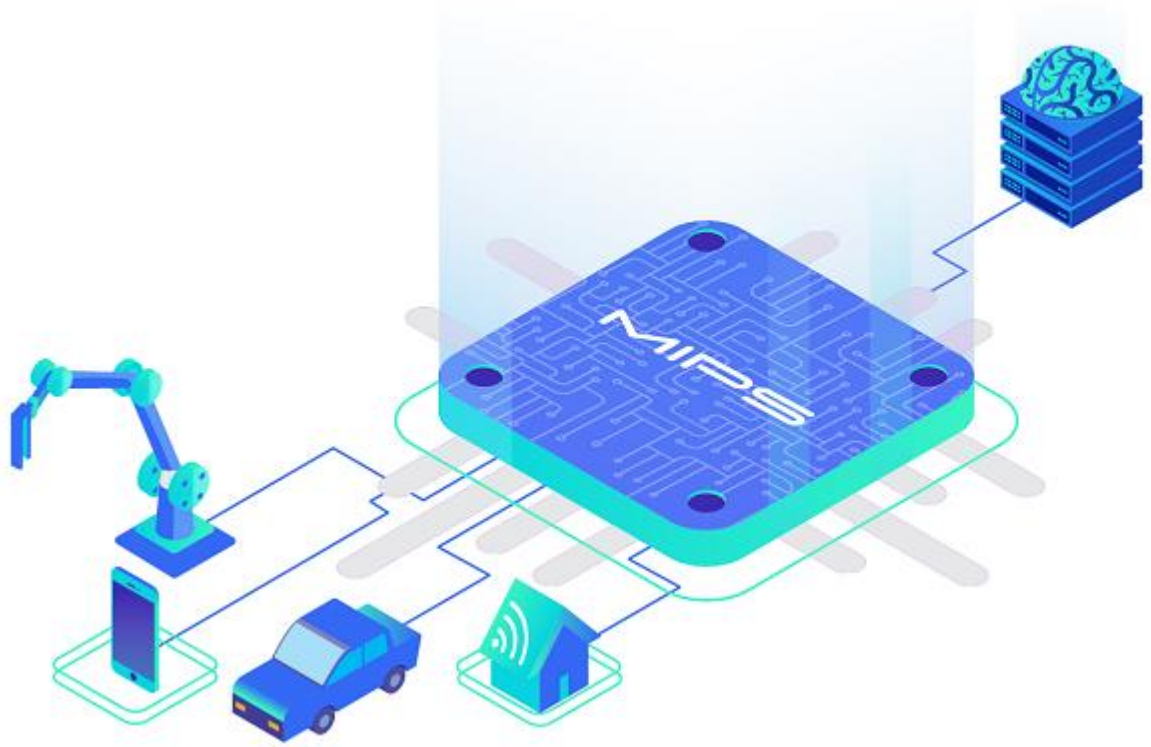
EC-340

COMPUTER ORGANIZATION AND ARCHITECTURE



UTKARSH MAHAJAN 201EC164

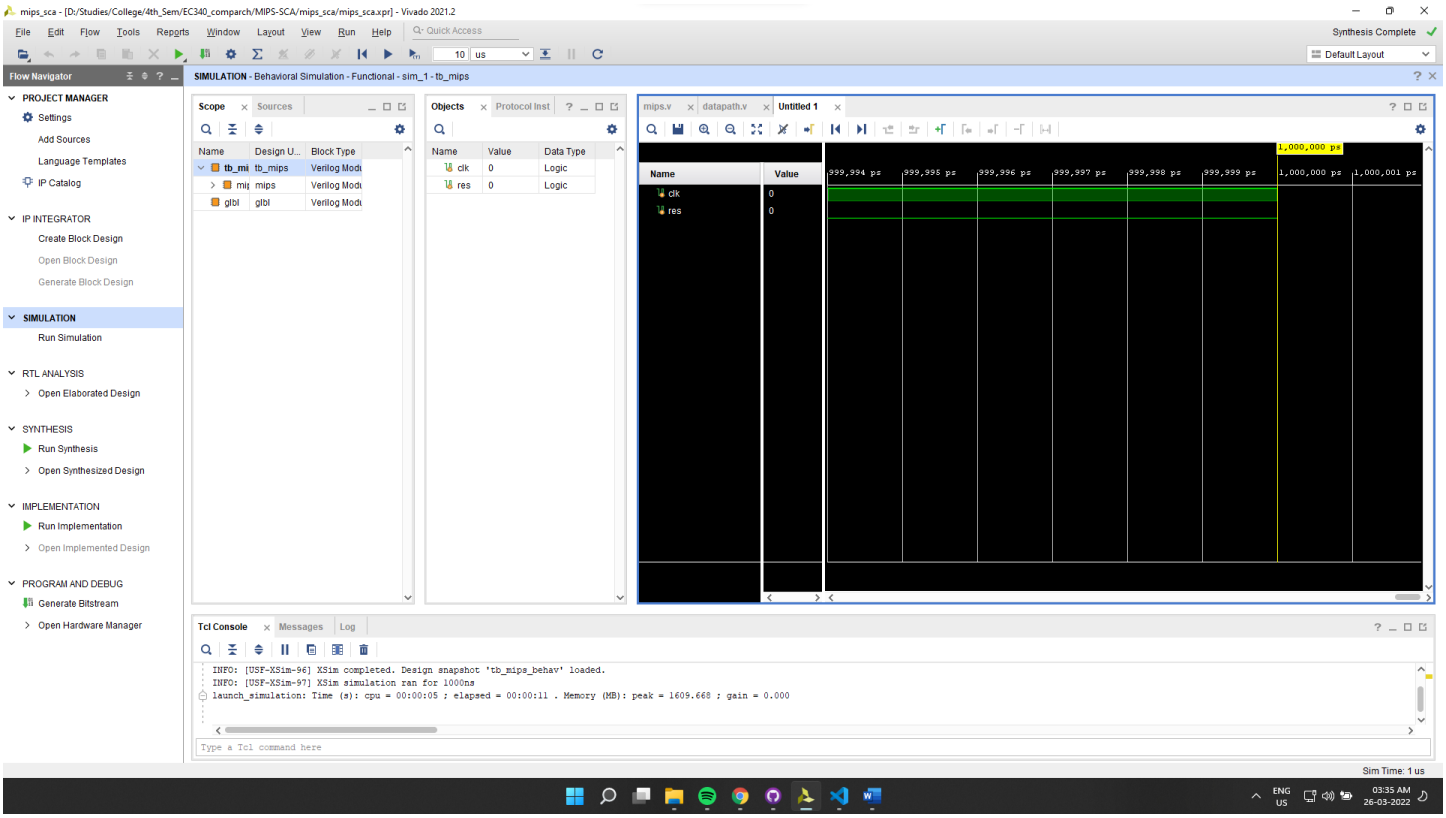
DEVVRAT ASHTAPUTRE 201EC118

R S MUTHUKUMAR 201EC149

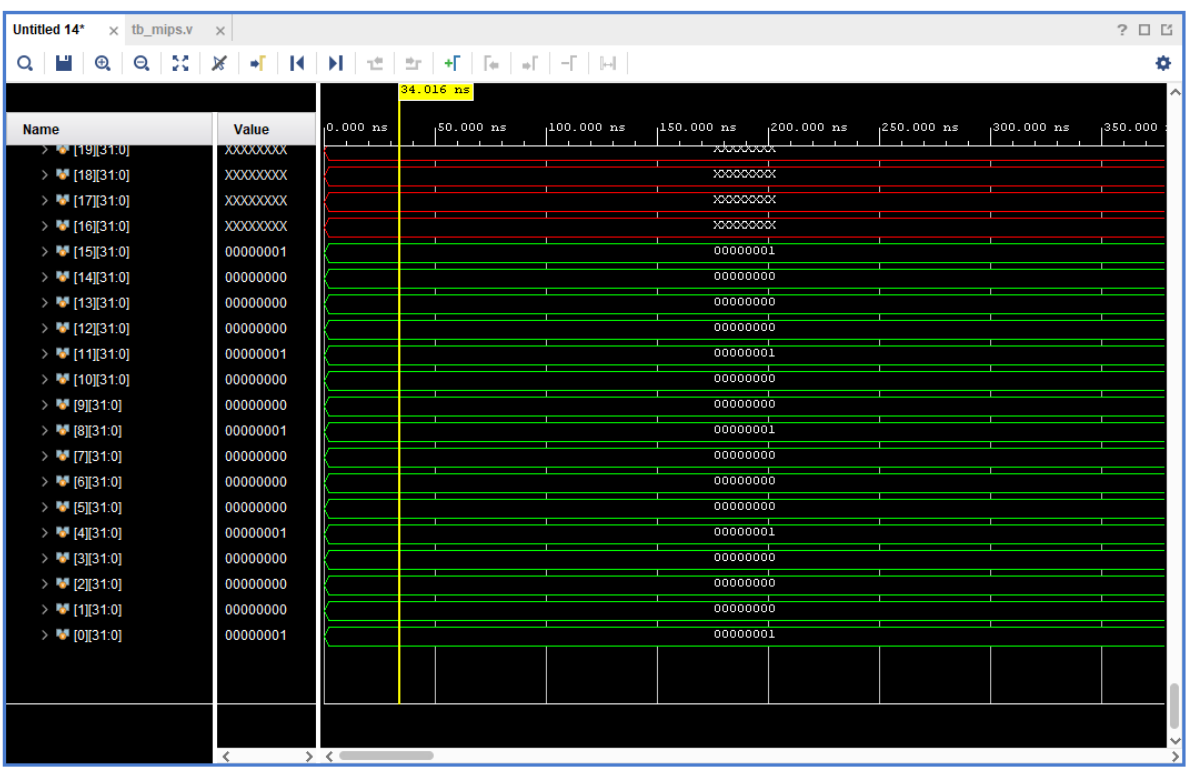
https://github.com/Utkar5hM/MIPS-single-cycle-Assignment

Synthesizing and simulating the given testbench code using Vivado

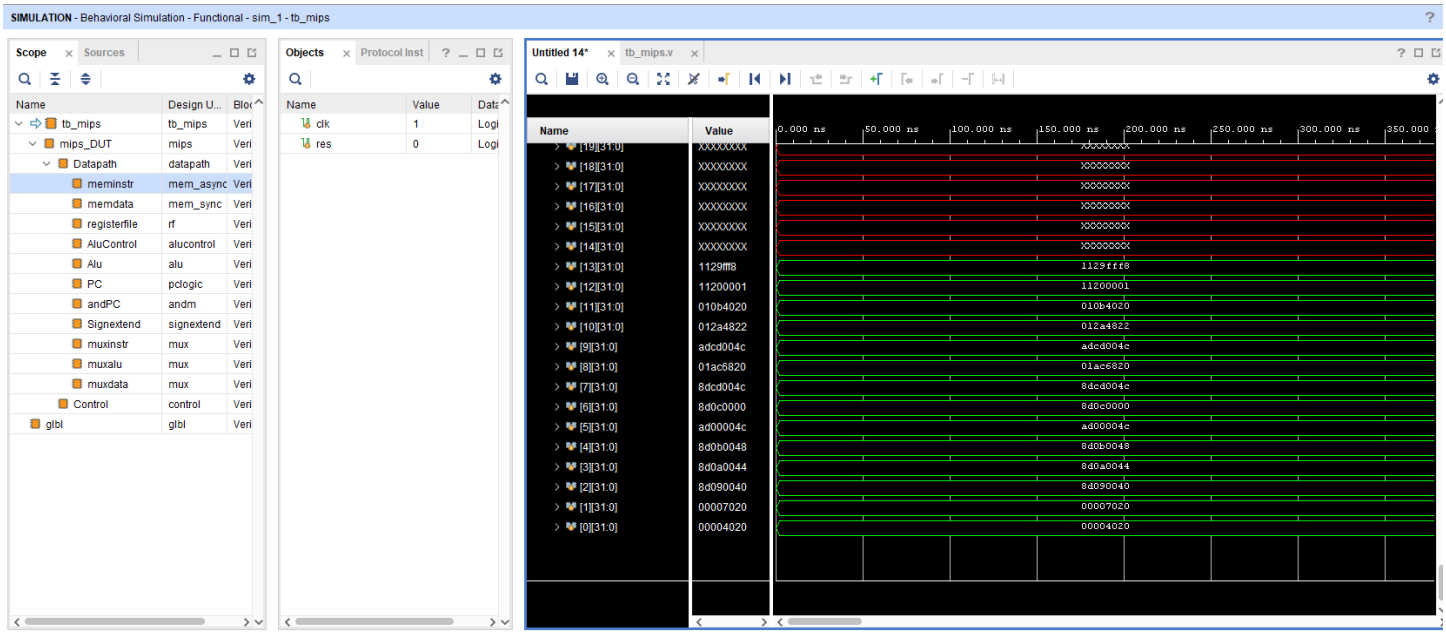
Simulation result:



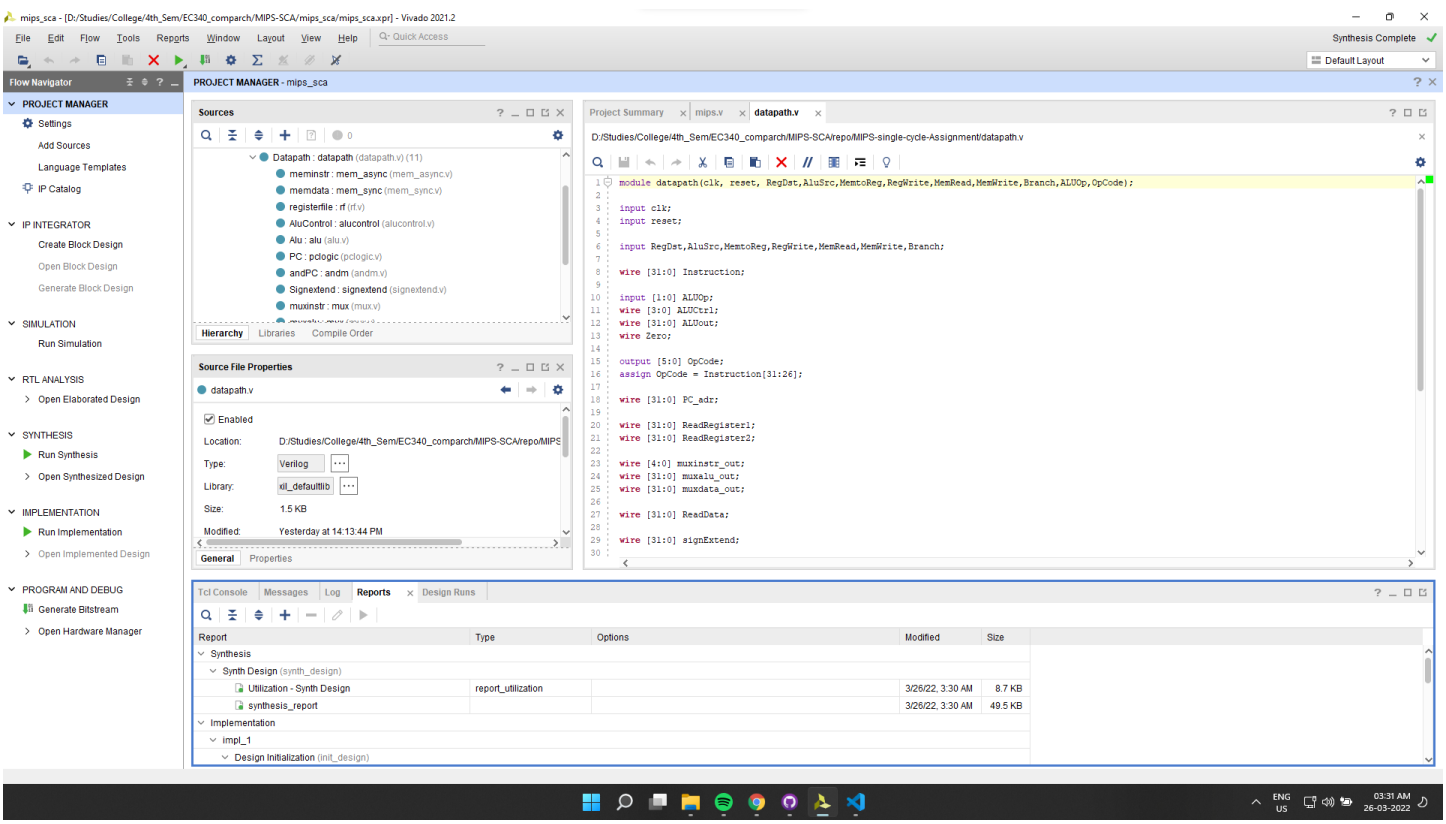
Memory sync(inputs from the given file):



Instruction memory(inputs from the given file):



Synthesis result: complete



1. Add the following MIPS instructions – addi, bne, j (use the MIPS instruction encoding format)

BNE:

31 – 26	25 – 21	20 – 16	15 – 0
BNE 000101	rs	rt	offset
6	5	5	16

Format: `BNE rs, rt, offset` **MIPS32**

We will start by adding the instruction bne as it is pretty similar to beq.

By analyzing how beq works, we can figure out that the decision making for branching happens in the **andm** module and it takes zero and branch condition as input while PCsel for the beq instruction.

```
assign out=inA&inB;
```

To add support for bne instruction. We need to have additional signal there such that it will check if we want to do the opposite where the branch is 1 and the zero should be false.

We will create a signal Ne(wire here) in the module as a input such that

```
assign out= (ne==0) ? inA&inB:(inA&(!inB));
```

now if ne is 0, it will proceed with normal beq instruction. Now we need to implement bne instruction in the control unit and add a additional register there that stores **Ne** and make required changes everywhere.

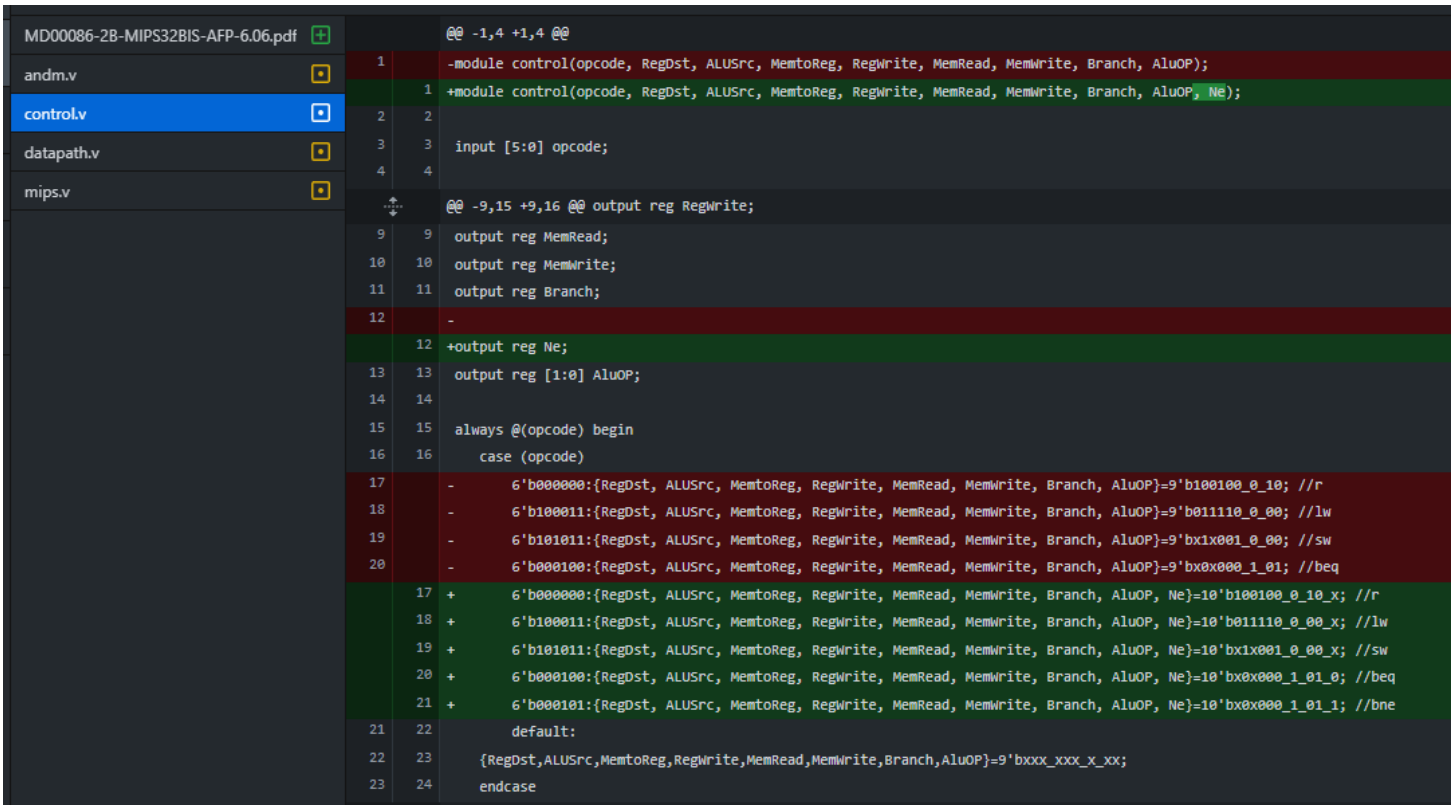
In Control unit, we will add the following new case for the bne instruction.

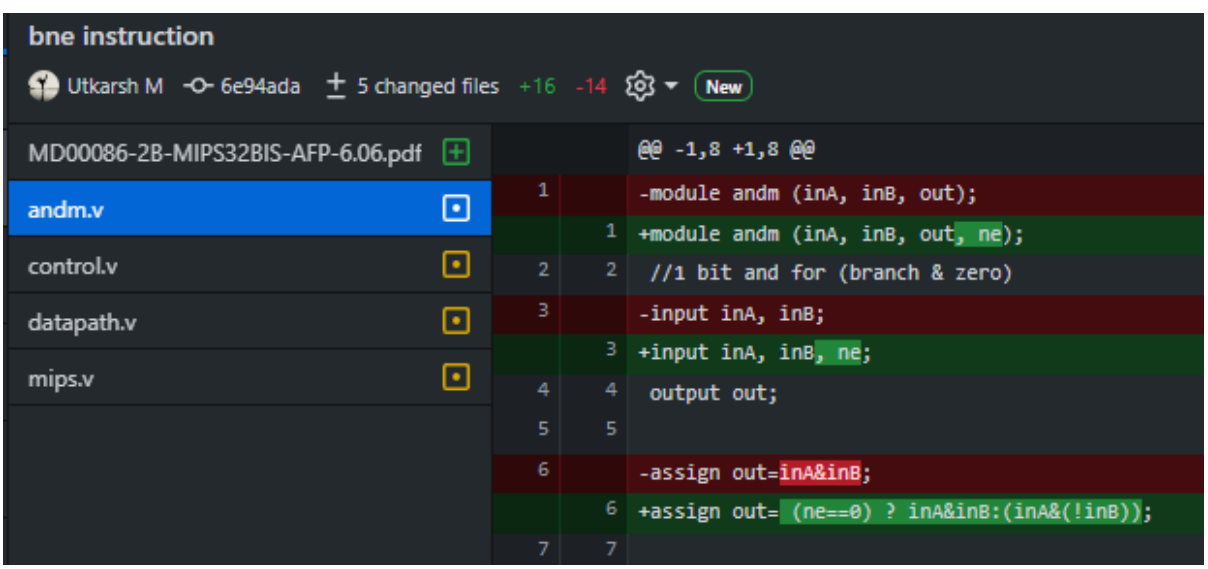
```
        6'b000101:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP,
Ne}=10'bx0x000_1_01_1; //bne
```

And set Ne=0; for other instructions.

<u>After making all the required changes to add bne instruction:</u>

Summarizing the changes with git version History:





MD00086-2B-MIPS32BIS-AFP-6.06.pdf
andm.v
control.v
datapath.v
mips.v

```diff
@@ -1,9 +1,9 @@
-module datapath(clk, reset, RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,OpCode);
+module datapath(clk, reset, RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,OpCode, Ne);

 input clk;
 input reset;

-input RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch;
+input RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch, Ne;

 wire [31:0] Instruction;

@@ -38,7 +38,7 @@ alucontrol AluControl(ALUOp, Instruction[5:0], ALUCtrl); //ALUControl
 alu Alu(ReadRegister1, muxalu_out, ALUCtrl, ALUout, Zero); //ALU

 pclogic PC(clk, reset, signExtend, PC_adr, PCsel); //generate PC
-andm andPC(Branch, Zero, PCsel); //AndPC (branch & zero)
+andm andPC(Branch, Zero, PCsel, Ne); //AndPC (branch & zero)
 signextend Signextend(signExtend, Instruction[15:0]); //Sign extend

 mux #(5) muxinstr(RegDst, Instruction[20:16],Instruction[15:11],muxinstr_out);//MUX for Write Register
```

MD00086-2B-MIPS32BIS-AFP-6.06.pdf
andm.v
control.v
datapath.v
mips.v

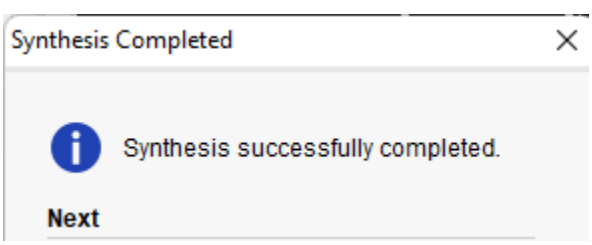
```diff
@@ -15,9 +15,10 @@ wire RegWrite;
 wire MemRead;
 wire MemWrite;
 wire Branch;
+wire Ne;

-datapath Datapath(clk,reset,RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,OpCode);
+datapath Datapath(clk,reset,RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,OpCode, Ne);

-control Control(OpCode,RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,Branch,ALUOp);
+control Control(OpCode,RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,Branch,ALUOp, Ne);

 endmodule
```

Synthesis result after adding bne instruction:



addi:

31–26	25–21	20–16	15–0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format: `ADDI rt, rs, immediate` **MIPS32, removed in Release 6**

Description: `GPR[rt] ← GPR[rs] + immediate`

The 16-bit signed *immediate* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rt*.

Restrictions:

Now we will add support for the addi instruction. It is a I -format Instruction, So its similar to lw and sw but not by much.

After looking at add and lw instruction. We can see that we just need to add proper case statement for addi instruction.

As we need instruction[20:16] as the destination register address.

RegDst = 0

As we need immediate value as 2nd operand for ALU.

ALUSrc = 1

As we will not be reading from or writing to the memory.

MemtoReg=0

MemRead=0

MemWrite=0

As we will be writing to the register

RegWrite=1

Since we need ALU to do the add operation, we can just use 00 as AluOp just like in case of lw or sw. Therefore,

AluOp=00

Ne =x; as we don't need to write to PC.

Therefore

In Control module, adding a new case statement:

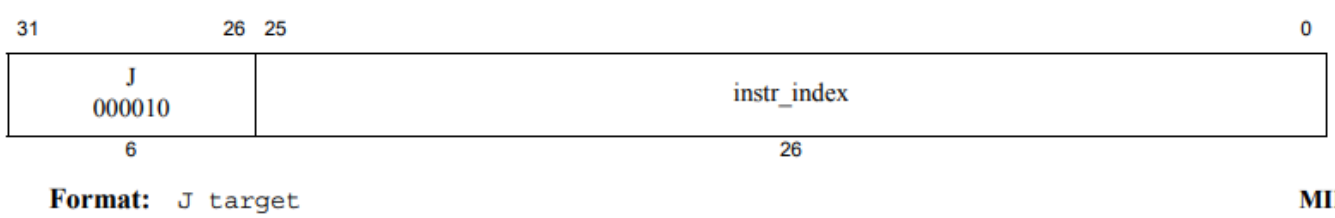
```
        6'b001000:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP,
Ne}=10'b010100_0_00_x; //addi
```

Synthesis result after adding addi Instruction:

Report	Type	Opti...	Modified
Synthesis			
Synth Design (synth_design)			
Utilization - Synth Design	report_utilization		3/26/22, 5:49 AM
synthesis_report			3/26/22, 5:49 AM

We can see that we have successfully implemented addi instruction.

J:



Format: `J target` **MIPS32**

Now we will add our final instruction J. It follows Jump addressing and since our current MIPS CPU doesn't support such instructions. We need to make major changes.

First in Control Module, we will need to send another output, rather than creating a new field. We will just extend our previously added bit Ne to 2 bit size.

Then let us handle the situation like:

Ne = 2'b00 = for the normal beq instruction.

Ne = 2'b01 = for handling bne.

Ne = 2'b10 = for handling J instruction

Making Appropriate changes in control module we will have the following case statement for J:

```
        6'b000010:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP,
Ne}=11'bxxxxxx_1_xx_10; //J
```

After making similar changes in other case statements to support the 11th bit.

Now for the Jump Address, we will create a new module Jaddress. This will take PC[31:28], Instruction[25:0] as input and will give output = PC[31:28] +2'b00 + (Instruction[25:00])

Note: Not output = PC[31:28] +(Instruction[25:00]<2) because here the memory is designed to be Word addressable and not Byte Addressable..

```
module Jaddress(
    input [25:0] in,
    input [3:0] pc_in,
    output [31:0] out
    );
    assign out ={pc_in, {2'b00}, in};

endmodule
```

Now to give this as an Input to the Program Counter, We will take a Mux with Ne[1] as select as that decides the factor to do the normal Branch instructions or Jump Instruction. The mux inputs will be the output of Jaddress and SignExtend as they are the required PC addresses for respective instructions (J and branch instructions.)

Since we already do have a Mux switch Module, we will just instantiate that and Jaddress module.

```
Jaddress jadd(Instruction[25:0], PC_adr[31:28], Jadr);
mux #(32) muxPC(Ne[1], signExtend, Jadr,  muxPC_in);
```

Now we need to change the logic for PCSel. This is handled in the andm module.

After making appropriate changes our andm module will be:

```
module andm (inA, inB, out, ne);
//1 bit and for (branch & zero)
input inA, inB;
input [1:0] ne;
output [1:0]out;

assign out= (ne==0) ? inA&inB:((ne==2'b01) ? (inA&(!inB)):((ne==2'b10) ?
({{1'b1},inA}):0));

endmodule
```

We need to make changes in PC to handle Jump Instruction. We will make use of ternary operator to operate depending upon the instruction.

After making changes to PC sub module instantiation

```
pclogic PC(clk, reset, muxPC_in, PC_adr, PCsel); //generate PC
```

```
module pclogic(clk, reset, ain, aout, pcsel);

input reset;
input clk;
input [31:0] ain;
//pecsel = branch & zero
input [1:0]pcsel;

output reg [31:0] aout;

always @(posedge clk ) begin
    if (reset==1)
        aout<=32'b0;
    else
        if ((pcsel==2'b00) || (pcsel==2'b10)) begin
            aout<=aout+1;
        end
        if (pcsel==2'b01) begin
            aout<=ain+aout+1; //branch
    end
        if (pcsel==2'b11) begin
            aout<=ain; //branch
    end
end


endmodule
```

After making required change like changing wire's width size wherever required.

We can use git version History to summarize the changes :

J instruction

Utkarsh M f463edd 7 changed files +60 -19 New

- Jaddress.v
- alucontrol.v
- andm.v
- control.v
- datapath.v
- mips.v
- pclogic.v

```diff
@@ -0,0 +1,29 @@
+`timescale 1ns / 1ps
+//////////////////////////////////////////////////////////////////////////////////
+// Company:
+// Engineer:
+//
+// Create Date: 26.03.2022 06:23:58
+// Design Name: Utkarsh M
+// Module Name: Jaddress
+// Project Name:
+// Target Devices:
+// Tool Versions:
+// Description:
+//
+// Dependencies:
+//
+// Revision:
+// Revision 0.01 - File Created
+// Additional Comments:
+//
+//////////////////////////////////////////////////////////////////////////////////
+
+
+module Jaddress(
+    input [25:0] in,
+    input [3:0] pc_in,
+    output [31:0] out
+    );
+    assign out ={pc_in,in,{2'b00}};
+endmodule
```

Jaddress.v
alucontrol.v
andm.v
control.v
datapath.v
mips.v
pclogic.v

```diff
@@ -8,7 +8,7 @@ output reg [3:0] AluCtrl;

 always@(AluOp or FnField)begin
    casex({AluOp,FnField})
-       8'b00_xxxxxx:AluCtrl=4'b0010; //lw / sw
+       8'b00_xxxxxx:AluCtrl=4'b0010; //lw / sw / add
        8'b01_xxxxxx:AluCtrl=4'b0110; //beq
        8'b1x_xx0000:AluCtrl=4'b0010; //add
        8'b1x_xx0010:AluCtrl=4'b0110; //sub
```

Jaddress.v
alucontrol.v
andm.v
control.v
datapath.v
mips.v
pclogic.v

```diff
@@ -1,8 +1,9 @@
 module andm (inA, inB, out, ne);
 //1 bit and for (branch & zero)
-input inA, inB, ne;
-output out;
+input inA, inB;
+input [1:0] ne;
+output [1:0]out;

-assign out= (ne==0) ? inA&inB:(inA&(!inB));
+assign out= (ne==0) ? inA&inB:((ne==2'b01) ? (inA&(!inB)):((ne==2'b10) ? ({{1'b1},inA}):0));

 endmodule
```

Jaddress.v
alucontrol.v
andm.v
control.v
datapath.v
mips.v
pclogic.v

```diff
@@ -9,17 +9,18 @@ output reg RegWrite;
 output reg MemRead;
 output reg MemWrite;
 output reg Branch;
-output reg Ne;
+output reg [1:0] Ne;
 output reg [1:0] AluOP;

 always @(opcode) begin
    case (opcode)
-       6'b000000:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=10'b100100_0_10_x; //r
-       6'b100011:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=10'b011110_0_00_x; //lw
-       6'b101011:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=10'bx1x001_0_00_x; //sw
-       6'b000100:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=10'bx0x000_1_01_0; //beq
-       6'b000101:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=10'bx0x000_1_01_1; //bne
-       6'b001000:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=10'b010100_0_00_x; //addi
+       6'b000000:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'b100100_0_10_0x; //r
+       6'b100011:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'b011110_0_00_0x; //lw
+       6'b101011:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'bx1x001_0_00_0x; //sw
+       6'b000100:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'bx0x000_1_01_00; //beq
+       6'b000101:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'bx0x000_1_01_01; //bne
+       6'b001000:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'b010100_0_00_0x; //addi
+       6'b000010:{RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, AluOP, Ne}=11'bxxxxxx_1_xx_10; //J
        default:
    {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,AluOP}=9'bxxx_xxx_x_xx;
    endcase
```

J instruction

Utkarsh M · f463edd · 7 changed files +60 -19 · New

- Jaddress.v
- alucontrol.v
- andm.v
- control.v
- datapath.v
- mips.v
- pclogic.v

```diff
@@ -3,11 +3,11 @@ module datapath(clk, reset, RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Bra
 input clk;
 input reset;

-input RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch, Ne;
+input RegDst,AluSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch;

 wire [31:0] Instruction;

-input [1:0] ALUOp;
+input [1:0] ALUOp, Ne;
 wire [3:0] ALUCtrl;
 wire [31:0] ALUout;
 wire Zero;
@@ -24,11 +24,15 @@ wire [4:0] muxinstr_out;
 wire [31:0] muxalu_out;
 wire [31:0] muxdata_out;

+wire [31:0] muxPC_in;
+
+wire [31:0] Jadr;
+
 wire [31:0] ReadData;
```

```diff
-wire PCsel;
+wire [1:0] PCsel;

 mem_async meminstr(PC_adr[7:0],Instruction); //Instruction memory
 mem_sync memdata(clk, ALUout[7:0], ReadData, ReadRegister2, MemRead, MemWrite); //Data memory
@@ -37,10 +41,13 @@ rf registerfile(clk,RegWrite,Instruction[25:21],Instruction[20:16],muxinstr_out,
 alucontrol AluControl(ALUOp, Instruction[5:0], ALUCtrl); //ALUControl
 alu Alu(ReadRegister1, muxalu_out, ALUCtrl, ALUout, Zero); //ALU

-pclogic PC(clk, reset, signExtend, PC_adr, PCsel); //generate PC
+pclogic PC(clk, reset, muxPC_in, PC_adr, PCsel); //generate PC
 andm andPC(Branch, Zero, PCsel, Ne); //AndPC (branch & zero)
 signextend Signextend(signExtend, Instruction[15:0]); //Sign extend

+Jaddress jadd(Instruction[25:0], PC_adr[31:28], Jadr);
+mux #(32) muxPC(Ne[1], signExtend, Jadr,  muxPC_in);
+
```

Jaddress.v
alucontrol.v
andm.v
control.v
datapath.v
mips.v
pclogic.v

```diff
@@ -15,7 +15,7 @@ wire RegWrite;
 wire MemRead;
 wire MemWrite;
 wire Branch;
-wire Ne;
+wire [1:0] Ne;

 datapath Datapath(clk,reset,RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,OpCode, Ne);
```

Jaddress.v
alucontrol.v
andm.v
control.v
datapath.v
mips.v
pclogic.v

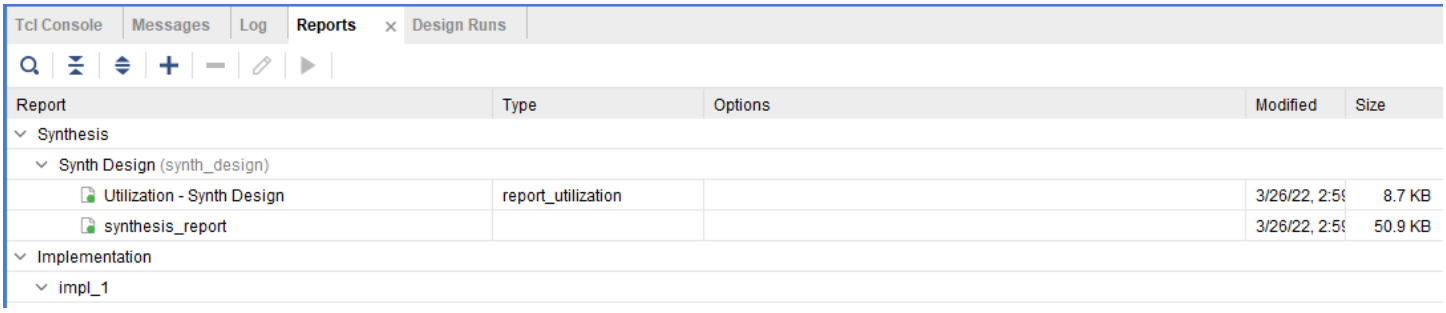
```diff
@@ -4,7 +4,7 @@ input reset;
 input clk;
 input [31:0] ain;
 //pecsel = branch & zero
-input pcsel;
+input [1:0]pcsel;

 output reg [31:0] aout;

@@ -12,12 +12,15 @@ always @(posedge clk ) begin
    if (reset==1)
        aout<=32'b0;
    else
-       if (pcsel==0) begin
+       if ((pcsel==2'b00) || (pcsel==2'b10)) begin
            aout<=aout+1;
        end
-       if (pcsel==1) begin
+       if (pcsel==2'b01) begin
            aout<=ain+aout+1; //branch
    end
+       if (pcsel==2'b11) begin
+           aout<=ain; //branch
+   end
 end
```

```diff
@@ -25,5 +25,5 @@ module Jaddress(
     input [3:0] pc_in,
     output [31:0] out
     );
-    assign out ={pc_in,in,{2'b00}};
+    assign out ={pc_in, {2'b00}, in};
 endmodule
```

Synthesis Results: Success



Report	Type	Options	Modified	Size
Synthesis				
Synth Design (synth_design)				
Utilization - Synth Design	report_utilization		3/26/22, 2:59	8.7 KB
synthesis_report			3/26/22, 2:59	50.9 KB
Implementation				
impl_1				

2. Test using the assembly level code for Q1 in Exercise L22 (Assume that you have an array of 10 elements with base address in $s0. Write an assembly program to find the minimum value from the array and swap it with the last element in the array) Use SPIM to get the machine language code. Make sure your code uses the 3 new instructions you added (addi, bne & j)

->

After making slight changes to the MIPS assembly code written in Exercise L22 to use only the supported instruction. we get: (we will only use the part from main to done, instructions like to print on the console etc have been removed)

```
        .data
#  storing required data into memory
array:  .word 67 43 3 7 2 35 9 62 4 8

        .text
        .globl main
main:
        add     $t1, $zero, $zero   # i (index) = 0
        add     $s0, $zero, $zero   # base address =0
        # initializing minimum =  storing a[0] value in minimum
        lw      $t0, 0($s0)
        add     $t7, $zero, $zero    # index of minimum
        # j (index) = 0 for printing updated array
        add     $t9, $zero, $zero
        addi    $s1, $zero, 10

loop:   slt    $t3, $t1, $s1         # if i == 10 goto done
```

```
        beq     $t3, $zero, swapmin
        add     $t6, $t1, $t1         # offset = index * 4
        add     $t6, $t6, $t1         # offset = index * 4
        add     $t6, $t6, $t1         # offset = index * 4
        add     $t5, $s0, $t6       # address = base_address + offset;
        lw      $t4, 0($t5)         # t4= arr[i]
        slt     $t2, $t4, $t0       # setting less than in t2
        # switching to min branch if a[i]<current_min
        bne     $t2, $zero, min
# label b_loop for returning after min has been updated
b_loop: addi    $t1, $t1, 1         # i++
        j       loop
# min updates the minimum value and the index containing it
min:    add     $t0, $zero, $t4     # updating minimum value
        # updating index of minimum so that we can swap values letter
        add     $t7, $zero, $t1
        j      b_loop
# for swapping the minimum and last array
swapmin:
        lw      $t4,  36($s0)     # saving the last value of the array
        add     $t6, $t7, $t7        # offset = min_index *4
        add     $t6, $t6, $t7
        add     $t6, $t6, $t7
        sw      $t0, 36($s0)        # storing min value in last position
        add     $t5, $s0, $t6       # min_address = base + offset
        # storing the value of last element in the position of minimum value
        sw      $t4, 0($t5)
        j       done          # to print the updated array


done:
```

In the final code we need to make change to make it compatible to word addressable instead of byte addressable.

So we will not multiply indexes/offsets by 4.

IIke where we previously did multiply by 4(here add is used 4times). We just add it ones with a zero.

And where load store operation occurs.

For example for store instruction.

```
        sw      $t0, 36($s0)        # storing min value in last position
```

We will divide this by 4,

So in the end it becomes

```
        sw      $t0, 9($s0)        # storing min value in last position
```

Considering this for the entire code:

The final assembly hex Code:

```
00004820  //add $9, $0, $0           ; 8: add $t1, $zero, $zero # i (index) = 0
00008020  //add $16, $0, $0          ; 9: add $s0, $zero, $zero # base address =0
8e080000  //lw $8, 0($16)            ; 11: lw $t0, 0($s0)
00007820  //add $15, $0, $0          ; 12: add $t7, $zero, $zero # index of minimum
0000c820  //add $25, $0, $0          ; 14: add $t9, $zero, $zero
2011000a  //addi $17, $0, 10         ; 15: addi $s1, $zero, 10
0131582a  //slt $11, $9, $17         ; 17: slt $t3, $t1, $s1 # if i == 10 goto done
1160000c  //beq $11, $0, 12 [swapmin-0x00400040]
01207020  //add $14, $9, $zero          ; 19: add $t6, $t1, $t1 # offset = index //
020e6820  //add $13, $16, $14        ; 22: add $t5, $s0, $t6 # address = base_address +
offset;
8dac0000  //lw $12, 0($13)           ; 23: lw $t4, 0($t5) # t4= arr[i]
0188502a  //slt $10, $12, $8         ; 24: slt $t2, $t4, $t0 # setting less than in t2
15400002  //bne $10, $0, 2 [min-0x0040005c]
21290001  //addi $9, $9, 1           ; 28: addi $t1, $t1, 1 # i++

08000006  //j 6 should go to line 7 as indexing is from 0 we jump to 6  j loop

000c4020  //add $8, $0, $12          ; 31: add $t0, $zero, $t4 # updating minimum value
00097820  //add $15, $0, $9          ; 33: add $t7, $zero, $t1
0800000D  //j Go to line 14 [b_loop] ; 34: j b_loop
8E0C0009  //lw $12, 9($16)          ; 37: lw $t4, 9($s0) # saving the last value of the
array
```

```
01E07020  //add $14, $15, $zero        ; 38: add $t6, $t7, $zero # offset = min_index //
AE080009  //sw $8, 9($16)           ; 41: sw $t0, 9($s0) # storing min value in last
position
020e6820  //add $13, $16, $14        ; 42: add $t5, $s0, $t6 # min_address = base + offset
ADEC0000  //sw $12, 0($15)           ; 44: sw $t4, 0($t5)
11200001 //beq
1120FFFF //beq
```

Note: text after // is ignored while reading the data.

It can be seen from the assembly hex code that we have removed the multiple add operations while accessing/storing in memory to avoid the multiples of 4 which is usually required for byte addressing.

Similarly Jump/branch instructions are modified to go to the required nth line considering the same.

We need to store our array in the memory(mem_sync) and store the base address in memory.

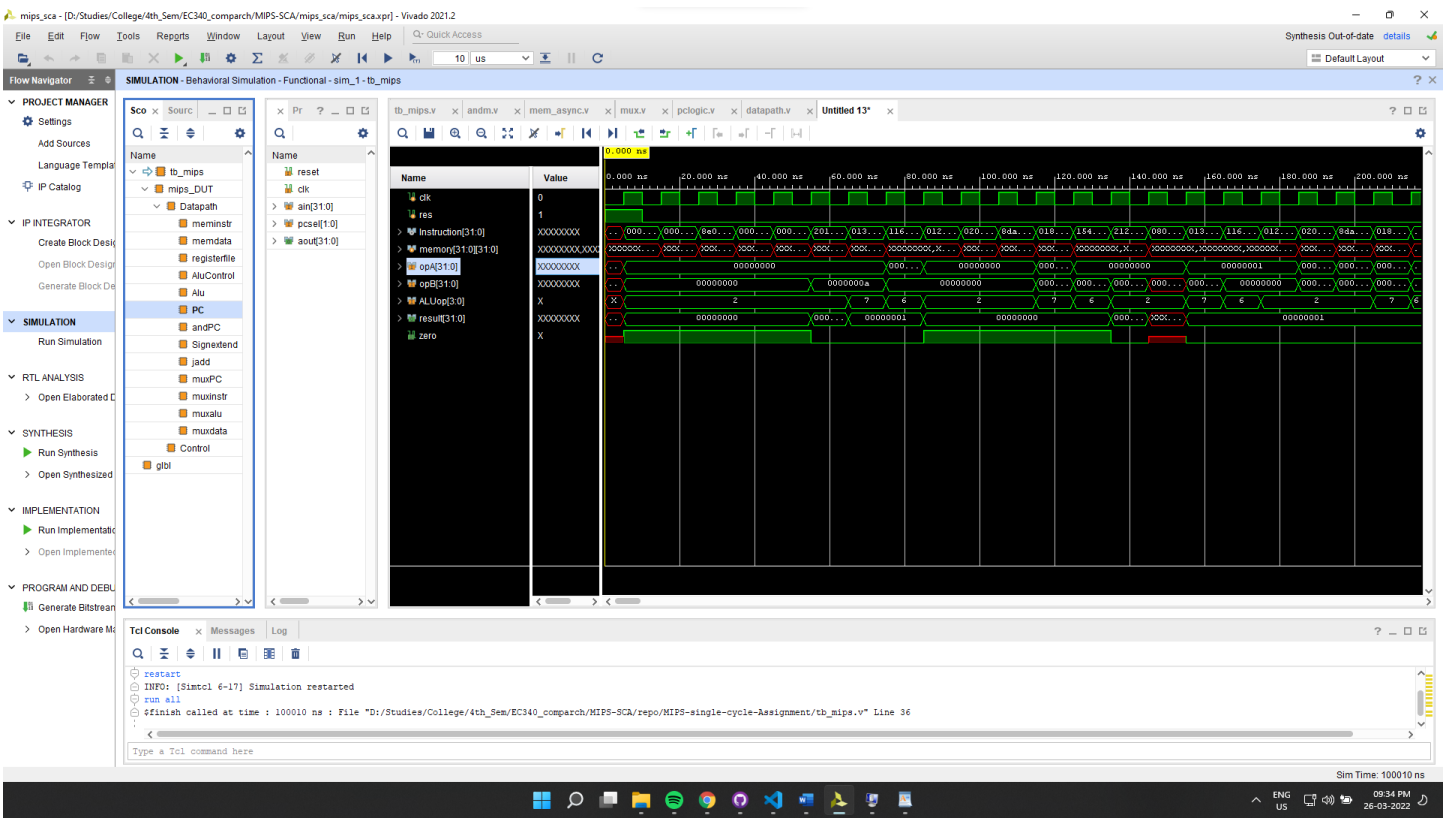
Now memory data : we will store the array starting at location 0 in the memory.

We will use the same input as from the previous mips assembly code.
67 43 3 7 2 35 9 62 4 8

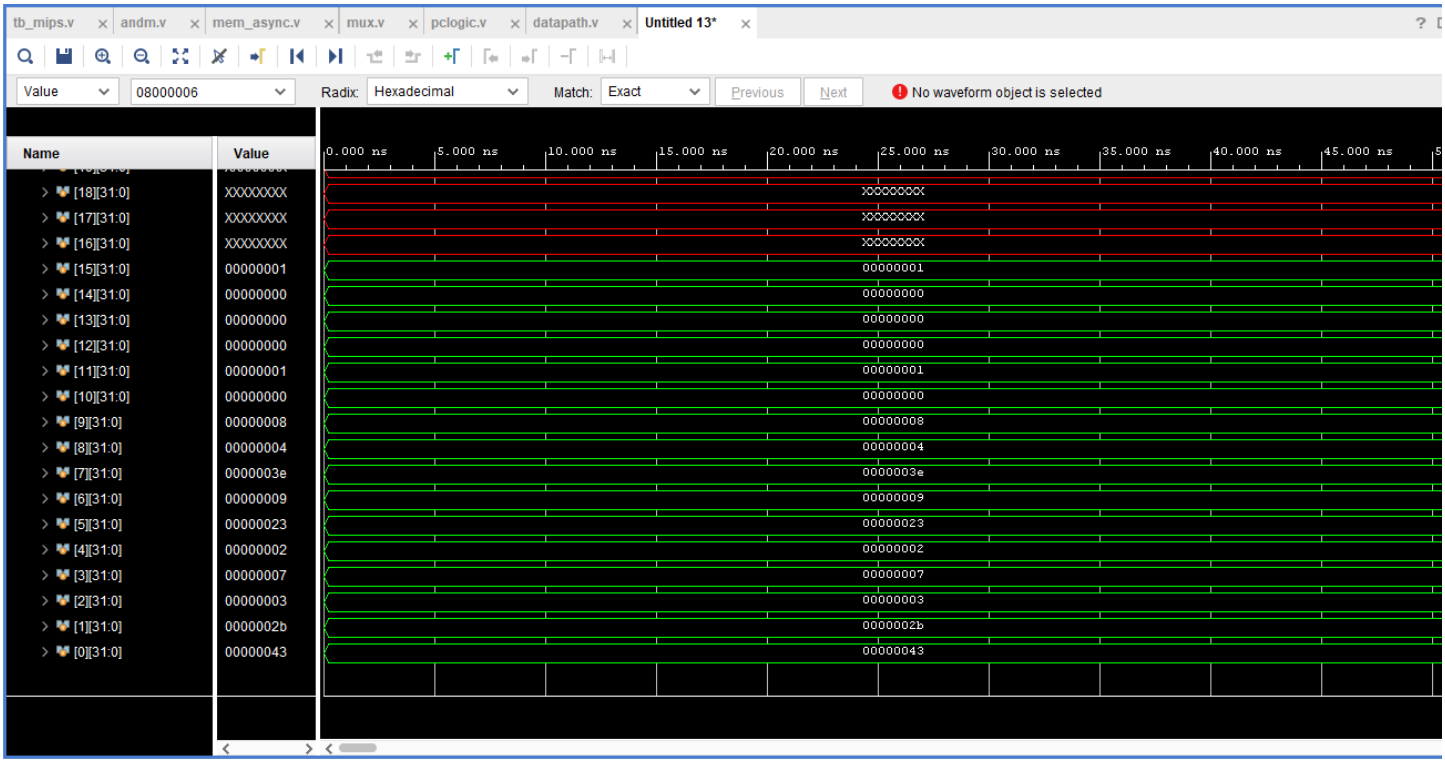
```
0000_0043 //0
0000_002B //1
0000_0003 //2
0000_0007 //3
0000_0002 //4
0000_0023 //5
0000_0009 //6
0000_003E //7
0000_0004 //8
0000_0008 //9
0000_0000 //10
0000_0001 //11
```

```
0000_0000 //12
0000_0000 //13
0000_0000 //14
0000_0001 //15
@40
0000_0010
@44
0000_0001
@48
0000_0001
```

Simulation:

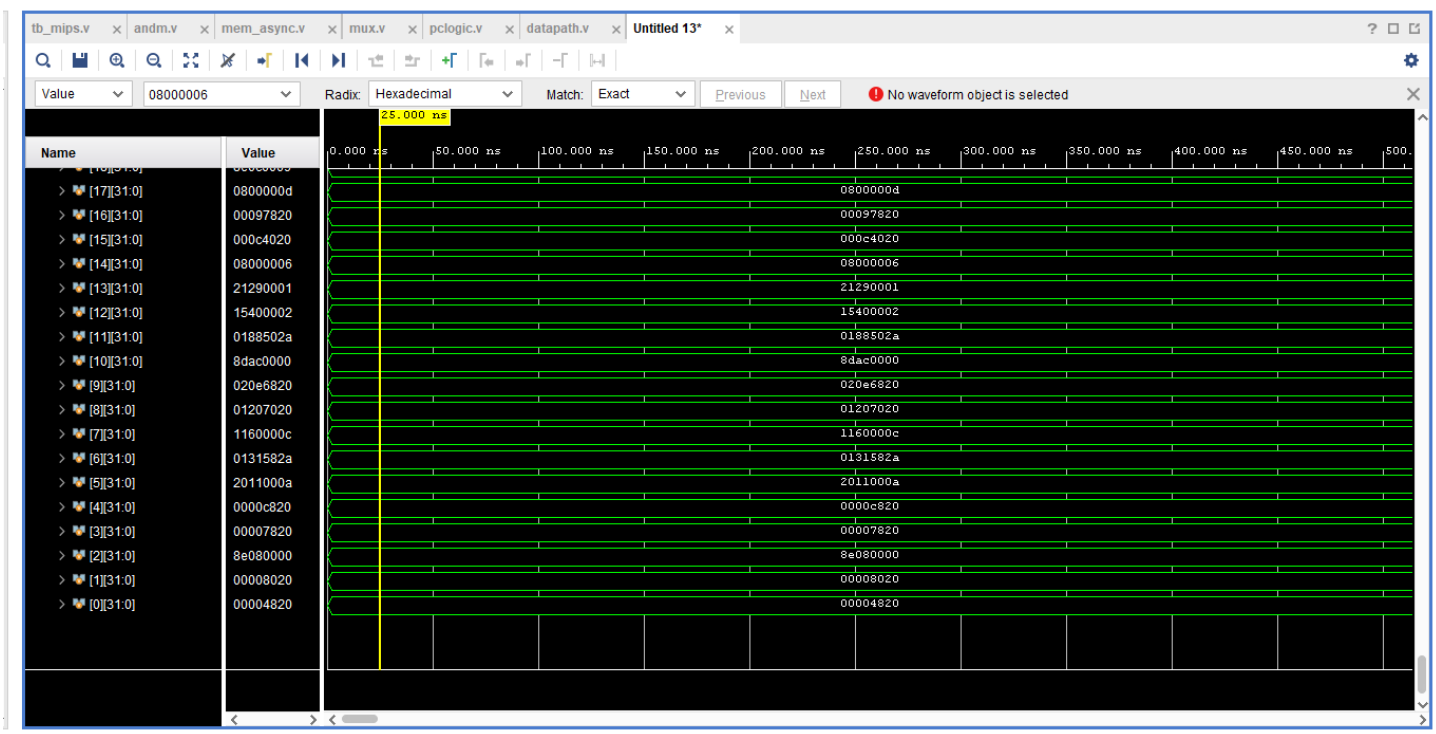


Initial memory:



We can see that the memory does store the data from our input file.

Instructions memory:



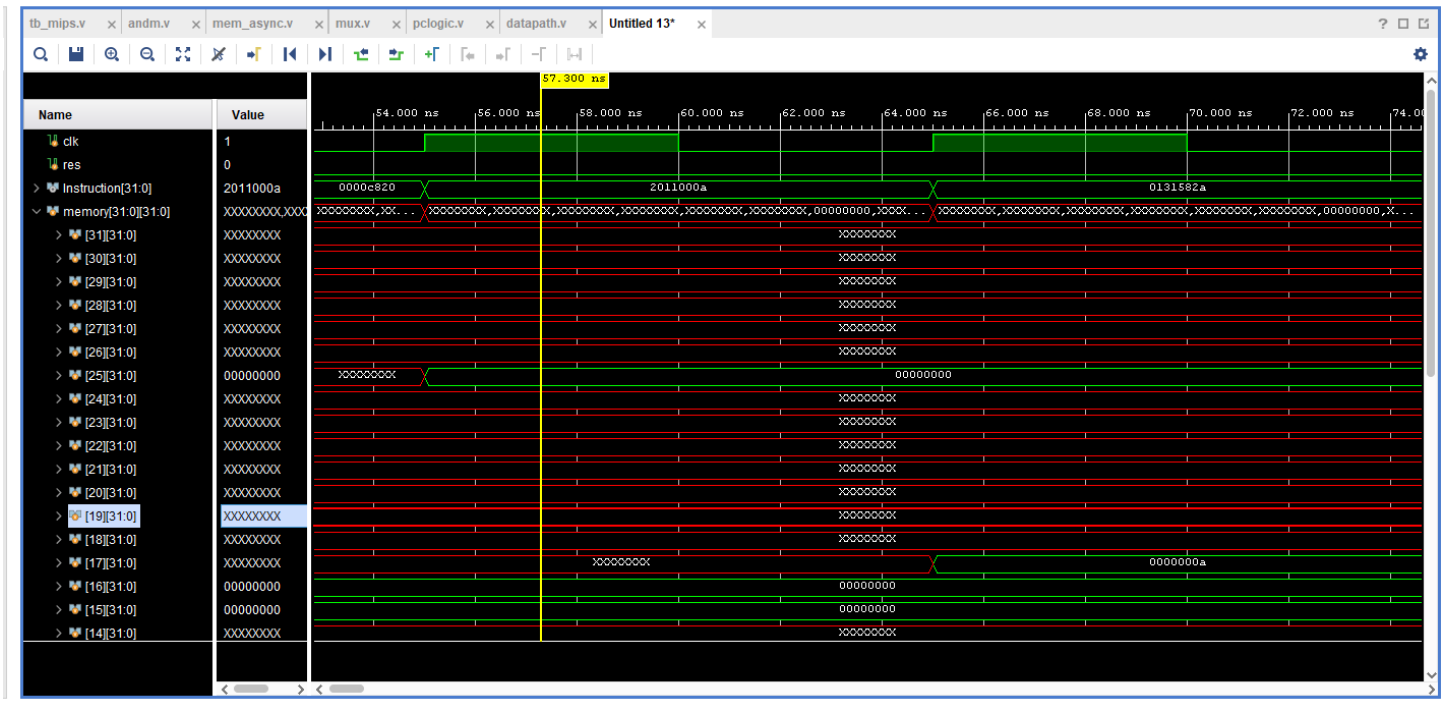
We can see that all the instructions are added into the instruction memory.

We will verify the working of individual Instructions.

addi:

For the instruction:

```
2011000a  //addi $17, $0, 10        ; 15: addi $s1, $zero, 10
```



We can see that after the value of $17($s1) after the instruction 2011000 is 0000000a which is equal to 10 and matches our required result.

Bne:

For the Instruction:

```
15400002  //bne $10, $0, 2 [min-0x0040005c]
```

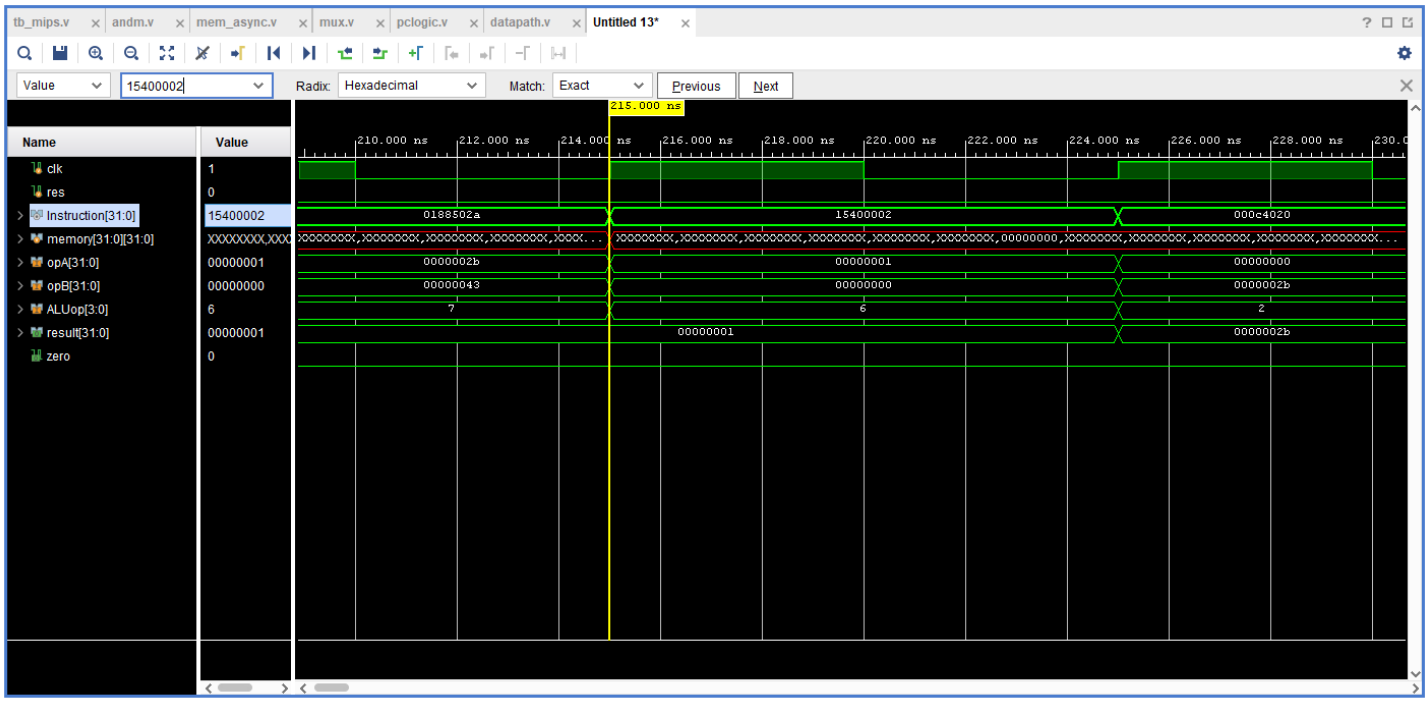
Next Instruction:

```
21290001  //addi $9, $9, 1          ; 28: addi $t1, $t1, 1 # i++
```

Instruction If Branched:

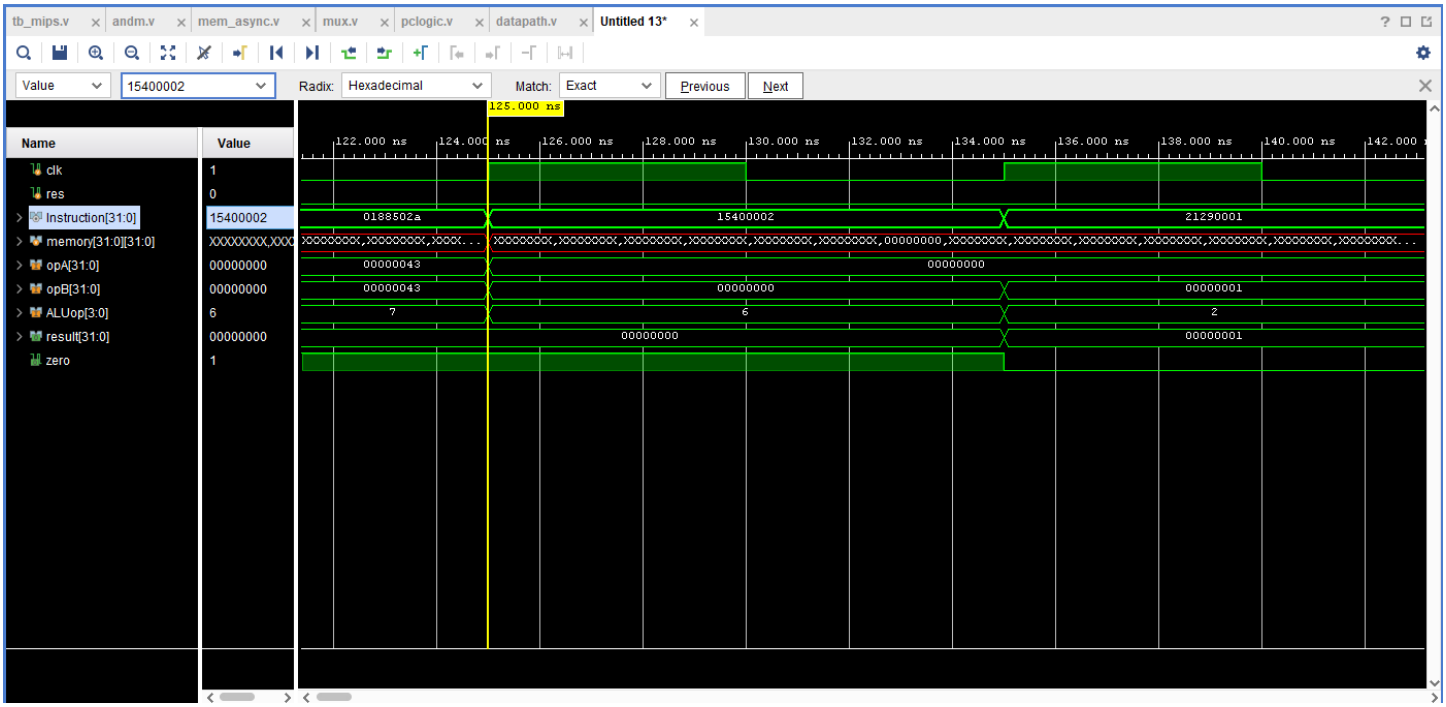
```
000c4020  //add $8, $0, $12         ; 31: add $t0, $zero, $t4 # updating minimum value
```

<u>When branching is done:</u> (zero from alu is low)



We can see that the instruction changes from 15400002 to 000c4020.

When branching is not done: (zero from alu is high)



We can see that it goes from 15400002 to 21290001

From the above two conditions we can see that it work as required.

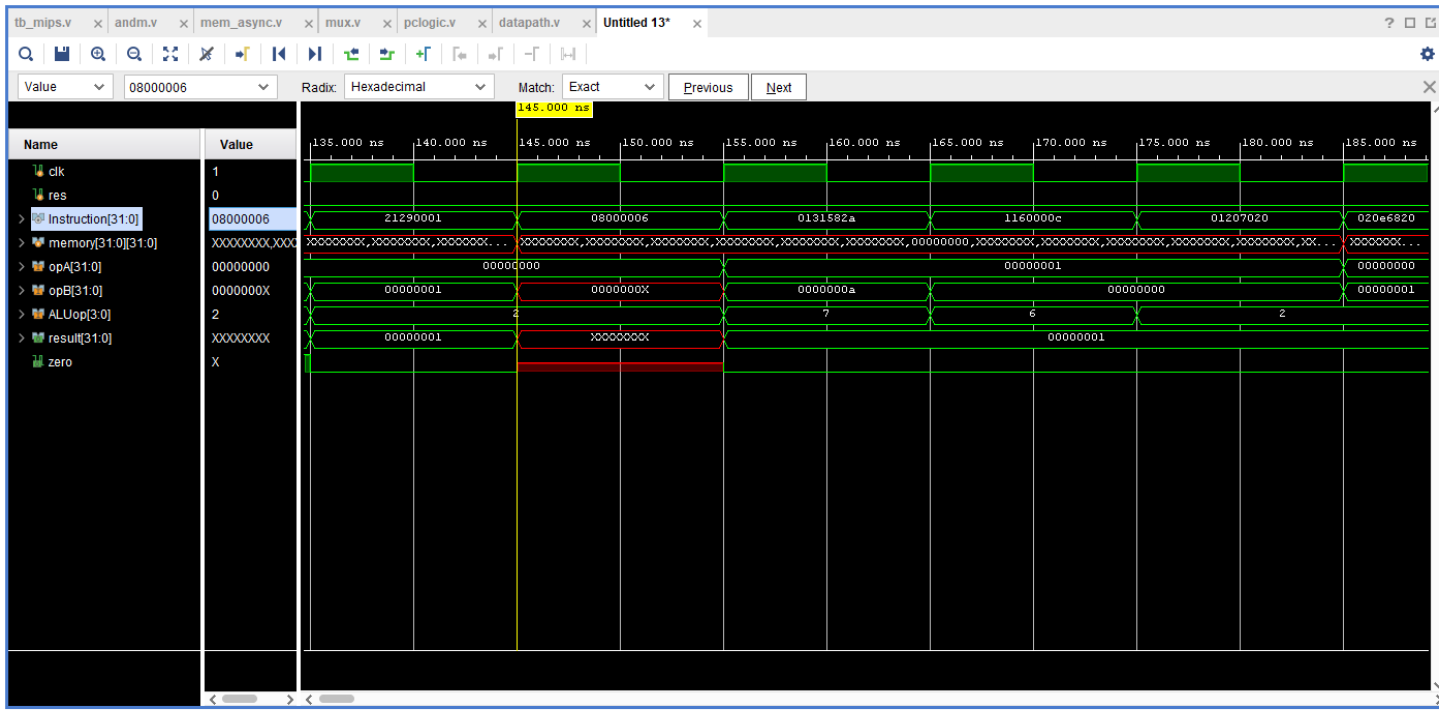
J:

For the Instruction:

```
08000006  //j 6 should go to line 7 as indexing is from 0 we jump to 6  j loop
```

Instruction at line 7 (counting from 1):

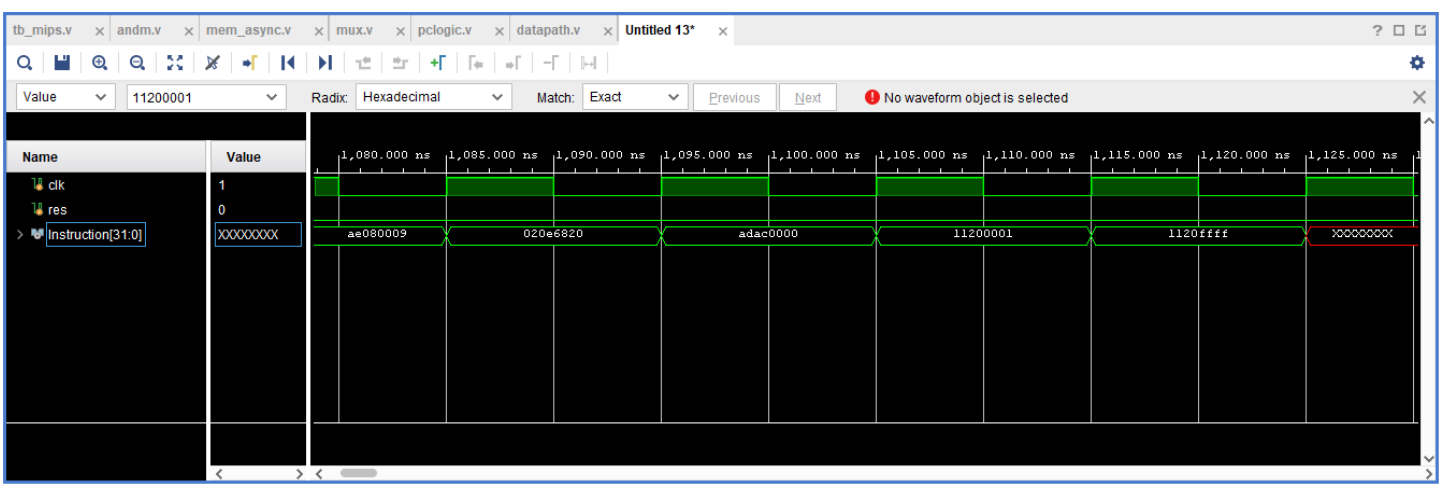
```
0131582a  //slt $11, $9, $17         ; 17: slt $t3, $t1, $s1 # if i == 10 goto done
```



We can see that the jump instruction does change the next instruction to 0131582 from 0800006 instead of 000c4020 which is at next line.

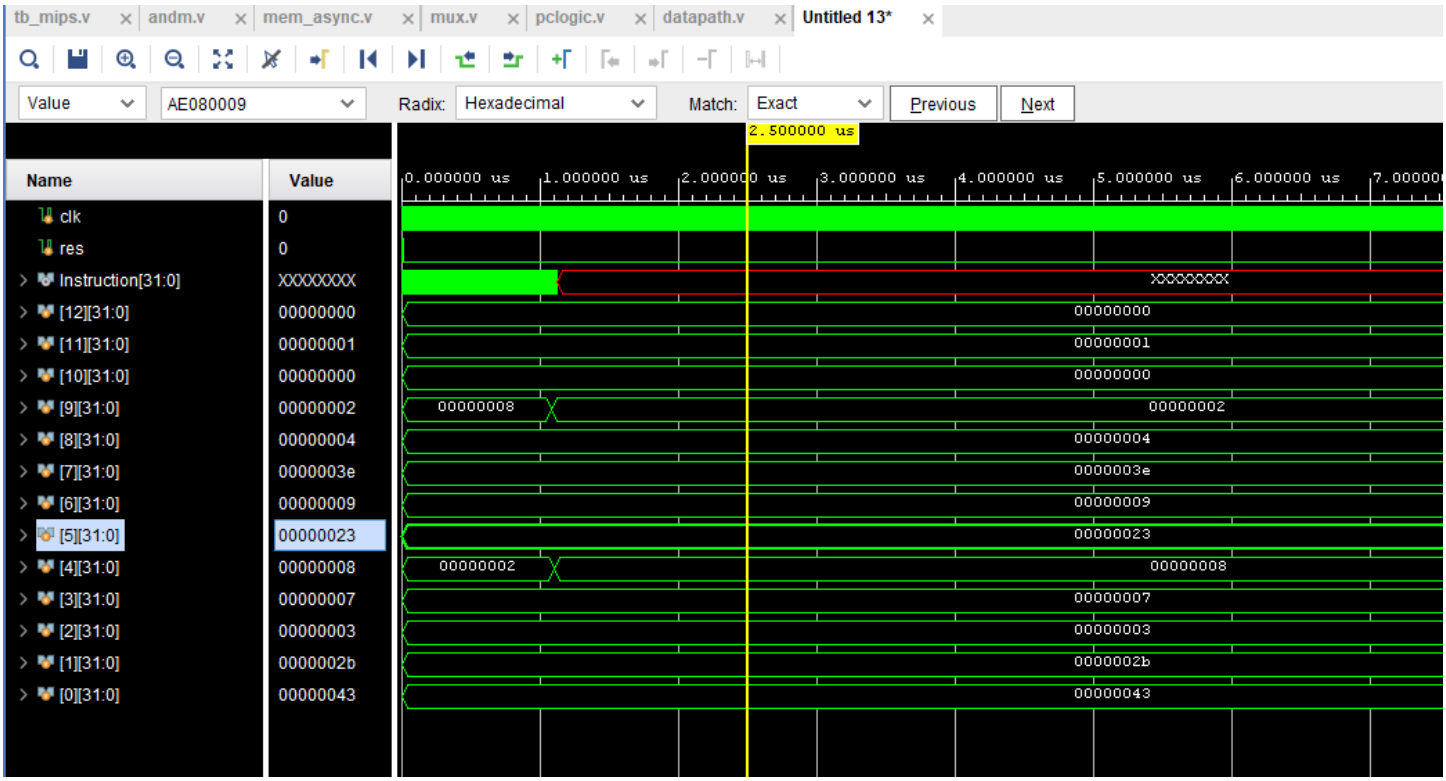
Confirming all the 3 instructions added works properly.

Competition of Execution:



We can see that the program does reach the end(last instruction).

Final Memory:



We can see that the value 00000008 in the last index gets replaced by the minimum value 00000002 and it gets placed into where 00000002 was present.

Confirming Our Code works as expected.