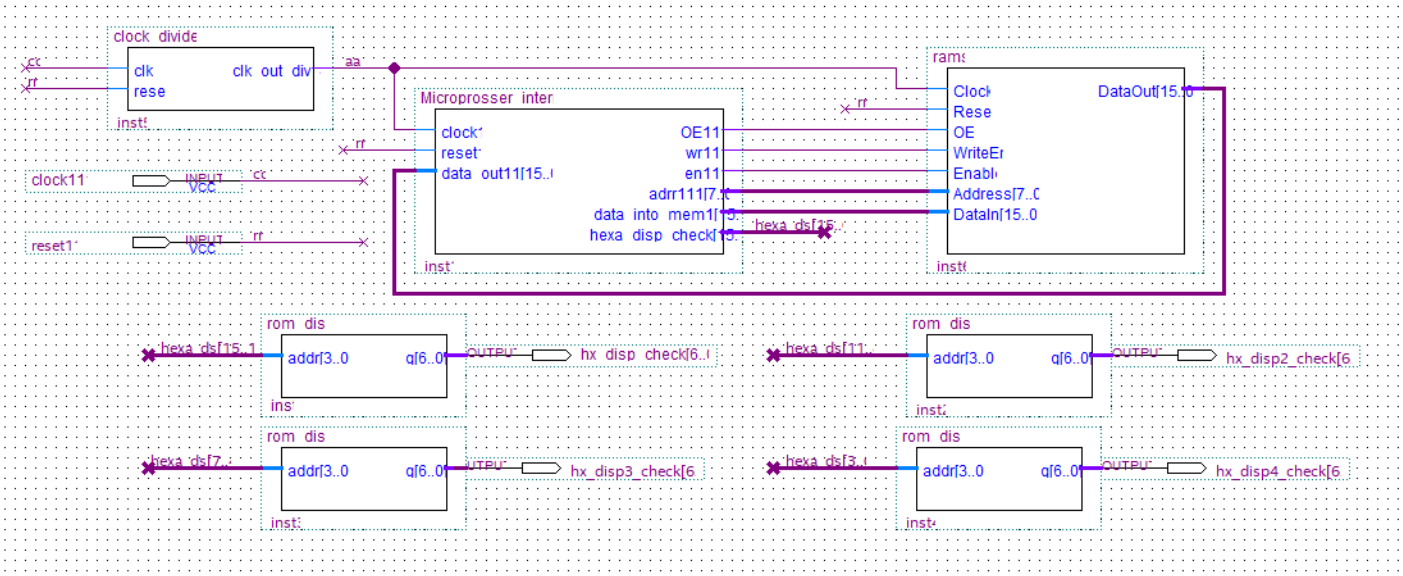

Figure 5.34: FULL DESIGN BDF

6. TEST RESULT

Hexa display check1 is the output of 16-bit for software simulation.

The 16-bit output from the Hexa display check1 port is divided into 4 parts of 4-bits and each 4-bits are fed to the rom. This rom contains the hexa decimal display informations.

hx disp check (most significant), hx disp2 check, hx disp3 check, hx disp4 (least significant) check are the final output of 7 - bits, which is sent from rom and fed to the fpga display.

6.1 SOFTWARE COMPLETE DESIGN SIMULATION TEST

6.1.1 A=B+C Execution

The figure 6.1 shows the execution of A=B+C. If the reset is pressed the instruction and data is loaded into the ram. The address 0,1,2,3(in decimal) contains the instruction to be executed and address 6,7,8 contains the data.

1. memory(0)<="0000001000000110" (0x0206); –load [address(6)]
    - First, the control units starts from the idle state after the reset is pressed. Then it moves to the fetch stage.
    - At fetch stage, the information '0x0206' stored in the address '0x00' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x01'.
    - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x02) and 8-bit address (0x06). According to the opcode "0x02", the load operation is performed.
    - At load stage, the data '0x0002' stored in the address "0x06" is extracted and it is stored in the accumulator. After the load stage is completed, it enters fetch stage.
2. memory(1)<="0000000000000111" (0x0007); –add [address(7)]
    - At fetch stage, the information '0x0007' stored in the address '0x01' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x02'.
    - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x00) and 8-bit address (0x07). According to the opcode "0x00", the add operation is performed.

- At add stage, the data '0x0005' stored in the address "0x07" is extracted and sent to the ALU port B, then the data '0x0002' stored in the accumulator is sent to the ALU port A and then control unit signals ALU to perform add operation and the resulted data '0x0007' is stored in the accumulator. After the add stage is completed, it enters fetch stage again.

3. memory(2)<="0000000100001000" (0x0108); –store [address(8)]

   - At fetch stage, the information '0x0108' stored in the address '0x02' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x03'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x01) and 8-bit address (0x08). According to the opcode "0x01", the store operation is performed.
   - At store stage, the data '0x0007' stored in the accumulator is sent to the ram and it is stored at the ram address '0x08'. After the store stage is completed, it enters fetch stage again.

4. memory(3)<="0000010100001000" (0x0508); –nop [address(8)]

   - At fetch stage, the information '0x0508' stored in the address '0x03' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x04'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x05) and 8-bit address (0x08). According to the opcode "0x05", the nop operation is performed.
   - At Nop stage, 8-bit address '0x08' extracted from the 16-bit instruction is sent to the MIF, then it jumps to the that specific address and now the result '0x0007' is displayed. It stays at that address until the reset is pressed.

```
process (Clock,reset)
begin
   if Reset='1' then
     memory(0)<="0000001000000110";  --load [address(6)]
      memory(1)<="0000000000000111";  --add [address(7)
      memory(2)<="0000000100001000";  --store [address(8)]
      memory(3)<="0000010100001000";  --nop [address(8)]
      memory(6)<=std_logic_vector(to_signed(2,16)); --addr 6 data =2
      memory(7)<=std_logic_vector(to_signed(5,16)); --addr 7 data =5
      memory(8)<=std_logic_vector(to_signed(10,16));
```

Figure 6.1: A=B+C CODE

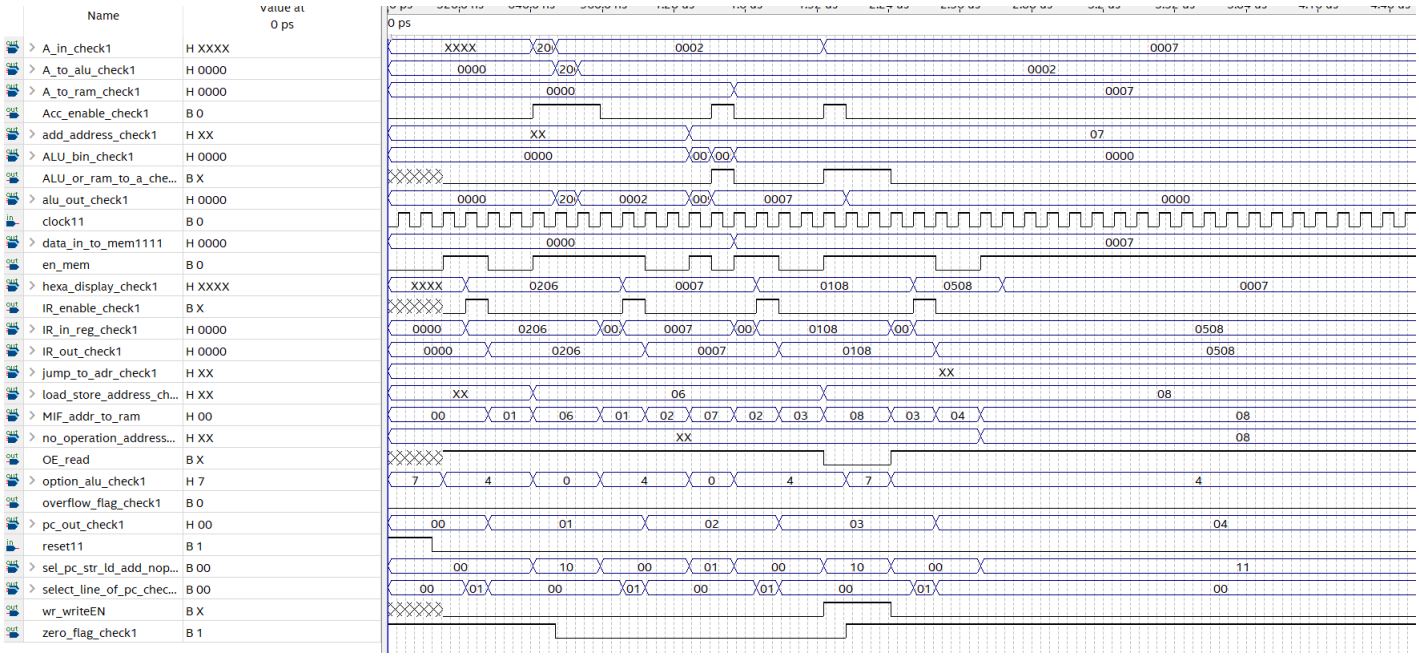

Figure 6.2: A=B+C EXECUTION

6.1.2 IF AC>0 THEN B=C Execution

The figure 6.3 shows the execution of IF AC>0 THEN B=C. If the reset is pressed the instruction and data is loaded into the ram. The address 0,1,2,3,4,5(in decimal) contains the instruction to be executed and address 6,7,8 contains the data.

1. memory(0)<="0000001000000110" (0x0206); –load [address(6)]

   - First, the control units starts from the idle state after the reset is pressed. Then it moves to the fetch stage.
   - At fetch stage, the information '0x0206' stored in the address '0x00' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x01'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x02) and 8-bit address (0x06). According to the opcode "0x02", the load operation is performed.
   - At load stage, the data '0x0009' stored in the address "0x06" is extracted and it is stored in the accumulator. After the load stage is completed, it enters fetch stage.

2. memory(1)<="0000000000000111" (0x0007); –add [address(7)]

   - At fetch stage, the information '0x0007' stored in the address '0x01' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x02'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x00) and 8-bit address (0x07). According to the opcode "0x00", the add operation is performed.

- At add stage, the data '0x0003' stored in the address "0x07" is extracted and sent to the ALU port B, then the data '0x0009' stored in the accumulator is sent to the ALU port A and then control unit signals ALU to perform add operation and the resulted data '0x000C' is stored in the accumulator. After the add stage is completed, it enters fetch stage again.

3. memory(2)<="0000010000001000" (0x0408); –jneg [address(8)]

   - At fetch stage, the information '0x0408' stored in the address '0x02' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x03'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x04) and 8-bit address (0x08). Before entering into the jneg state, the control unit will check whether the accumulator is having a negative value. If negative value is stored then it goes to the jneg stage else it moves to the fetch stage. In this case, accumulator is holding positive data '0x000C'. So it moves to the fetch stage.

4. memory(3)<="0000001000000110" (0x0206); –load [address(6)]

   - At fetch stage, the information '0x0206' stored in the address '0x03' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x04'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x02) and 8-bit address (0x06). According to the opcode "0x02", the load operation is performed.
   - At load stage, the data '0x0009' stored in the address "0x06" is extracted and it is stored in the accumulator. After the load stage is completed, it enters fetch stage.

5. memory(4)<="0000000100000111" (0x0107); –store [address(7)]

   - At fetch stage, the information '0x0107' stored in the address '0x04' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x05'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x01) and 8-bit address (0x07). According to the opcode "0x01", the store operation is performed.
   - At store stage, the data '0x0009' stored in the accumulator is sent to the ram and it is stored at the ram address '0x07'. After the store stage is completed, it enters fetch stage again.

6. memory(5)<="0000010100000111" (0x0507); –nop [address(7)]

   - At fetch stage, the information '0x0507' stored in the address '0x05' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x06'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x05) and 8-bit address (0x07). According to the opcode "0x05", the nop operation is performed.
   - At Nop stage, 8-bit address '0x07' extracted from the 16-bit instruction is sent to the MIF, then it jumps to the that specific address and now the result '0x0009' is displayed. It stays at that address until the reset is pressed.

```
process (Clock,reset)
begin
   if Reset='1' then
     memory(0)<="0000001000000110";  --load [address(6)]
      memory(1)<="0000000000000111";  --add [address(7)]
      memory(2)<="0000010000001000";  --jneg [address(8)]
      memory(3)<="0000001000000110";  --load [address(6)]
      memory(4)<="0000000100000111";  --store [address(7)]
      memory(5)<="0000010100000111";  --nop [address(7)]
      memory(6)<=std_logic_vector(to_signed(9,16)); --addr 5 data =9
      memory(7)<=std_logic_vector(to_signed(3,16)); --addr 6 data =3
      memory(8)<=std_logic_vector(to_signed(10,16));
```

Figure 6.3: IF AC>=0 THEN B=C CODE

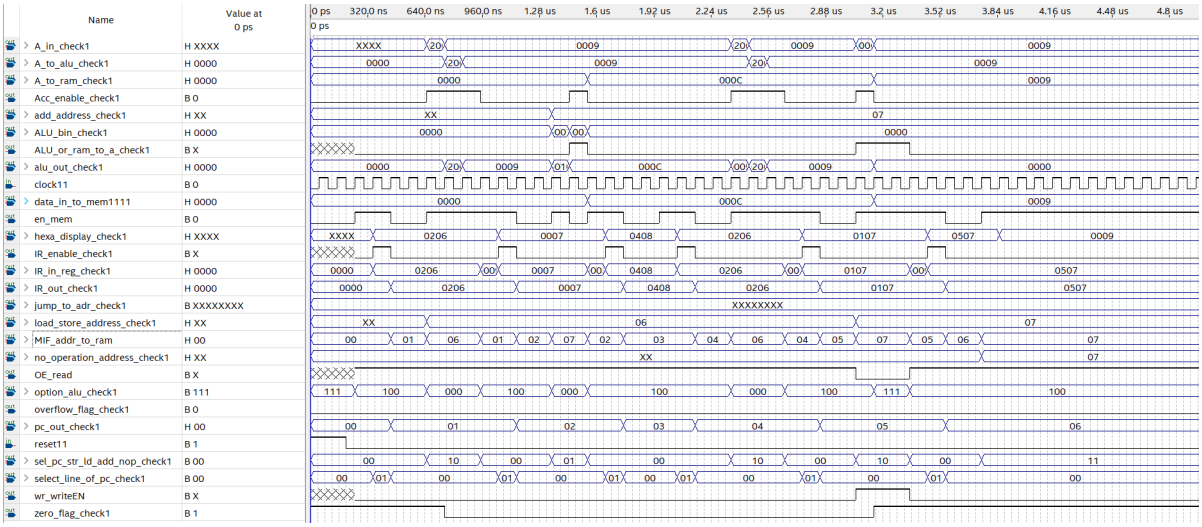

Figure 6.4: IF AC>=0 THEN B=C EXECUTION

6.1.3 IF AC<0 THEN PC<=ADDRESS EXECUTION

The figure 6.3 shows the execution of IF AC<0 THEN PC<=ADDRESS. If the reset is pressed the instruction and data is loaded into the ram. The address 0,1,2,3,4,5(in decimal) contains the instruction to be executed and address 6,7,8,9 contains the data.

1. memory(0)<="0000001000000110" (0x0206); –load [address(6)]

   - First, the control units starts from the idle state after the reset is pressed. Then it moves to the fetch stage.
   - At fetch stage, the information '0x0206' stored in the address '0x00' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x01'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x02) and 8-bit address (0x06). According to the opcode "0x02", the load operation is performed.
   - At load stage, the data '-9' (in decimal) stored in the address "0x06" is extracted and it is stored in the accumulator. After the load stage is completed, it enters fetch stage.

2. memory(1)<="0000000000000111" (0x0007); –add [address(7)]

   - At fetch stage, the information '0x0007' stored in the address '0x01' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x02'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x00) and 8-bit address (0x07). According to the opcode "0x00", the add operation is performed.
   - At add stage, the data '0x0003' stored in the address "0x07" is extracted and sent to the ALU port B, then the data '-9' stored in the accumulator is sent to the ALU port A and then control unit signals ALU to perform add operation and the resulted data '-6' is stored in the accumulator. After the add stage is completed, it enters fetch stage again.

3. memory(2)<="0000010000001000" (0x0408); –jneg [address(8)]

   - At fetch stage, the information '0x0408' stored in the address '0x02' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x03'.
   - At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x04) and 8-bit address (0x08). Before entering into the jneg state, the control unit will check whether the accumulator is having a negative value. If negative value is stored then it goes to the jneg stage else it moves to the fetch stage. In this case, accumulator is holding negative data '-6'. So it moves to the jneg stage.
   - At jneg stage, , 8-bit address '0x08' extracted from the 16-bit instruction is sent to the Program counter from control unit, now program counter jumps directly to the address '0x08' skipping in between address. After the jneg stage is completed, it enters fetch stage again.

4. memory(8)<="0000010100001001" (0x0509); –nop [address(9)]

   - Initially program counter is at address '0x08'. At fetch stage, the information '0x0509' stored in the address '0x08' in the ram is extracted and send to the instruction register and then to the control unit and program counter is incremented by '1'. So, next address becomes '0x09'.

- At decode stage, 16 bit instruction from the IR register is decoded , where the 16 bit instruction is divided into 8-bit opcode (0x05) and 8-bit address (0x09). According to the opcode "0x05", the nop operation is performed.
- At Nop stage, 8-bit address '0x09' extracted from the 16-bit instruction is sent to the MIF, then it jumps to the that specific address and now the result 0x000A is displayed. It stays at that address until the reset is pressed.

```
process (Clock,reset)
begin
   if Reset='1' then
     memory(0)<="0000001000000110";  --load [address(6)]
      memory(1)<="0000000000000111";  --add [address(7)]
      memory(2)<="0000010000001000";  --jneg [address(8)]
      memory(3)<="0000001000000110";  --load [address(6)]
      memory(4)<="0000000100000111";  --store [address(7)]
      memory(5)<="0000010100000111";  --nop [address(7)]
      memory(6)<=std_logic_vector(to_signed(-9,16)); --addr 6 data =-9
      memory(7)<=std_logic_vector(to_signed(3,16)); --addr 7 data =3
      memory(8)<="0000010100001001"; ----nop [address(9)]
      memory(9)<=std_logic_vector(to_signed(10,16));
```

Figure 6.5: IF AC<0 THEN PC<=ADDRESS (WHEN A IS A NEGATIVE NUMBER) CODE

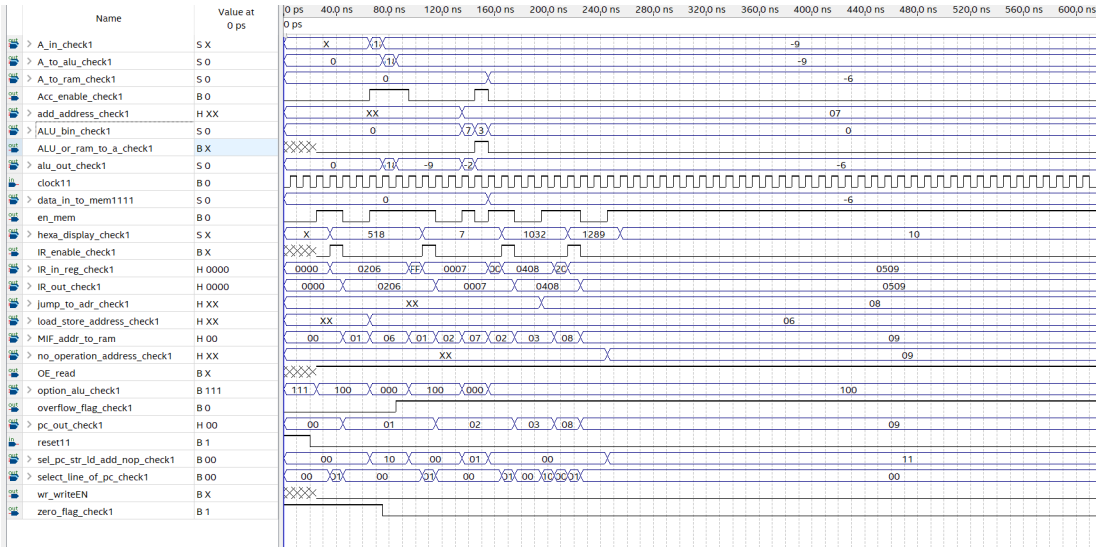

Figure 6.6: IF AC<0 THEN PC<=ADDRESS (WHEN A IS A NEGATIVE NUMBER) EXECUTION

6.2 HARDWARE COMPLETE DESIGN SIMULATION TEST

```
process (Clock,reset)
begin
   if Reset='1' then
     memory(0)<="0000001000000110";  --load [address(6)]
      memory(1)<="0000000000000111";  --add [address(7)
      memory(2)<="0000000100001000";  --store [address(8)]
      memory(3)<="0000010100001000";  --nop [address(8)]
      memory(6)<=std_logic_vector(to_signed(2,16)); --addr 6 data =2
      memory(7)<=std_logic_vector(to_signed(5,16)); --addr 7 data =5
      memory(8)<=std_logic_vector(to_signed(10,16));
```

Figure 6.7: A=B+C CODE

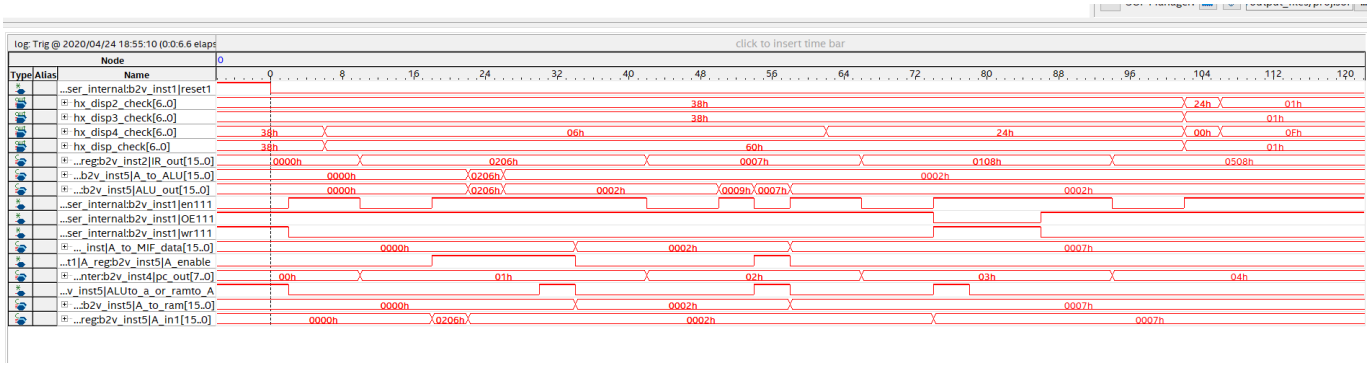

Figure 6.8: A=B+C EXECUTION ZOOM IN

While performing hardware test I added the 4 rom for hexadecimal display and a clock divider. Have set the sampling rate to 4k. By using clock divider I have reduced a the clock frequency to 12.5 MHZ, which is 1/4th of the system clock frequency (50 MHZ).

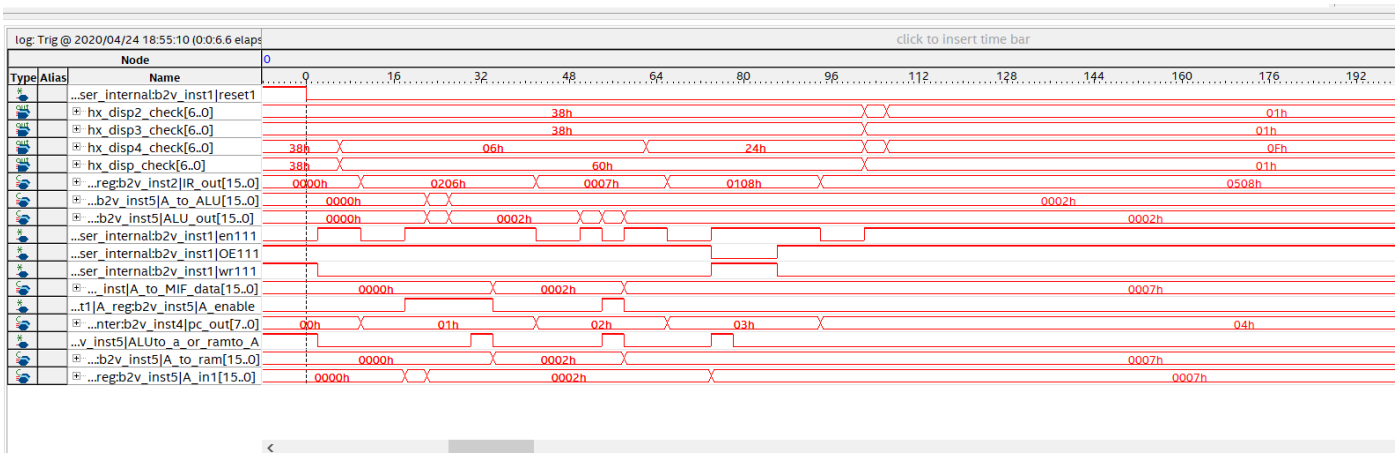

Figure 6.9: A=B+C EXECUTION ZOOM OUT

```
process (Clock,Reset)
begin
   if Reset='1' then
      memory(0)<="0000001000000110";  --load [address(6)]
      memory(1)<="0000000000000111";  --add [address(7)]
      memory(2)<="0000010000001000";  --jneg [address(8)]
      memory(3)<="0000001000000110";  --load [address(6)]
      memory(4)<="0000000100000111";  --store [address(7)]
      memory(5)<="0000010100000111";  --nop [address(7)]
      memory(6)<=std_logic_vector(to_signed(9,16)); --addr 6 data =9
      memory(7)<=std_logic_vector(to_signed(3,16)); --addr 7 data =3
      memory(8)<=std_logic_vector(to_signed(10,16));
```

Figure 6.10: IF AC>=0 THEN B=C CODE

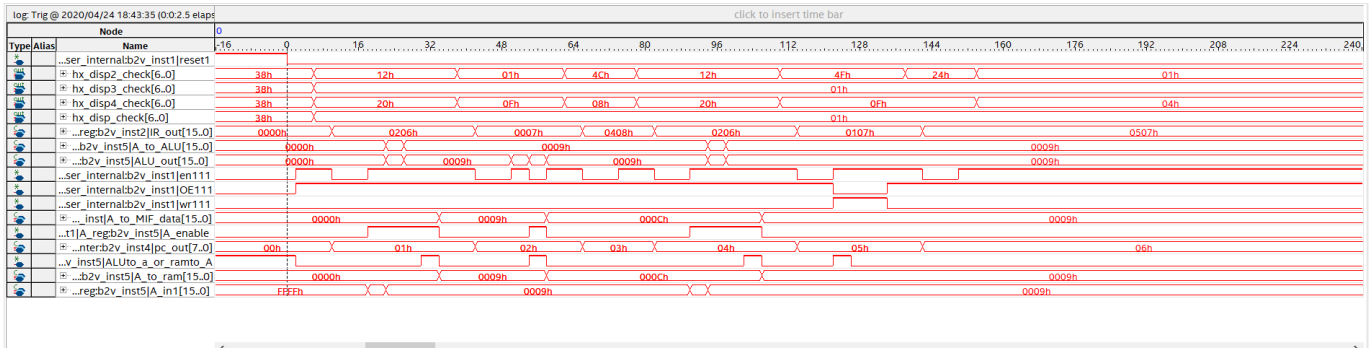

Figure 6.11: IF AC>=0 THEN B=C EXECUTION ZOOM IN

Figure 6.12: IF AC>=0 THEN B=C EXECUTION ZOOM OUT

```
process (Clock,reset)
begin
   if Reset='1' then
     memory(0)<="0000001000000110";  --load [address(6)]
      memory(1)<="0000000000000111";  --add [address(7)]
      memory(2)<="0000010000001000";  --jneg [address(8)]
      memory(3)<="0000001000000110";  --load [address(6)]
      memory(4)<="0000000100000111";  --store [address(7)]
      memory(5)<="0000010100000111";  --nop [address(7)]
      memory(6)<=std_logic_vector(to_signed(-9,16)); --addr 6 data =-9
      memory(7)<=std_logic_vector(to_signed(3,16)); --addr 7 data =3
      memory(8)<="0000010100001001"; ----nop [address(9)]
      memory(9)<=std_logic_vector(to_signed(10,16));
```

Figure 6.13: IF AC<0 THEN PC<=ADDRESS (WHEN A IS A NEGATIVE NUMBER) CODE

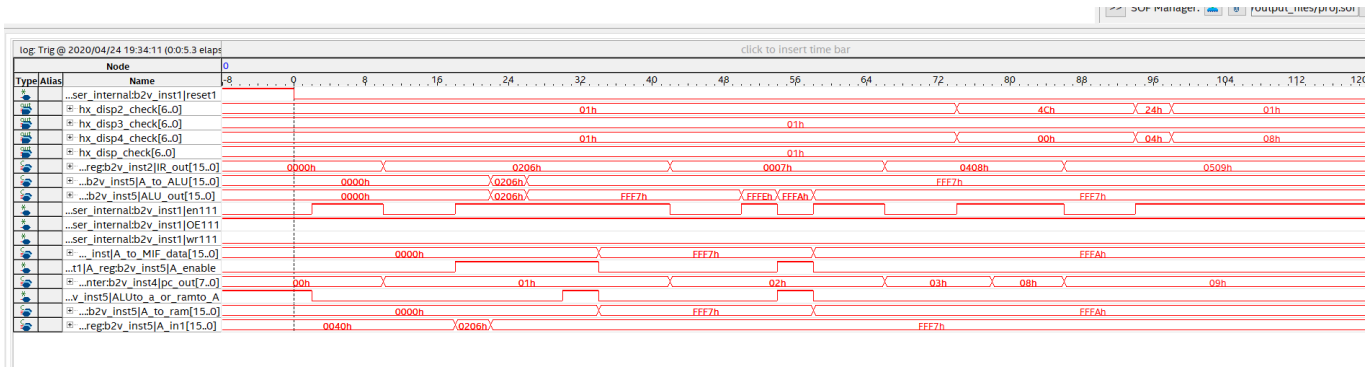

Figure 6.14: IF AC<0 THEN PC<=ADDRESS (WHEN A IS A NEGATIVE NUMBER) EXECUTION ZOOM IN

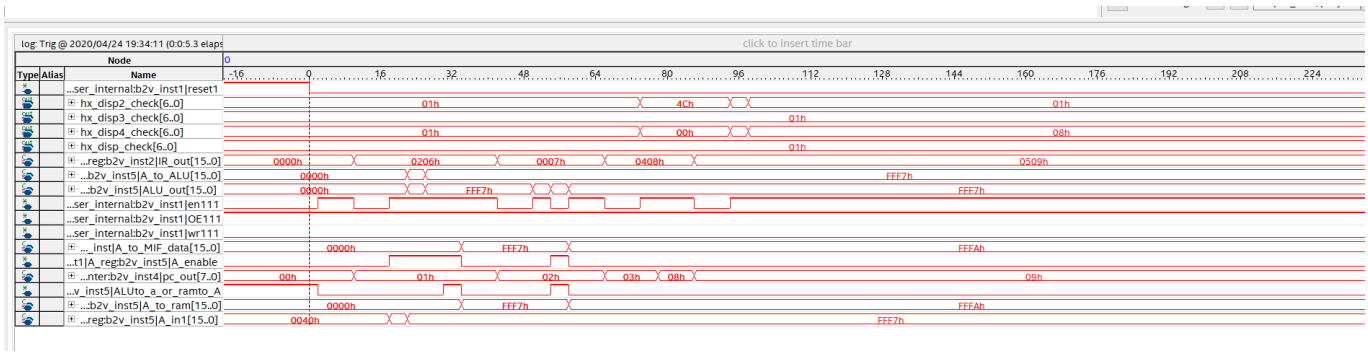

Figure 6.15: IF AC<0 THEN PC<=ADDRESS (WHEN A IS A NEGATIVE NUMBER) EXECUTION ZOOM OUT

7. PROJECT DEMO LINKS

A=B+C: =>CLICK ME TO WATCH THE VIDEO.

IF AC>=0 THEN B=C (Ac greater than zero): =>CLICK ME TO WATCH THE VIDEO.

IF AC<0 THEN PC<= ADDRESS(Ac less than zero):=>CLICK ME TO WATCH THE VIDEO.

8. CONCLUSION

The instruction stored in the ram is successfully executed in 16 Bit Microprocessor. The behavior of 16 bit Microprocessor is shown in functional simulation and signal tap simulation. I have implemented using 2 cycles of fetch, 4 cycles of load, 2 cycles of add, 3 cycles of store, 1 cycle jump, 1 cycle jneg and 1 cycle NOP (no operation) but in future we could reduce the number of cycles to speed up the microprocessor. I have constructed memory interface using simple multiplexer and demultiplexer but we could also use wishbone interface to reduce the bus complexity.

This 16 – bit microprocessor uses only one register , this results in less memory space but as the number of instructions increases for a program, the execution time increases too, in order to decrease the execution time we could also add additional registers for better performance.

Bibliography

1) https://blog.classycode.com/implementing-a-cpu-in-vhdl-part-1-6afd4c1ed491
2)https://www.youtube.com/watch?v=XM4lGflQFvA
3)https://www.youtube.com/watch?v=jFDMZpkUWCw
4)https://surf-vhdl.com/how-to-implement-clock-divider-vhdl/
5)http://labs.domipheus.com/blog/designing-a-cpu-in-vhdl-part-6-program-counter-instruction-fetch-branching/
6) VHDL for designers stefan sjoholm

9. Appendix A: SOURCE CODE

9.1 RAM CODE

9.1.1 RAM COMPONENT CODE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
Package ram1comp is
     component ram1
          port(
                Clock      : in  STD_LOGIC;
                Address    : in  STD_LOGIC_VECTOR (7 downto 0);
                Reset      : in  STD_LOGIC;
                WriteEn    : in  STD_LOGIC;
                Enable     : in  STD_LOGIC;
                OE         : in  STD_LOGIC;
                Data_IO    : inout STD_LOGIC_VECTOR (15 downto 0)
                 );
end component ram1;
end package ram1comp;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ram1 is

    Port (
        Clock        : in  STD_LOGIC;
        Address      : in  STD_LOGIC_VECTOR (7 downto 0);
        Reset        : in  STD_LOGIC;
        WriteEn      : in  STD_LOGIC;
        Enable       : in  STD_LOGIC;
        OE           : in  STD_LOGIC;
        Data_IO      : inout STD_LOGIC_VECTOR (15 downto 0)

    );
end ram1;

architecture rtl of ram1 is
    type Ram_256 is array ( 0 to 255) of STD_LOGIC_VECTOR (15 downto 0);
    signal memory : Ram_256;
    signal DataIn,DataOut : STD_LOGIC_VECTOR (15 downto 0);
begin


    process (Clock,Reset)
    begin
       if Reset = '1' then
            memory(0)<="0000001000000110";  --load [address(6)]
            memory(1)<="0000000000000111";  --add [address(7)]
            memory(2)<="0000000100000111";  --store [address(7)]
            memory(3)<="0000010100000111";  --nop [address(7)]
            memory(6)<=std_logic_vector(to_signed(9,16)); --addr 6 data =9
            memory(7)<=std_logic_vector(to_signed(3,16)); --addr 7 data =3
            memory(8)<="0000010100001001";  --nop [address(7)]
            memory(9)<=std_logic_vector(to_signed(10,16));
```

```
53
54
55
56
57          elsif rising_edge(Clock) then
58               if Enable = '1' then
59                   if WriteEn = '1' then   memory(to_integer(unsigned(Address))) <= DataIn;
60                       DataOut <= DataIn;
61                   else
62                       DataOut <= memory(to_integer(unsigned(Address)));
63                   end if;
64               end if;
65          end if;
66      end process;
67  Data_IO  <= DataOut when (OE='1') else (others=>'Z');
68  DataIn   <= Data_IO;
69  end rtl;
```

9.1.2 MAIN RAM CODE

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4   use work.ram1comp.all;
5   entity rams is
6       port (      Clock       : in  STD_LOGIC;
7                   Reset       : in  STD_LOGIC;
8                   OE          : in  STD_LOGIC;
9                   WriteEn     : in  STD_LOGIC;
10                  Enable      : in  STD_LOGIC;
11                  Address     : in  STD_LOGIC_VECTOR (7 downto 0);
12                  DataIn      : in std_logic_vector(15 downto 0);
13                  DataOut     : out std_logic_vector(15 downto 0)
14
15
16
17
18
19
20            );
21       end entity rams;
22  architecture rtl of rams is
23  signal Data_IO : std_logic_vector(15 downto 0);
24  begin
25
26          U1          :ram1 port map (Clock,Address,Reset,WriteEn,Enable,OE,Data_IO);
27          Data_IO     <= DataIn  when (OE='0') else (others=>'Z');
28          DataOut     <= Data_IO;
29
30
31
32
33
34  end rtl;
```

9.2 ACCUMULATOR CODE

```
1   LIBRARY IEEE;
2   USE IEEE.STD_LOGIC_1164.ALL;
3   USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5   entity A_reg is
6       Port (
7               A_enable                :in std_logic;
8               ALUto_a_or_ramto_A      :in STD_LOGIC;
```

```
            ALU_out               :in STD_LOGIC_VECTOR (15 downto 0);
            A_in1                 :in  STD_LOGIC_VECTOR (15 downto 0);
            clk,reset:in STD_LOGIC;
            A_to_ram              :out  STD_LOGIC_VECTOR (15 downto 0);
            A_to_ALU              :out  STD_LOGIC_VECTOR (15 downto 0)
          );
end A_reg;

architecture rtl of A_reg is
begin
 process ( clk,reset)
 begin
        if reset = '1' then
              A_to_ram <= (others=>'0');
              A_to_ALU <= (others=>'0');
        elsif rising_edge(clk) then
             if A_enable = '1' then
                 case ALUto_a_or_ramto_A is
                   when  '0'     => A_to_ALU <=  A_in1;

                   when  others  => A_to_ram <=  ALU_out;
                 end case;
              end if;
        end if;
end process;
end rtl;
```

9.3 ARITHMETIC LOGIC UNIT CODE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity ALU is
     port (  op_code    :in std_logic_vector(2 downto 0);
             b_in       :in std_logic_vector(15 downto 0);
             a_in       :in std_logic_vector(15 downto 0);
             alu_out    :out std_logic_vector(15 downto 0);
             zero       :out std_logic;
             overflow   :out std_logic
                );
end ALU;

Architecture rtl of ALU is
     signal a1 : std_logic_vector(16 downto 0);
        begin
              a1 <=       ((a_in(15) & a_in) + (b_in(15) & b_in)) when op_code="000"     else
                          ((a_in(15) & a_in) - (b_in(15) & b_in)) when op_code="001"     else
                          ((a_in(15) & a_in) and (b_in(15) & b_in)) when op_code="010"   else
                          ((a_in(15) & a_in) or (b_in(15) & b_in))when op_code="011"     else
                          (a_in(15) & a_in) when op_code="100" else (others=>'0');

alu_out  <= a1(15 downto 0);
overflow <= '1' when a1(15)='1'    else '0';
zero     <= '1' when a1="00000000" else '0';
end rtl;
```

9.4 CLOCK DIVIDER CODE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

entity clock_divider is
```

```
port

(
clk                  :in std_logic;
reset                :in std_logic;
clk_out_div_4        :out std_logic

);
end entity;
architecture rtl of clock_divider is
signal Q: std_logic_vector(3 downto 0);
begin
process(clk,reset)
 begin
     if(reset = '1') then
   Q <= ( 0 => '0',OTHERS =>'0');
     elsif rising_edge(clk) then
   Q <= Q+1;
  end if;
 end process;

 clk_out_div_4 <= Q(1);



end rtl;
```

9.5 CONTROL UNIT CODE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
entity control_unit is
    port (    clk, reset               :in std_logic;
              zero_flag,overflow_flag  :in std_logic;
              instr_reg                :in std_logic_vector(15 downto 0);
              jump_add_pc              :out std_logic_vector(7 downto 0);
              select_line_of_pc        :out std_logic_vector(1 downto 0);
              re                       :out std_logic;
              wr                       :out std_logic;
              en                       :out std_logic;
              add_address1             :out std_logic_vector(7 downto 0);
              load_str_address1        :out std_logic_vector(7 downto 0);
              nop_address1             :out std_logic_vector(7 downto 0);
              sel_pc_str_ld_add_nop    :out  STD_LOGIC_vector(1 downto 0);
              Ir_enable1               :out std_logic;
              A_enable1                :out std_logic;
              ALU_or_ram_to_a          :out std_logic;
              op_select_ALU            :out std_logic_vector(2 downto 0)



            );
end control_unit;

Architecture moore of control_unit is
type state_type is (idle,fetch1,fetch2,decode1,decode2,add1,add2,store1,store2,store3,load1,load2,load3,
    load4,jump1,jneg,nop);
signal current_state, next_state: state_type;

begin
    process(current_state,zero_flag,overflow_flag,instr_reg)

     begin
     case current_state is
         when idle =>
                         next_state <= fetch1 ;

         when fetch1 =>
```

```
                        next_state <= fetch2;

        when fetch2 =>
                        next_state <= decode1;

        when decode1 =>
                        next_state <= decode2;

        when decode2 =>
               case instr_reg(15 downto 8) is
                      when "00000000" =>
                            next_state <= add1;
                      when "00000001" =>
                            next_state <= store1;
                      when "00000010" =>
                            next_state <= load1;
                      when "00000011" =>
                            next_state <= jump1;
                      when "00000100" =>

                            if overflow_flag = '1' and zero_flag = '0' then
                                next_state <= jneg;
                            else
                                next_state <= fetch1;
                            end if;
                      when "00000101" =>
                            next_state <= nop;

                      when others =>
                            next_state <= fetch1;
                end case;


        when add1 =>
                        next_state <= add2;
        when add2 =>

                        next_state <= fetch1;
        when store1 =>

                        next_state <= store2;
        when store2 =>

                        next_state <= store3;
        when store3 =>

                        next_state <= fetch1;
        when load1 =>
                        next_state <= load2;
        when load2 =>
                        next_state <= load3;

        when load3 =>
                        next_state <= load4;
        when load4 =>
                        next_state <= fetch1;

        when jump1 =>
                        next_state <= fetch1;




        when jneg =>
                        next_state <= fetch1;

        when nop =>     next_state <= nop;

        when others =>  next_state <= fetch1;


    end case;
  end process;
```

```
114
115     process ( clk,reset)
116
117     begin
118         if reset = '1' then
119               current_state <= idle;
120         elsif rising_edge(clk) then
121               current_state <= next_state;
122         end if;
123     end process;
124
125    process ( current_state,instr_reg)
126    begin
127                                 re                          <= '1';
128                                 wr                          <= '0';
129                                 select_line_of_pc           <= "00";
130                                 sel_pc_str_ld_add_nop       <= "00";
131                                 en                          <= '0';
132                                 Ir_enable1                  <= '0';
133                                 op_select_ALU               <= "111";
134                                 A_enable1                   <= '0';
135                                 ALU_or_ram_to_a             <= '0';
136                                 add_address1                <= instr_reg(7 downto 0);
137                                 load_str_address1           <= instr_reg(7 downto 0);
138                                 jump_add_pc                 <= instr_reg(7 downto 0);
139                                 nop_address1                <= instr_reg(7 downto 0);
140
141        case current_state is
142               when  idle     =>
143
144               when  fetch1   =>
145                                 re                          <= '1';
146                                 wr                          <= '0';
147                                 select_line_of_pc           <= "00";
148                                 sel_pc_str_ld_add_nop       <= "00";
149                                 en                          <= '1';
150                                 Ir_enable1                  <= '0';
151                                 op_select_ALU               <= "100";
152                                 A_enable1                   <= '0';
153                                 ALU_or_ram_to_a             <= '0';
154
155               when  fetch2   =>
156                                 select_line_of_pc           <= "01";
157                                 Ir_enable1                  <= '1';
158
159
160
161
162
163               when  decode1  =>
164
165                                 select_line_of_pc           <= "00";
166                                 en                          <= '0';
167                                 Ir_enable1                  <= '0';
168
169               when  decode2  =>
170
171
172
173               when  add1     =>
174                                 add_address1                <= instr_reg(7 downto 0);
175                                 en                          <= '1';
176                                 sel_pc_str_ld_add_nop       <= "01";
177                                 op_select_ALU               <="000";
178
179
180
181               when  add2     =>
182                                 add_address1                <= instr_reg(7 downto 0);
183                                 en                          <= '0';
184                                 A_enable1                   <= '1';
185                                 ALU_or_ram_to_a             <= '1';
186
187
```

```
188             when  store1   =>
189                           load_str_address1           <= instr_reg(7 downto 0);
190                           re                          <= '0';
191                           wr                          <= '1';
192                           en                          <= '1';
193                           sel_pc_str_ld_add_nop       <= "10";
194                           ALU_or_ram_to_a<='1';
195
196             when  store2   =>
197                           load_str_address1           <= instr_reg(7 downto 0);
198                           ALU_or_ram_to_a             <= '0';
199
200             when  store3   =>
201                           load_str_address1           <= instr_reg(7 downto 0);
202
203
204             when  load1    =>
205                           load_str_address1           <= instr_reg(7 downto 0);
206                           en                          <= '1';
207                           sel_pc_str_ld_add_nop       <= "10";
208                           op_select_ALU               <= "000";
209                           A_enable1                   <= '1';
210
211             when  load2    =>
212                           load_str_address1           <= instr_reg(7 downto 0);
213
214
215             when  load3    =>
216                           load_str_address1           <= instr_reg(7 downto 0);
217
218
219             when  load4    =>
220                           load_str_address1           <= instr_reg(7 downto 0);
221                           ALU_or_ram_to_a             <= '1';
222
223             when  jump1    =>
224                           jump_add_pc                 <= instr_reg(7 downto 0);
225                           select_line_of_pc           <= "10";
226                           en                          <= '1';
227
228
229             when  jneg     =>
230                           jump_add_pc                 <= instr_reg(7 downto 0);
231                           select_line_of_pc           <= "10";
232                           en                          <= '1';
233
234
235
236             when  nop       =>
237                           nop_address1                <= instr_reg(7 downto 0);
238                           sel_pc_str_ld_add_nop       <= "11";
239                           en                          <= '1';
240
241
242             when others     => null;
243
244       end case;
245     end process;
246     end moore;
```

9.6 INSTRUCTION REGISTER CODE

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity IR_reg is
6      Port (     clk,reset      :in STD_LOGIC;
7                 IR_enable      :in std_logic;
```

```
              IR_INstruc    :in STD_LOGIC_VECTOR (15 downto 0);
              IR_out        :out  STD_LOGIC_VECTOR (15 downto 0)
          );
end IR_reg;

architecture rtl of IR_reg is
begin
 process ( clk,reset)
 begin
        if reset = '1' then
             IR_out <= (others=>'0');

        elsif rising_edge(clk) then
             if IR_enable='1' then
                 IR_out <= IR_INstruc;
             end if;
        end if;
 end process;
end rtl;
```

9.7 MEMORY INTERFACE (MIF) CODE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MIF is
    Port (
            pc_address              :in  STD_LOGIC_VECTOR (7 downto 0);
            OE                      :in  STD_LOGIC;
            wr                      :in STD_logic;
            en                      :in std_logic;
            add_address             :in  STD_LOGIC_VECTOR (7 downto 0);
            load_store_address      :in  STD_LOGIC_VECTOR (7 downto 0);
            Nop_address             :in  STD_LOGIC_VECTOR (7 downto 0);
            sel_pc_str_ld_add_nop   :in  STD_LOGIC_vector(1 downto 0);
            data_out                :in  STD_LOGIC_VECTOR (15 downto 0);
            A_to_MIF_data           :in STD_LOGIC_VECTOR (15 downto 0);
            A_in                    :out  STD_LOGIC_VECTOR (15 downto 0);
            MIF_to_mem              :out STD_LOGIC_VECTOR (15 downto 0);
            OE1                     :out  STD_LOGIC;
            wr1                     :out STD_logic;
            en1                     :out std_logic;
            Address                 :out STD_LOGIC_VECTOR (7 downto 0);
            hexa_disp               :out STD_LOGIC_VECTOR (15 downto 0);
            ALU_portb_in            :out STD_LOGIC_VECTOR (15 downto 0);
            IR_in                   :out STD_LOGIC_VECTOR (15 downto 0)
          );
end MIF;

architecture rtl of MIF is

begin

process (pc_address,add_address,load_store_address,Nop_address,sel_pc_str_ld_add_nop) is
begin
  case sel_pc_str_ld_add_nop is

      when "00" =>
          Address <= pc_address;
      when "01" =>
          Address <= add_address;
      when "10" =>
          Address <= load_store_address;
      when others =>
          Address <= Nop_address;
  end case;

end process;
```

```
OE1         <= OE;
wr1         <= wr;
en1         <= en;
MIF_to_mem <= A_to_MIF_data;

process (data_out,sel_pc_str_ld_add_nop) is
begin
IR_in          <= x"0000";
ALU_portb_in   <= x"0000";
A_in           <= x"0000";
hexa_disp      <= x"0000";

 case sel_pc_str_ld_add_nop is
  when "00" =>
  IR_in         <= data_out;
  when "01" =>
  ALU_portb_in <= data_out;
  when "10" =>
  A_in          <= data_out;
  when others =>
  hexa_disp     <= data_out;
 end case;

end process;


end rtl;
```

9.8 PROGRAM COUNTER CODE

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

entity program_counter is
    Port (  clk              :in  STD_LOGIC;
            reset            :in  STD_LOGIC;
            Jump_to_address  :in  STD_LOGIC_VECTOR (7 downto 0);
            op_select        :in  STD_LOGIC_VECTOR (1 downto 0);
            pc_out           :out  STD_LOGIC_VECTOR (7 downto 0)
          );
end program_counter;


architecture rtl of program_counter is
  signal current_pc: std_logic_vector( 7 downto 0) := X"00";
begin

  process (clk,reset)
  begin
    if reset = '1' then
         current_pc <= X"00";
    elsif rising_edge(clk) then
      case op_select is
        when "00" =>   -- NOP, keep PC the same/halt
        when "01" =>   -- increment
          current_pc <= std_logic_vector(unsigned(current_pc) + 1);
        when "10" =>
          current_pc <= Jump_to_address;
        when "11" =>     -- Reset
          current_pc <= X"00";
        when others =>
      end case;
    end if;
  end process;

  pc_out <= current_pc;

end rtl;
```

9.9 ROM FOR SEVEN SEGMENT DISPLAY CODE

```
1  LIBRARY IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
3
4
5  package rom is
6
7    subtype  rom_word is std_logic_vector(6 downto 0);
8    type rom_table is array (0 to 15) of rom_word;
9
10   constant rom: rom_table:= rom_table'(           "0000001",
11                                                   "1001111",
12                                                   "0010010",
13                                                   "0000110",
14                                                   "1001100",
15                                                   "0100100",
16                                                   "0100000",
17                                                   "0001111",
18                                                   "0000000",
19                                                   "0000100",
20                                                   "0001000",
21                                                   "1100000",
22                                                   "0110001",
23                                                   "1000010",
24                                                   "0110000",
25                                                   "0111000");
26
27   end;
28
29 LIBRARY IEEE;
30 USE IEEE.STD_LOGIC_1164.ALL;
31 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
32 USE WORK.ROM.ALL;
33
34   Entity rom_disp is
35     port(addr      :in std_logic_vector(3 downto 0);
36          q         :out std_logic_vector(6 downto 0)
37              );
38
39       end;
40
41   architecture rtl of rom_disp is
42   begin
43     q <= rom(to_integer(unsigned(addr)));
44   end;
```