Computer Organization and Design.

Design and Implementation of a MIPS CPU with Mutlicycle Datapath .

Due to: 12/07/2012

Adalberto Claudio.
A20294552.

Index

Introduction 3

Implementation 4
- Program Counter 4
- ALU 5
- Bits5Reg 5
- BitsReg 5
- Instruction Register 5
- Memory 5
- Registers 5
- Mux2 6
- Mux2_1 6
- Mux2_5b 6
- ClockBox 6
- Gate 6
- Or2 6
- ShiftLeft2 7
- ShiftLeft22 7
- SignExtend 7
- Mux4 7
- CUnit 7
- New-Current 7

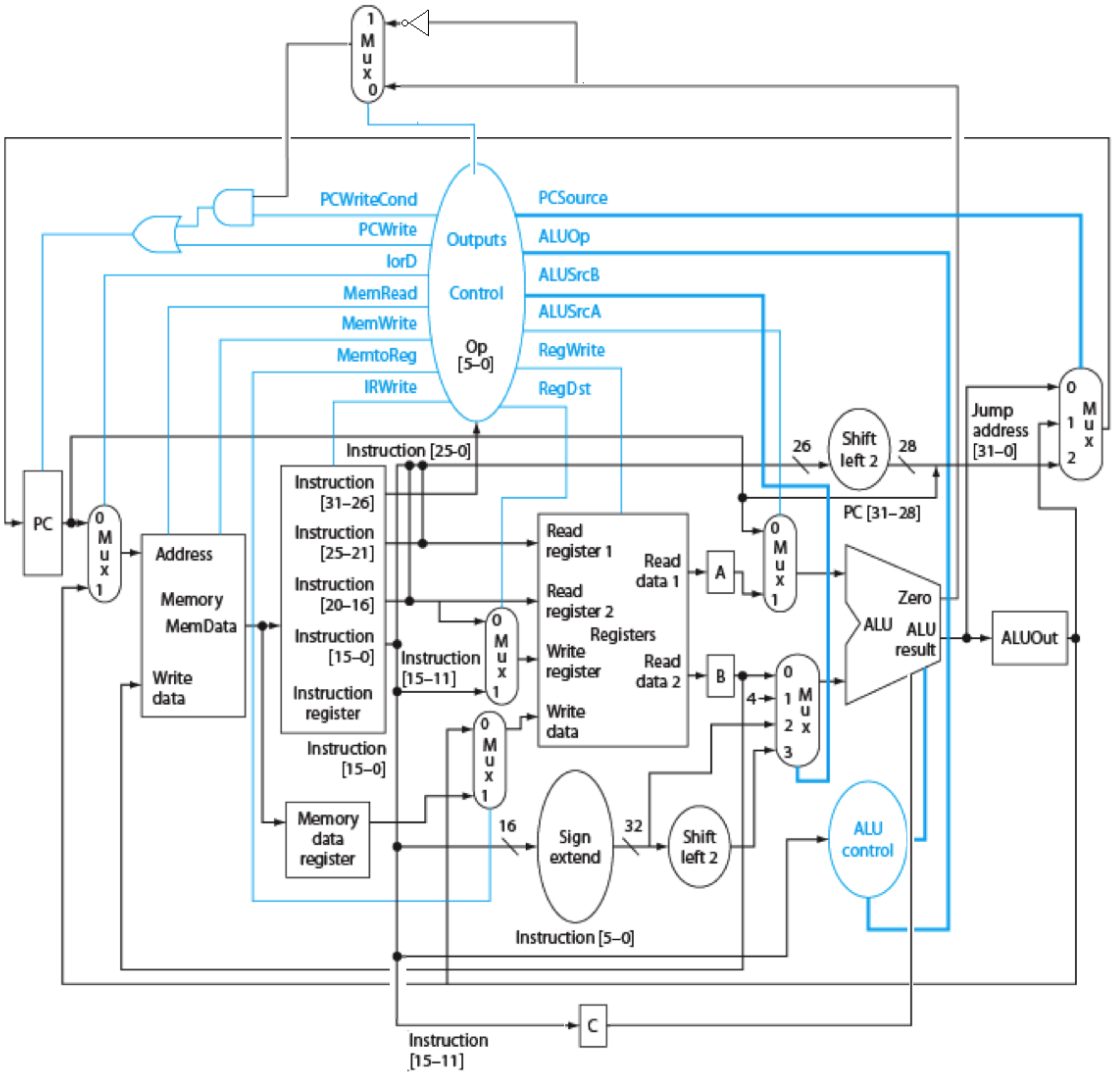
MIPS Multicycle Datapath test. 8
- Add Instruction, add $t3, $t2, $s2 8
- Branch beq $t5, $s2, 600 9
- Jump Instruction, j 700. 11
- Load Instruction, lw $s3, 100($t2) 12
- Store Instruction, sw $s4, 200($t5). 13
- Nand Instruction. 14
- SLT instruction, slt $t6, $s1, $t2. 15
- Or instruction. 17

Appendix 1(Test Module Images) 19

Appendix 2(Module Codes) 26

Introduction.

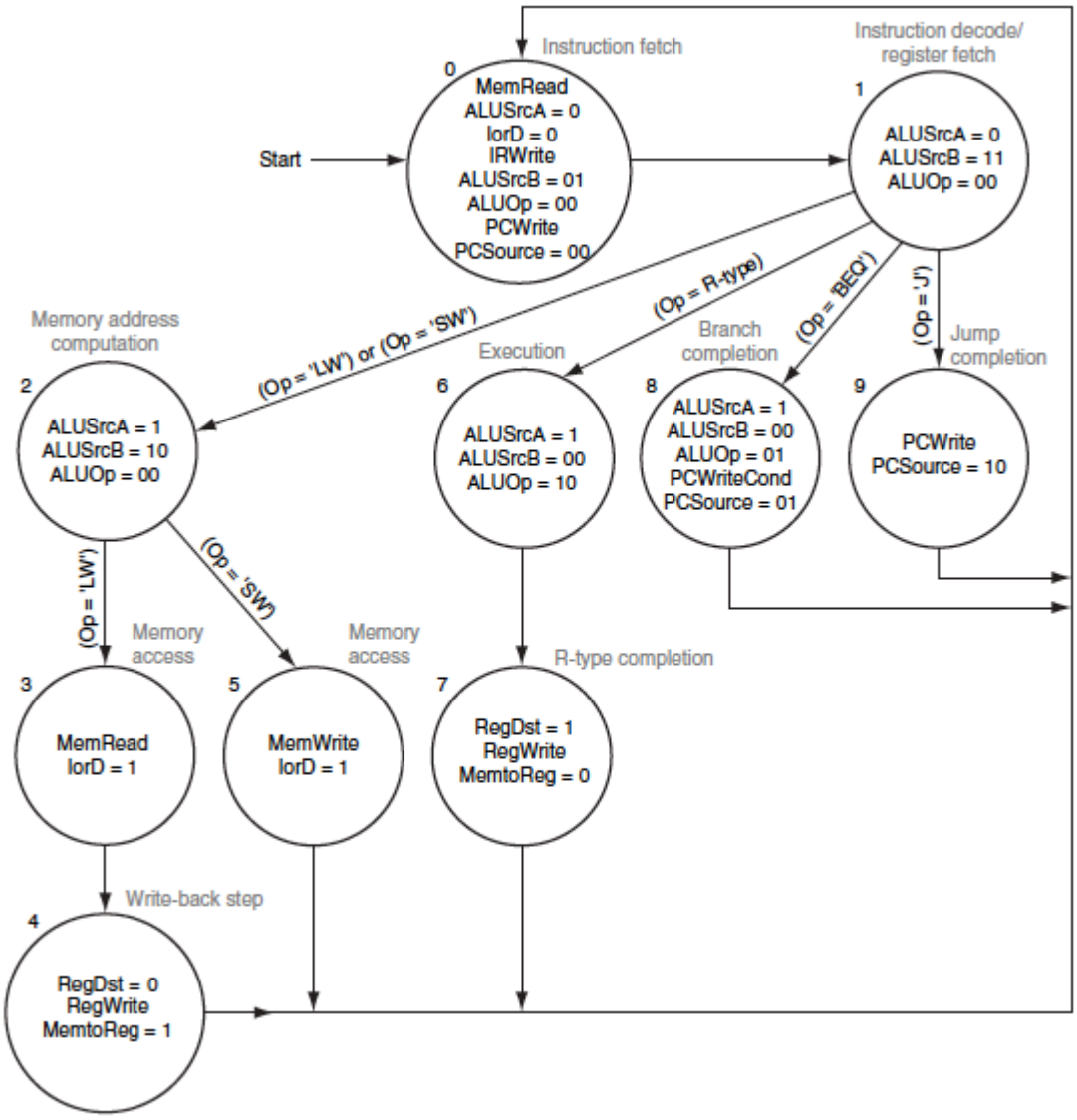
This report will explain the implementation and behavior of a sequential MIPS multicycle datapath, as the one shown in the following image.



As it can be noticed the MIPS model implemented has a few modifications from the original MIPS in order to perform the instructions required.

This MIPS multicycle datapath works sequentially; the next instruction is being done once the previous one has been made, load takes 5 cycles, store 4, R-Type 4, Jump and Branch 3 cycles.

This model have been implemented according to the next schematic of the final state machine that controls the Control Unit.



Implementation.

This MIPS multicycle datapath is performed by multiple modules, that will be specifically described, also with the test images referred to the appendix 1, and to the vhdl code in the appendix 2.

- Program Counter.

The register that contains the address of memory of the instruction. With the inputs, writeEnable (Controls when the address is saved) and addrInput (the input address), and the outputs addrOutput (the output address).

- Memory

The memory contains the instructions set and also the data that is going to be used in the instructions. With the inputs, addressIn (addres to be read or written), data3 (data input), MemRead (control signal of read), MemWrite (control signal of write) and the output data1 (output data).

- Instruction Register

The registers that saves the value of the instruction and decodes it. Inputs, IRWrite (control signal to write), instrucInput (data input), and outputs opCode (Opcode), regRs, regRt, regRd (registers Rs, Rt and Rd), imm (Immediate), jumpAddr (26 bits of jumps address) and funcCode (function code).

- Registers

The 32 registers with the instructions will operate with. Inputs address1(register Rs), address2(register Rt), address3 (register to be written), data3 (Input data register), RegWrite (control signal to write in register) outputs data1(output Rs), data2(output Rt).

- ALU

The ALU performs the operations with the data required in the instruction, addition, substraction, and, shif logical left and nand, set less than. Inputs RegA (data input), RegB (data input), RegC (modification made from the original in order to perform the sll operation, with this input the shamt value is introduced), Oper (operation code that defines the operation to be made) and output Result (output data), Zero (used for bne and beq that when the subtraction is equal to 0, take the value of '1').

- Bits5Reg

A 5-Bits Flip-Flop register to save the value of the shamt, to be used in the ALU in the sll instruction, in the schematic figure the register C. Inputs data5 (5 bits input), clk (clk that actives the flip-flop with rising edge) and output q5 (5 bits output data).

- BitsReg

A 32-bits Flip-Flop register, used in the modules ALUOut, Memory Data Register, Register A and Register B. Inputs data (32 bits input), clk (clk that actives the flip-flop with rising edge) and output q (32 bits output data).

- ClockBox

The clock implemented to control the Control Unit and the registers. InOutput clk.

- Gate

A 32-Bit and gate used to perform the final control bit in the writePC, as shown in the schematic figure, takes Zero(beq) or $\overline{Zero}$(bnq) to do an and with the Signal PCWriteCond that is active with these two signals. Inputs a,b (1-bits data input) and output y (1-bit data ouput).

- Mux2_1

Multiplexor of 1 bit input data, used to choose between Zero(beq) and $\overline{Zero}$(bnq) with the control signal Branchsignal which the Control Unit will active or not if the corresponding opcode is beq or bnq. Inputs e0,e1 (input data 1-bit), C (control signal) and outputs s (output data 1-bit).

- Or2

A 1-bit or gate to perform finally the value of FinalPC (PCwrite intput PC), doing an or between PCwrite and the and of Zero(beq) and $\overline{Zero}$(bnq), PCWriteCond. Inputs a,b (1-bits data input) and output y (1-bit data ouput).

- Mux2_5b

Multiplexor of 5 bit input data, used to choose between Rt (load instruction) and $Rd(R-Type\ intruction)$ with the control signal RegDst. Inputs e0,e1 (input data 5-bit), C (control signal) and outputs s (output data 5-bit).

- Mux2

Multiplexor of 32 bit input data, used in the multiplexors:
-Pc-Memory (to chose between PC[seq] and memory address[load,sotre]) control signal IorD
-MemoryDataReg,ALUOut-Registers(to wirte in the registers, depending if it is R-Type or Load) control signal MemtoReg
-PC,A-ALU (to do PC+4 or any operation with register A) control signal ALUSrcA.

Inputs e0,e1 (input data 32-bit), C (control signal) and outputs s (output data 32-bit).

- Mux4

Multiplexor of 32 bits data and 4 inputs and 2-bits control signal:
-Input of ALU to choose between RegB(R-type), 4(PC+4), SignExtended (load,store) and ShiftLeft2 (Branch Address) with the control signal ALUSrcB
-Input PC, chooses between Jump address, Branch address and PC+4.

Inputs e0,e1, e2, e3 (input data 32-bit), C (control signal 2-bits) and outputs s (output data 32-bit).

- New-Current

A 2-bit Flip-Flop that control the transition between the current and the next state, controls the final state machine that is performed with the ALU Control Unit. Inputs data (2 bits input), clk (clk that actives the flip-flop with rising edge) and output q (2 bits output data).

- ShiftLeft2

This module does a left logical shift of 2 bits of an input data of 32 bits. Input dataIn (32 bits) and output dataOut (32 bits).

- ShiftLeft22

This module does a left logical shift of 2 bits of an input data of 26 bits, with an output of 28 for the Jump Address. Input dataIn (26 bits) and output dataOut (28 bits).

- SignExtend

This module extend the sign of an input of 16 bits up to 32 bits. Input dataIn (16 bits) and output dataOut (32 bits).

- CUnit

This is the control unit, the most important part that controls the signals to be activated depending which is the current state and the opcode f the instruction.
Inputs:

- OPC: Opcode of the instruction.
- CS: Current State

Outputs:

- ALUOp: Control signal that tell the ALU what to do.
- ALUSrcA: : Chooses between PC and RegA.
- ALUSrcB: : Chooses between RegB, 4, SignExtend and ShiftLeft2.
- PCSource: Chooses between Jump address, PC+4 and Branch address
- PCWriteCond: Activates with branch.
- PCWrite: Write PC+4 in PC unconditionally.
- IorD: Chooses between PC or memory address.
- MemRead: Allow to read the memory.
- MemWrite: Allow to write in the memory.
- MemtoReg: Chooses between MemoryDataRegister(load) and ALUOut(R-Type)
- IRWrite: Allow to write in the instruction file.

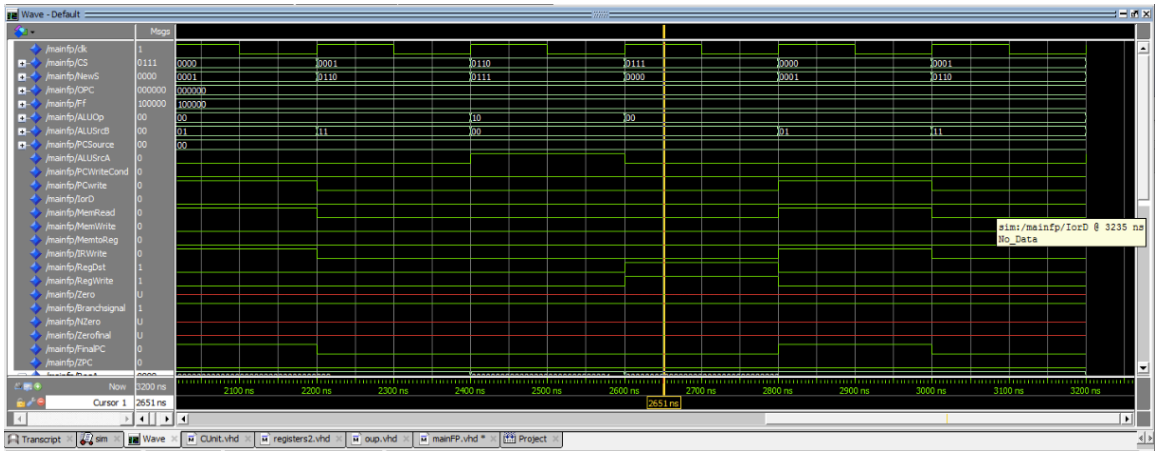
- RegDst: Chooses between Rt(load) and Rd(R-Type).
- RegWrite: Allow to write in the register file.
- NewS: Next State.
- Branchsignal: Chooses between Zero(beq) or $\overline{Zero}$(bnq).

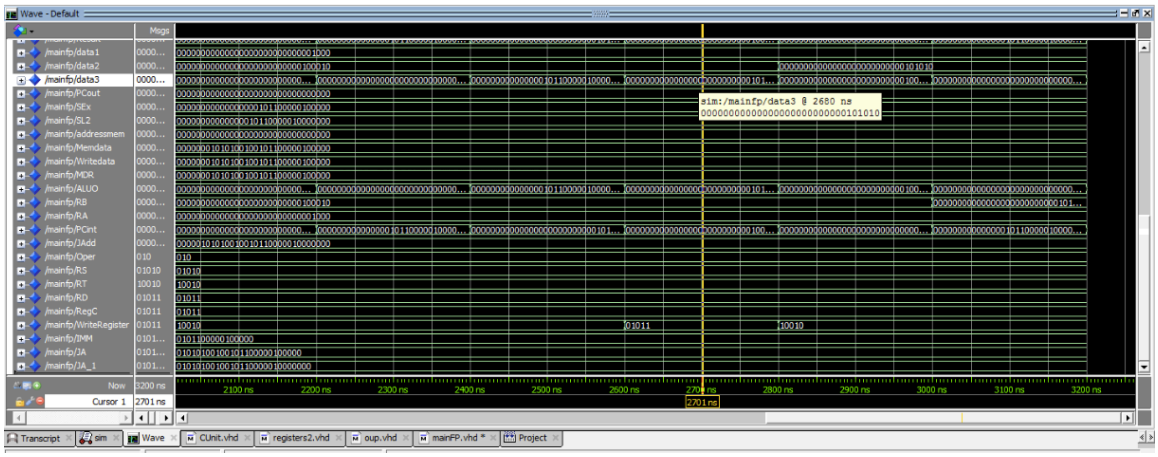
MIPS Multicycle Datapath test.

Implemented in the file MainFP.vhd, the test of various instruction have been made. In this section a test instruction will be done, and shown in a sequence of images that it works as it should be.

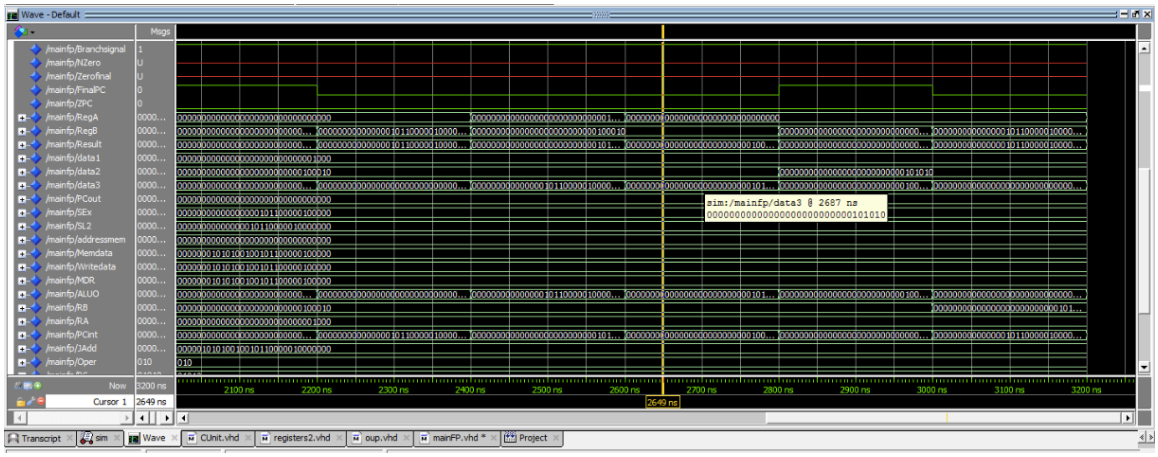
Although the instruction bne, subi and sll have being implemented it have not been proof to work fine. All the other instructions works as it should, proof in the following pictures.

- Add Instruction, add $t3, $t2, $s2

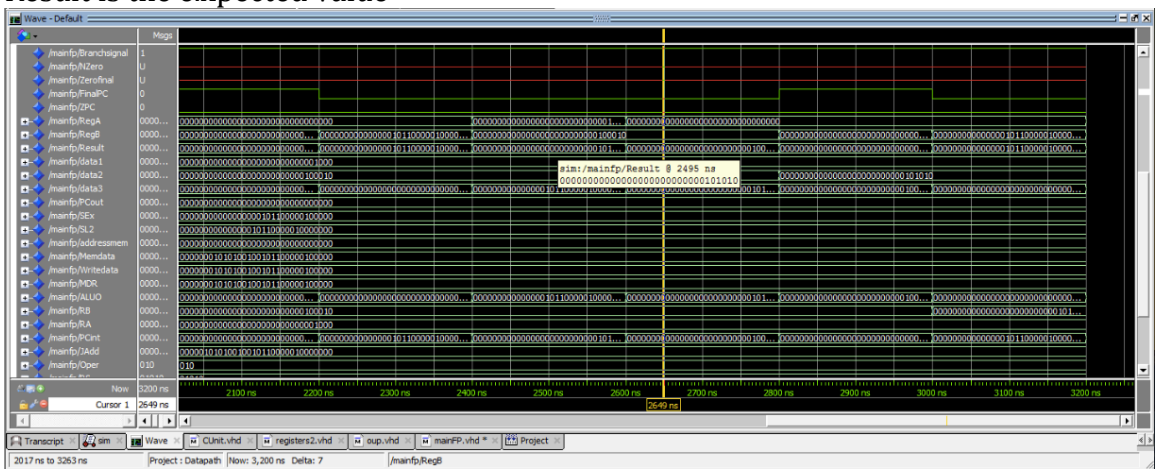




Data 3 is the expected value.

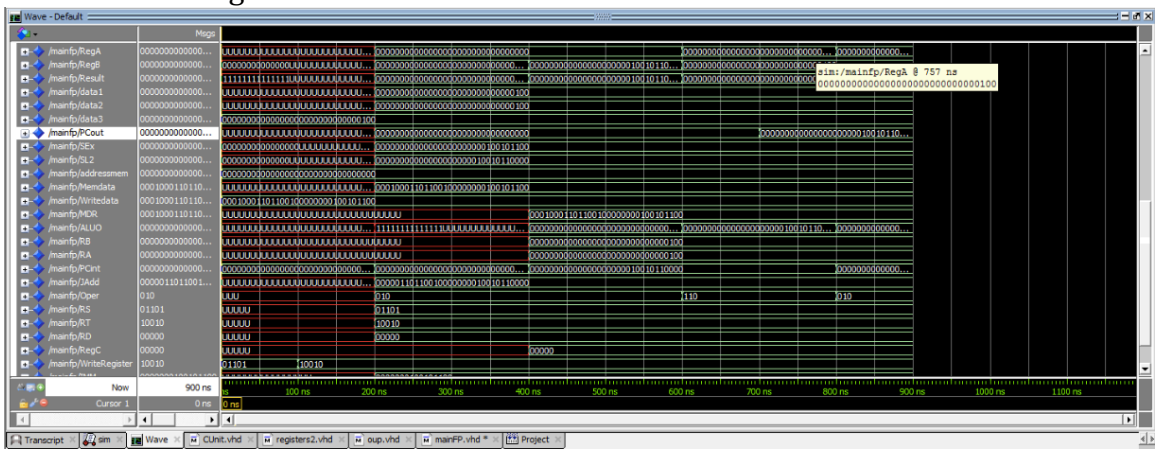


Result is the e.xpected value

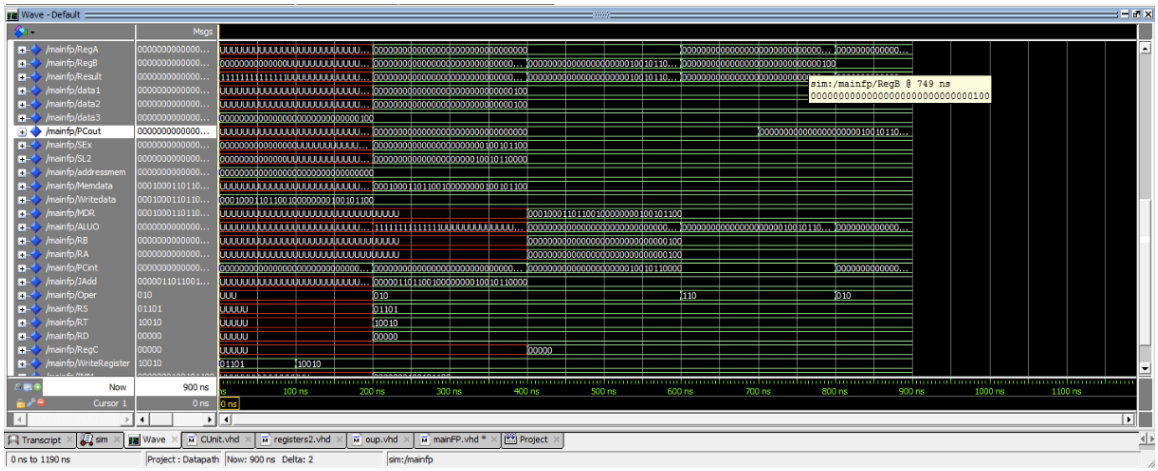


- Branch beq $t5, $s2, 600

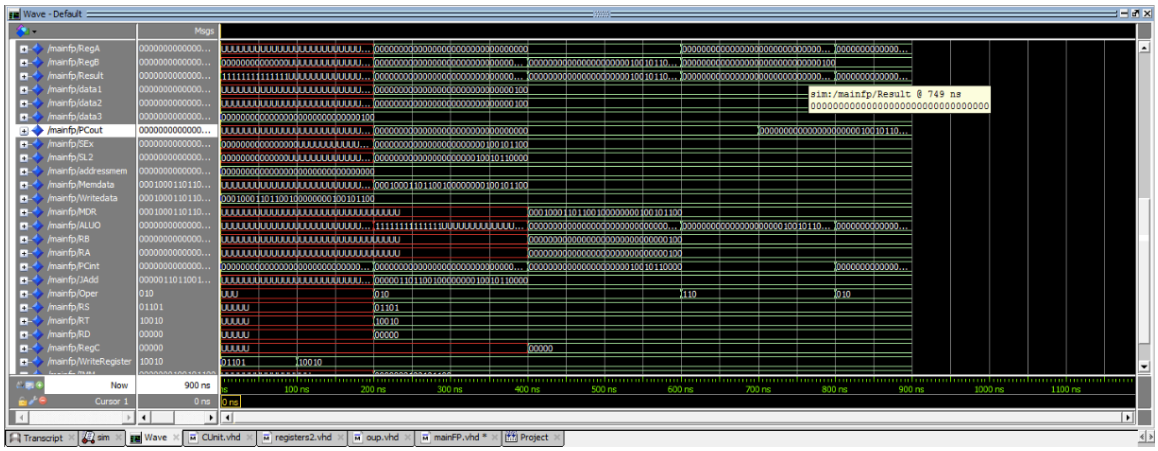
Load value in RegA.



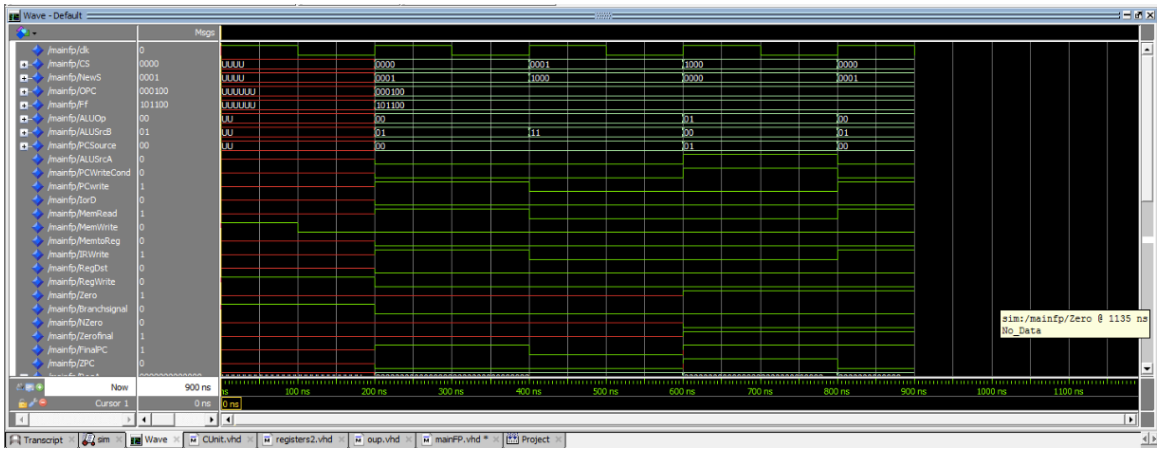
Load value in RegB.



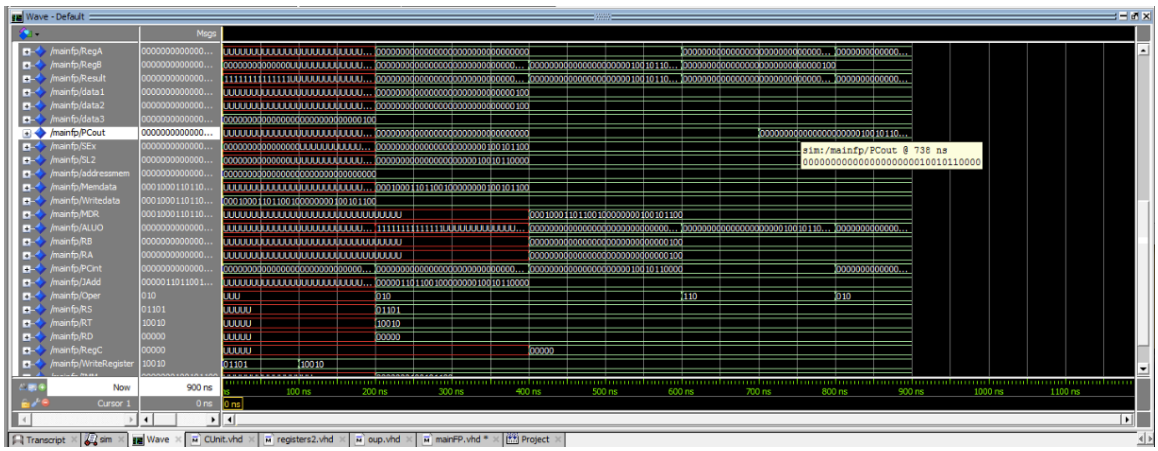
Result is all 0.



Zero is active.

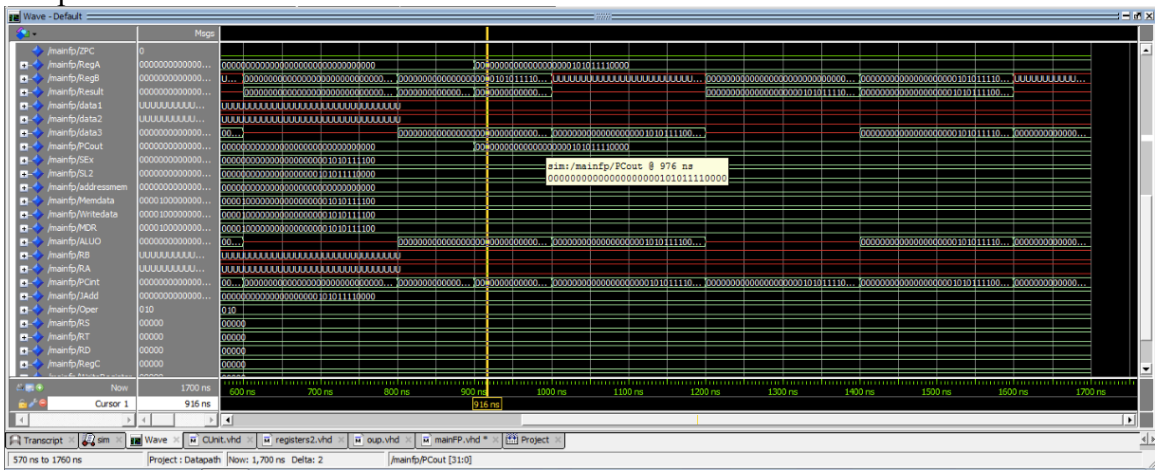


Branch Address

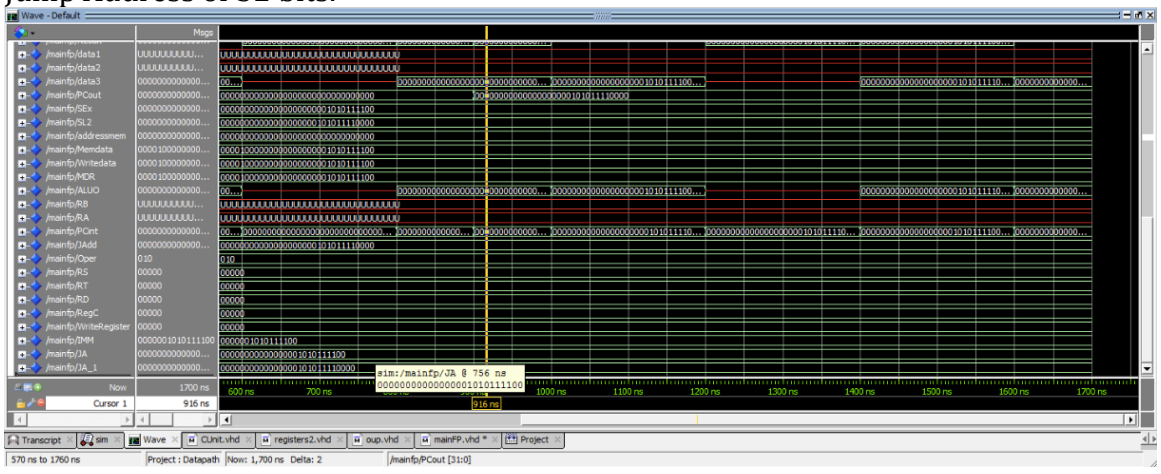


- Jump Instruction, j 700.

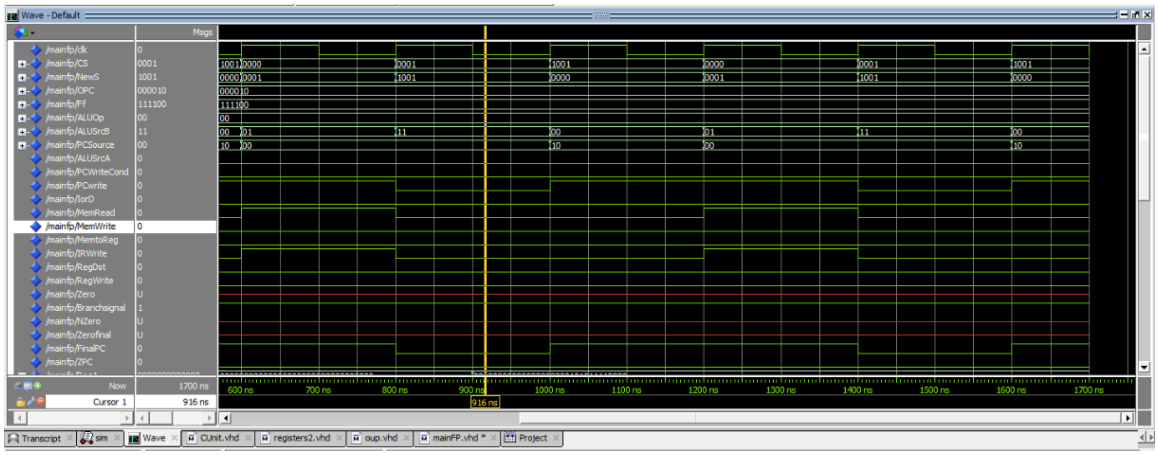
Jump Address 700*4



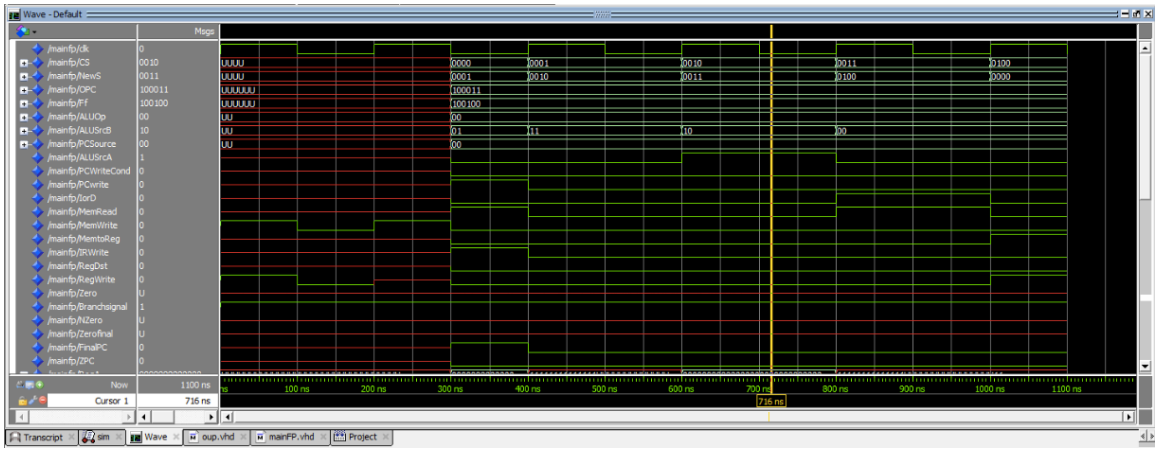
Jump Address of 32 bits.

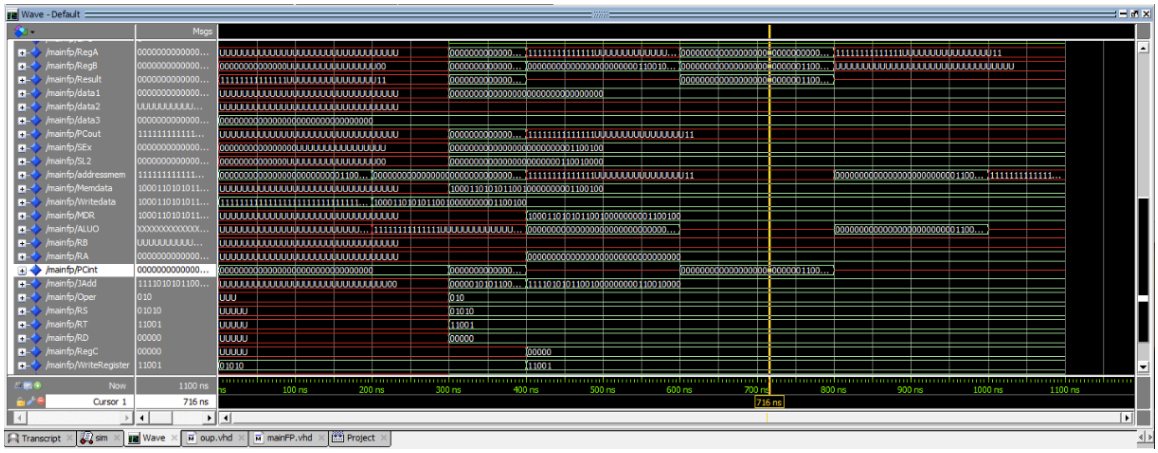


Final PC and PCWrite activates.

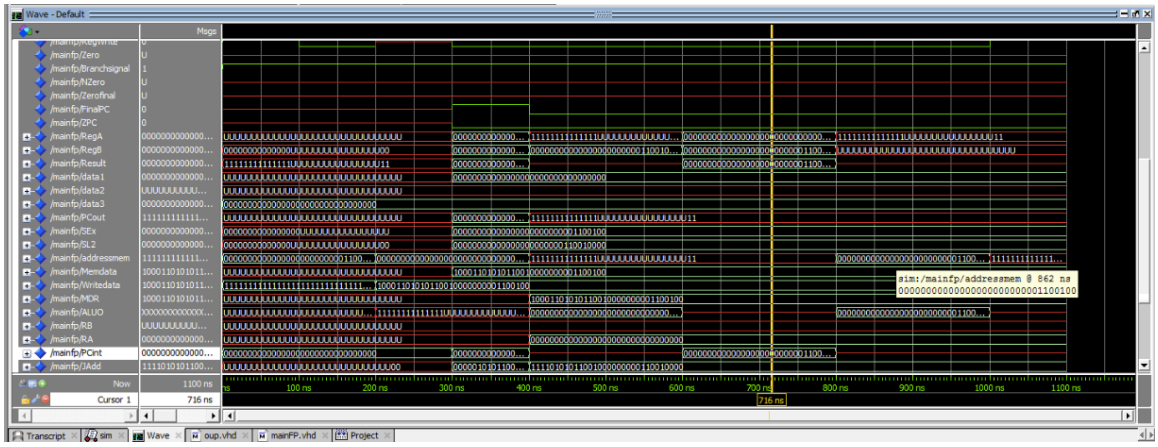


- Load Instruction, lw $s3, 100($t2)

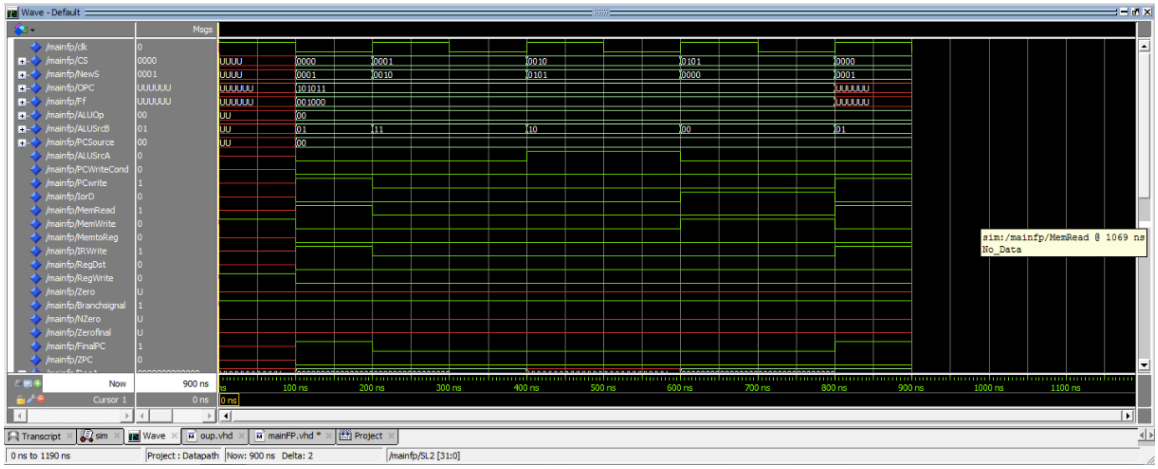


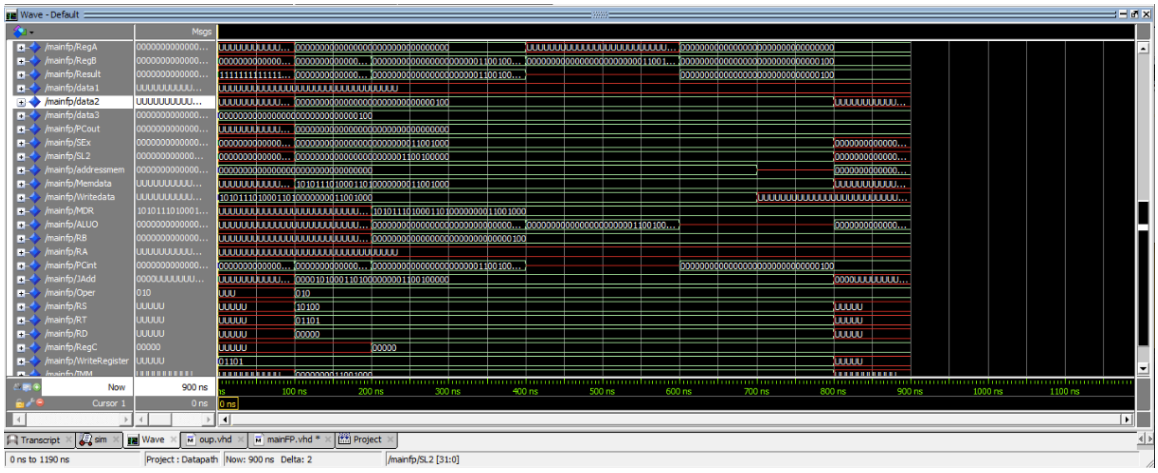


Load in the address 100.

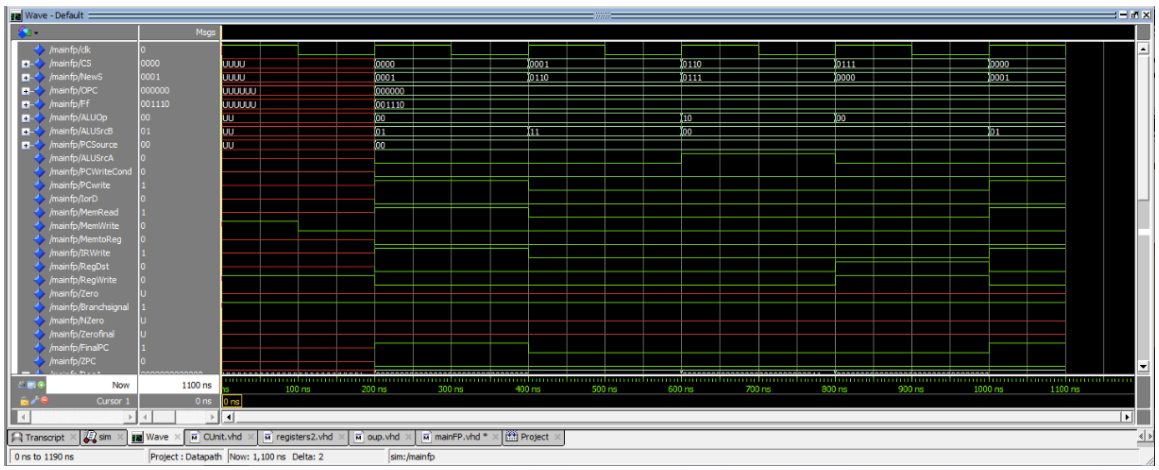


- Store Instruction, sw $s4, 200($t5).

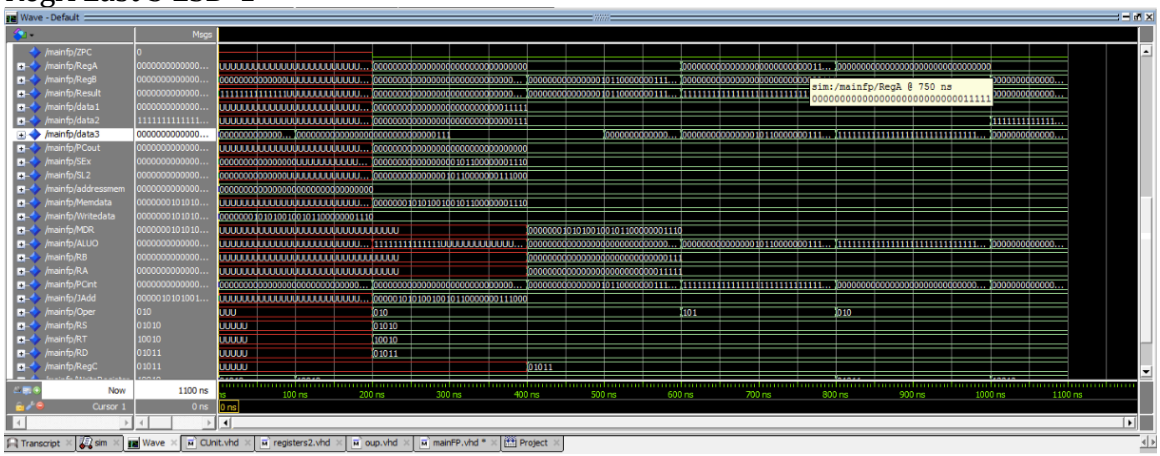




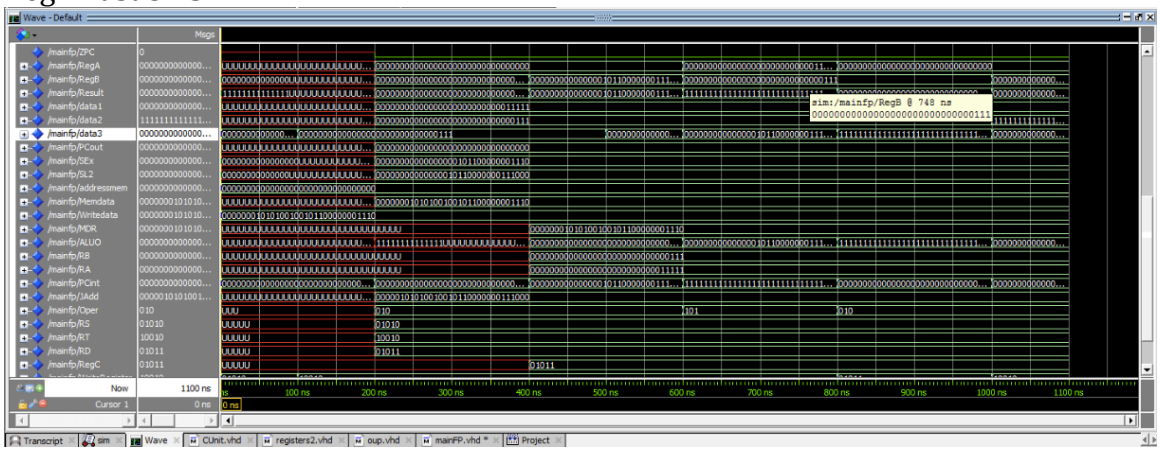
- Nand Instruction.



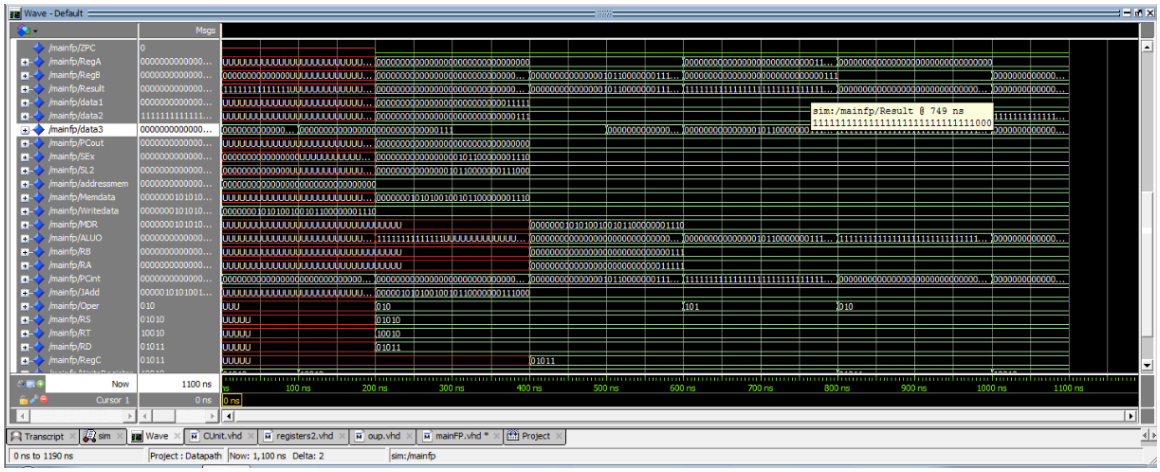
RegA Last 3 LSB ‘1’



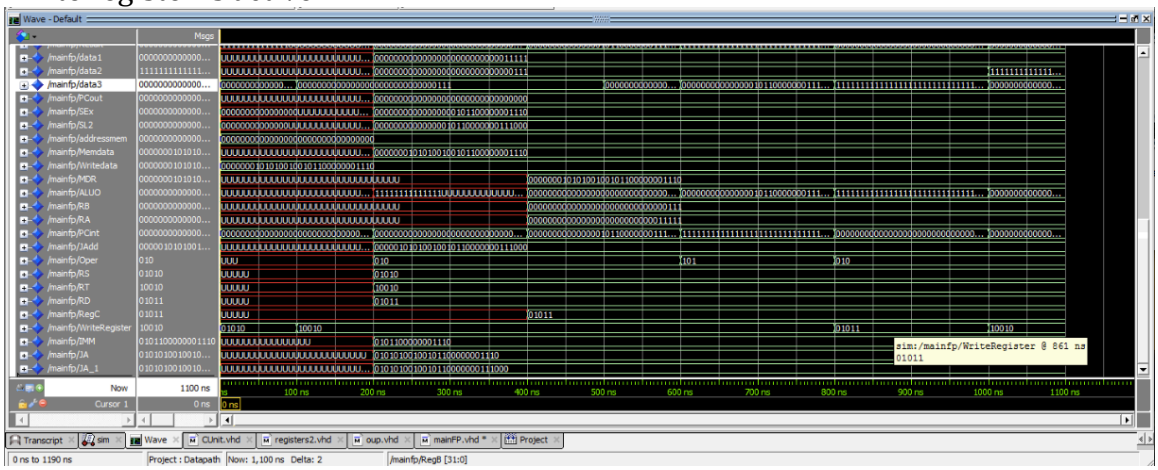
RegB Last 5 LSB ‘1’



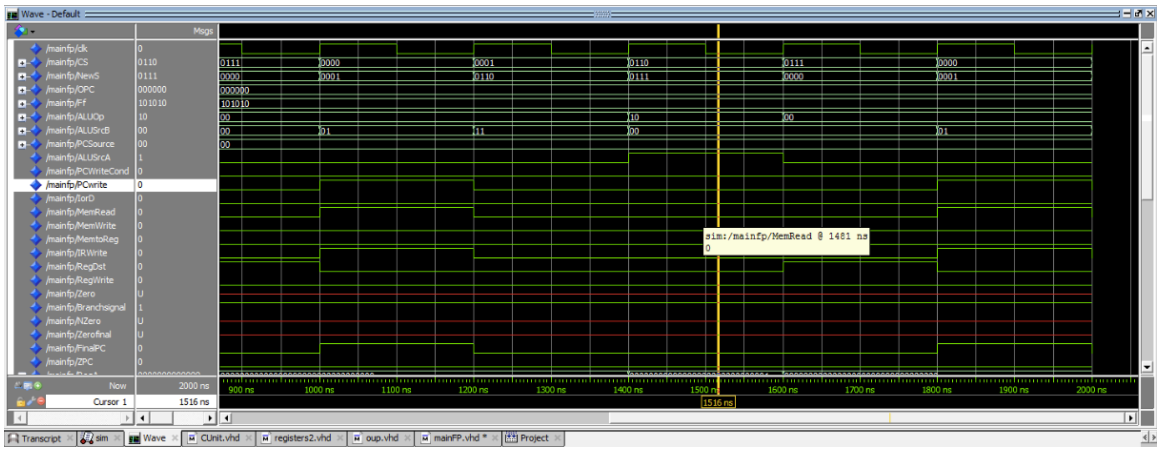
Result All ‘1’ except last 3 LSB.



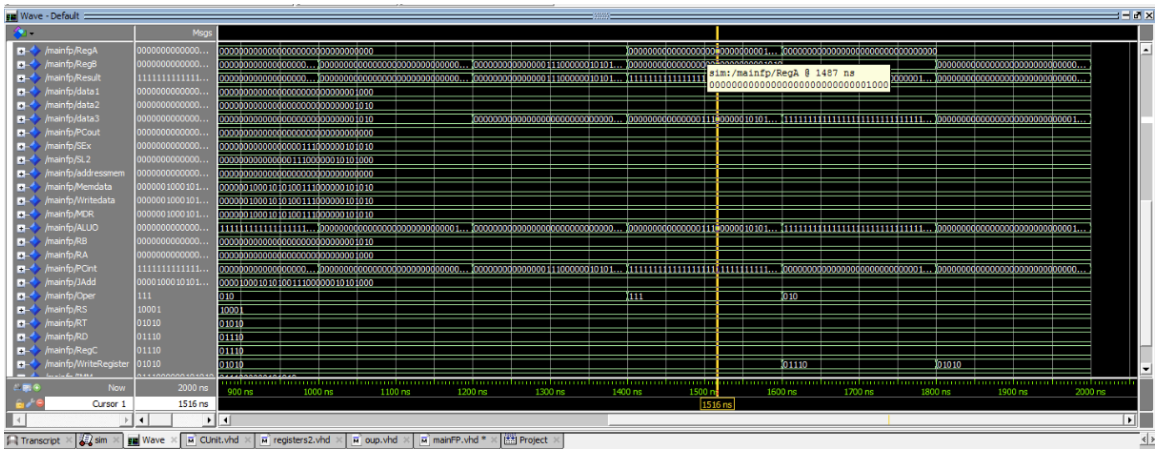
WriteRegister is active



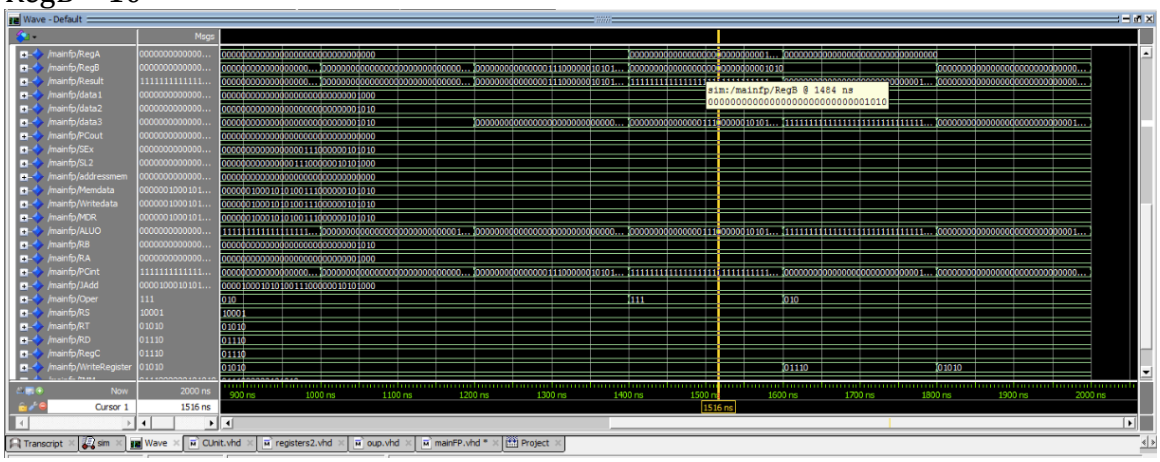
- SLT instruction, slt $t6, $s1, $t2.



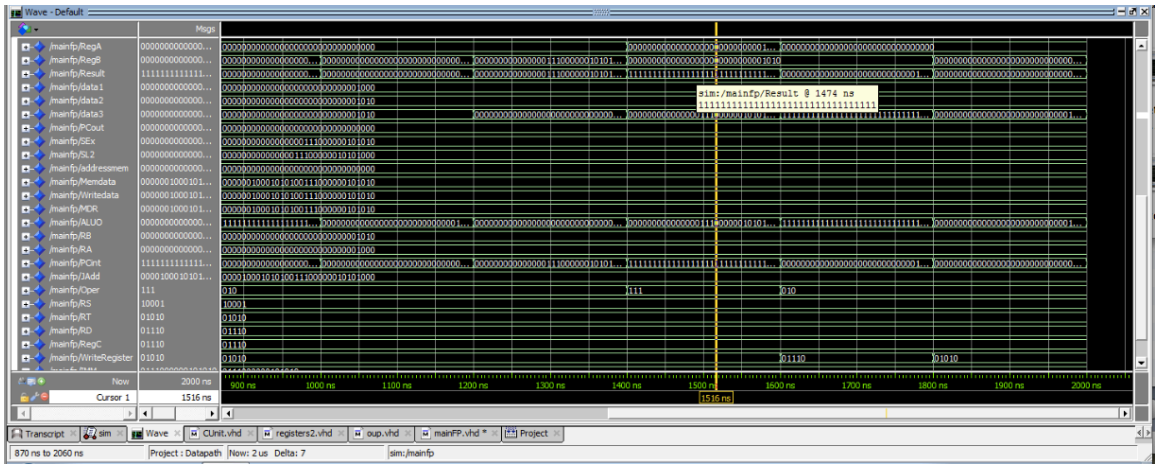
RegA = 8



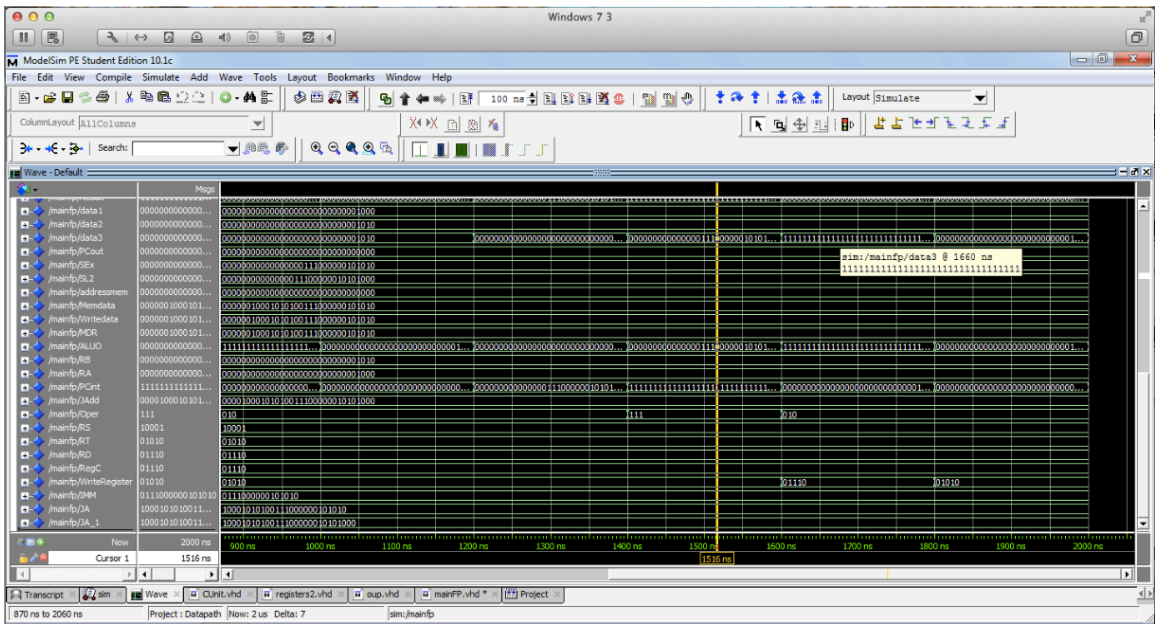
RegB = 10



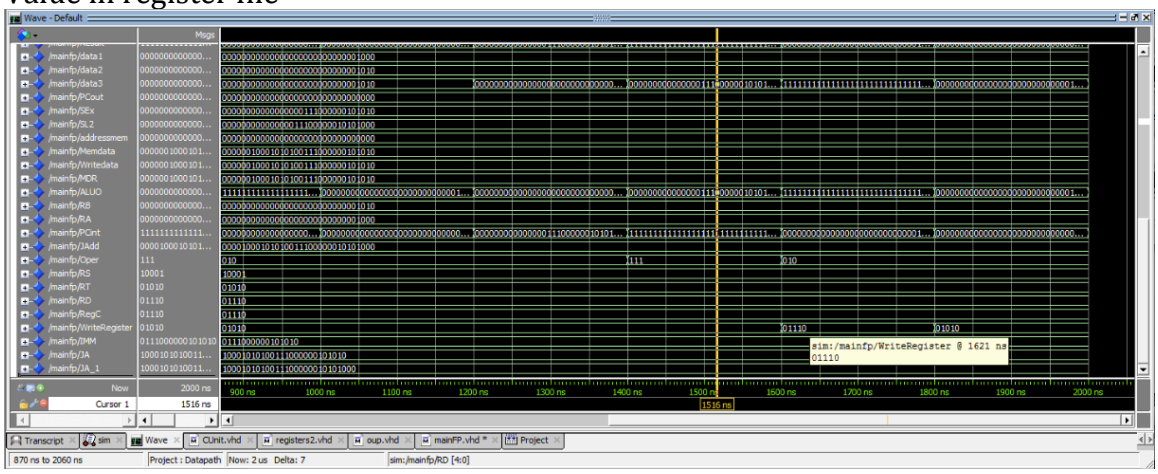
Result all ‘1’ A<B



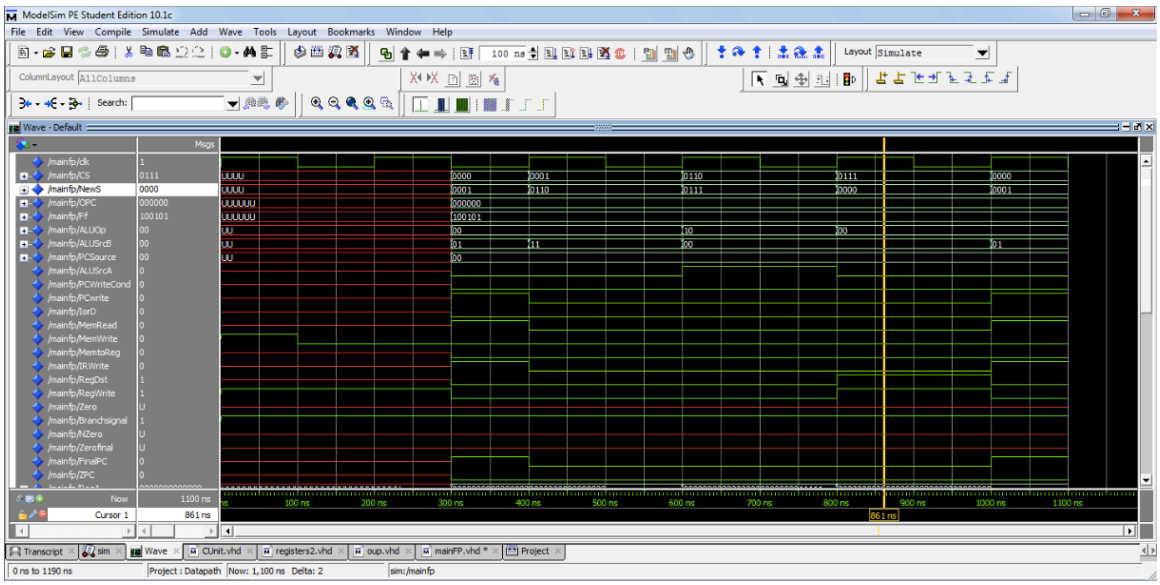
Data3(Input register File).



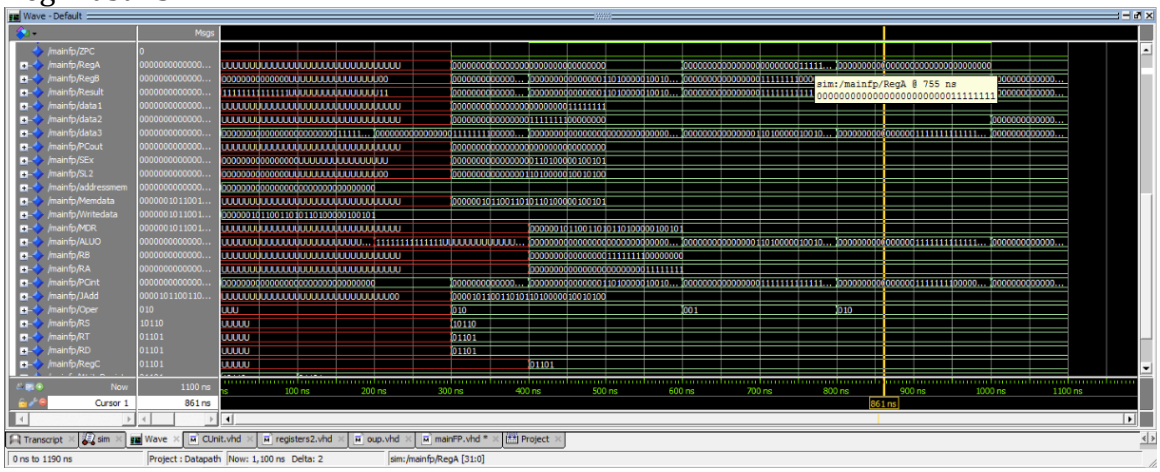
Value in register file



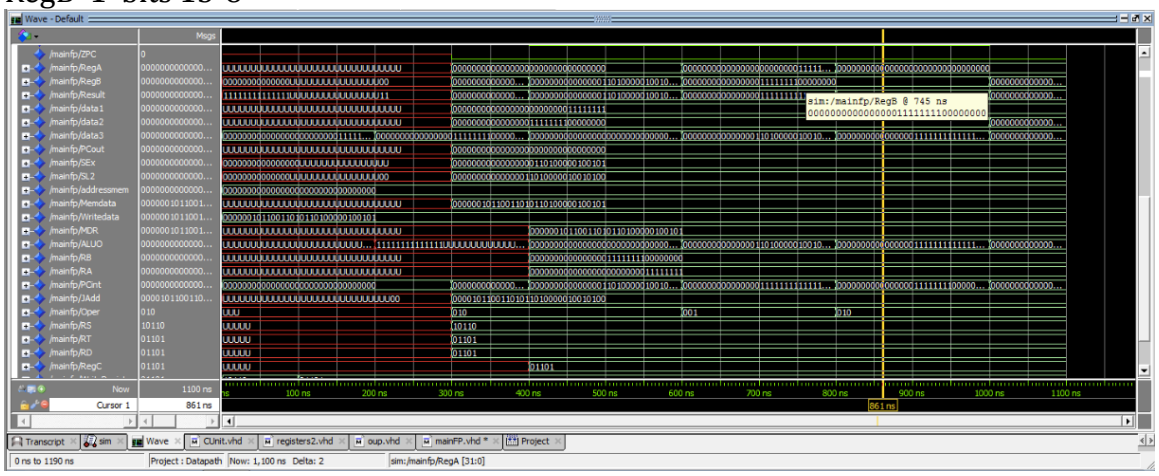
- Or instruction.



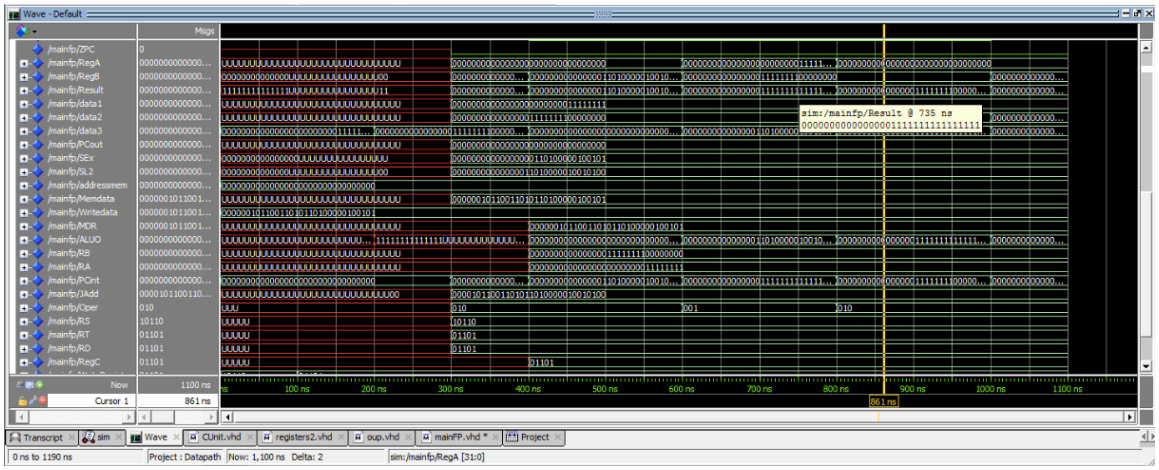
RegA last LSB ‘1’



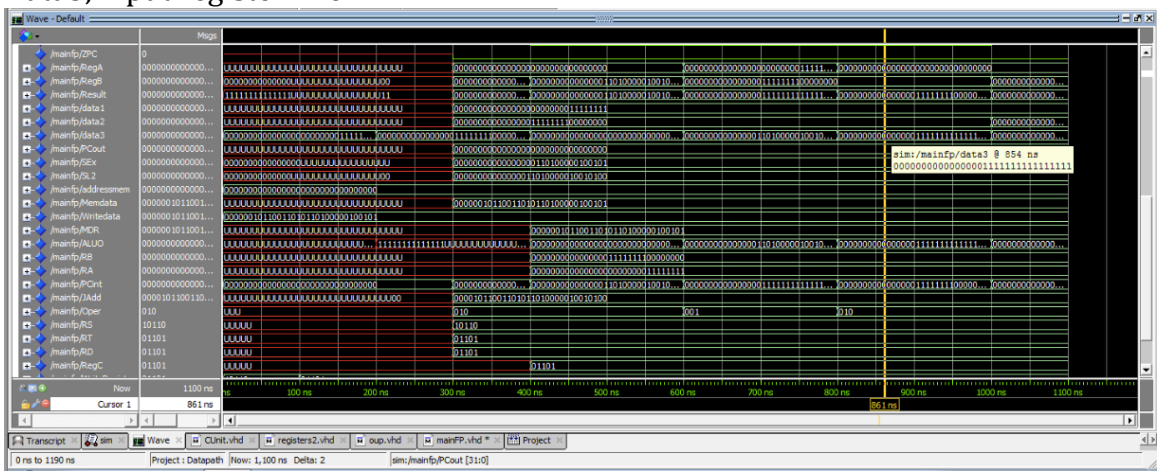
RegB ‘1’ bits 15-8



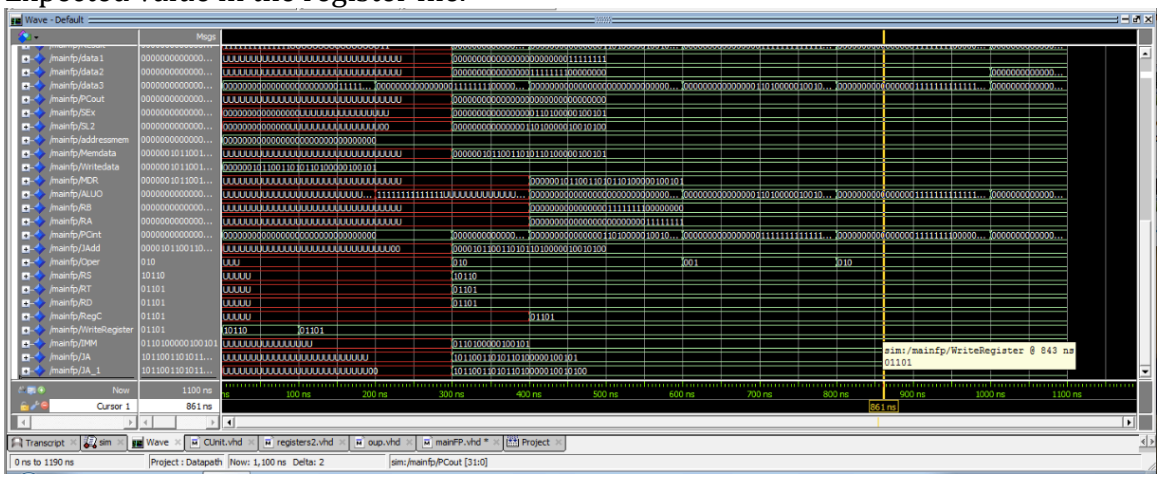
Result expected.



Data3, input register File.

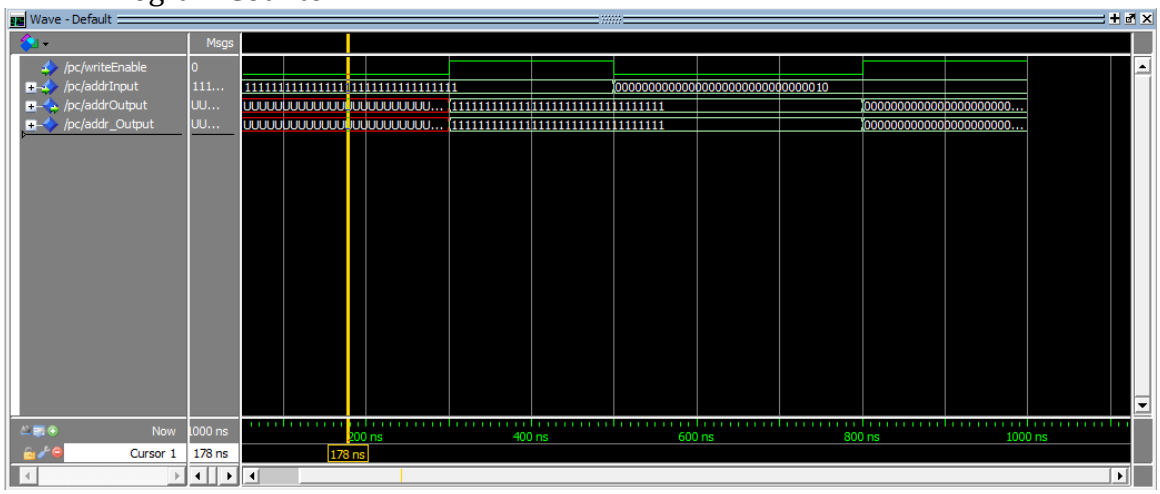


Expected value in the register file.

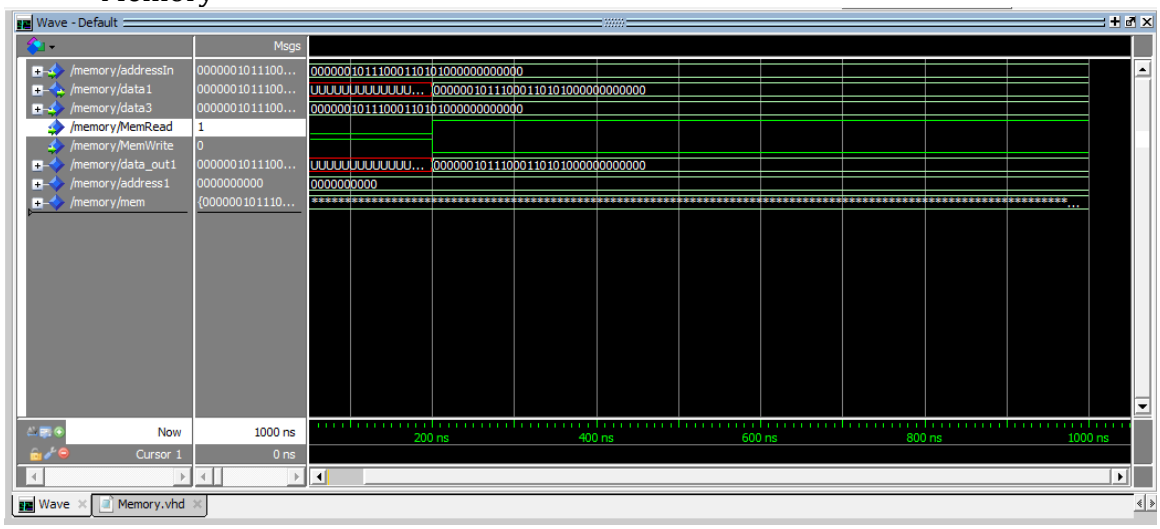


Appendix 1

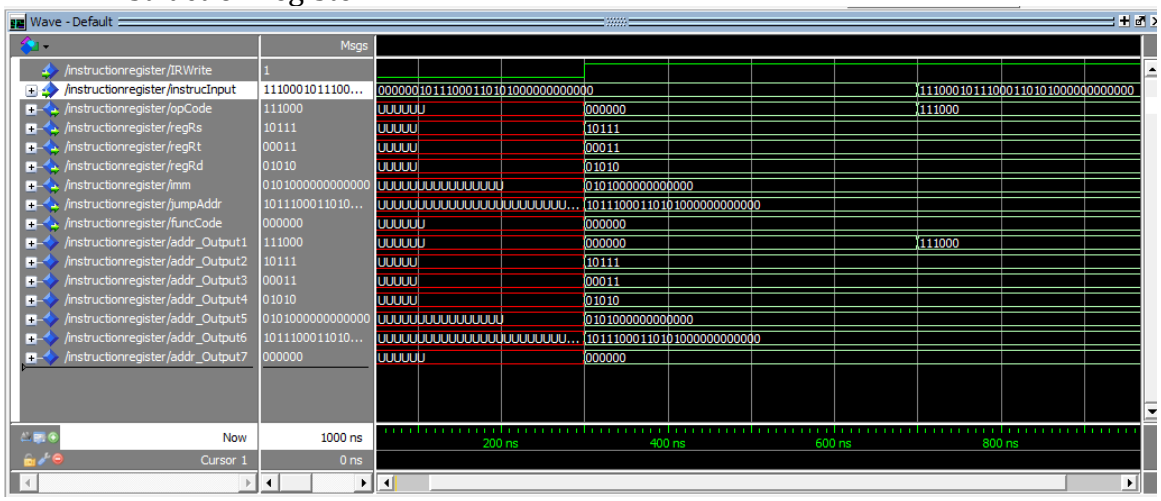
- Program Counter



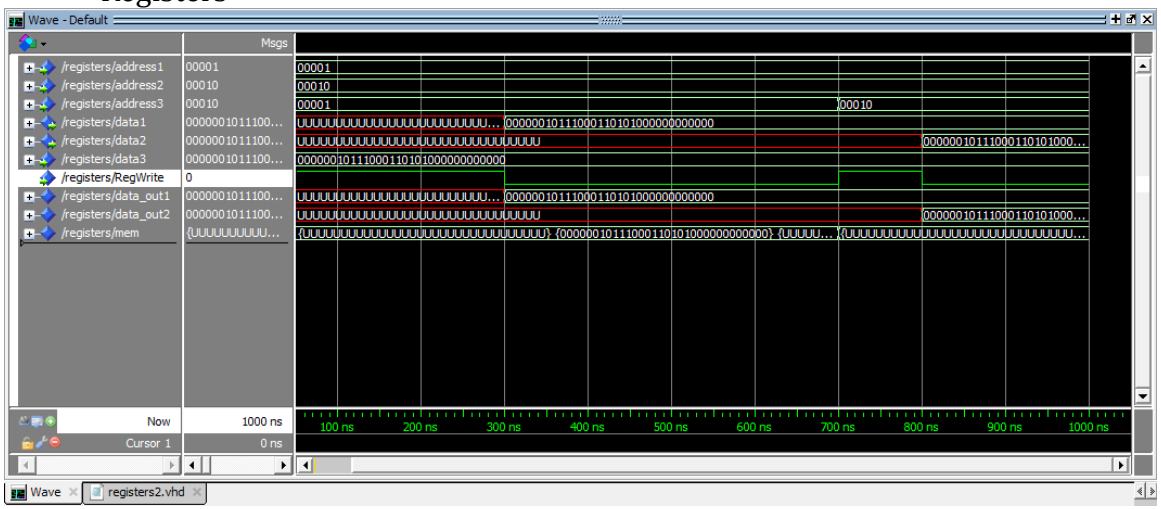
- Memory



- Instruction register

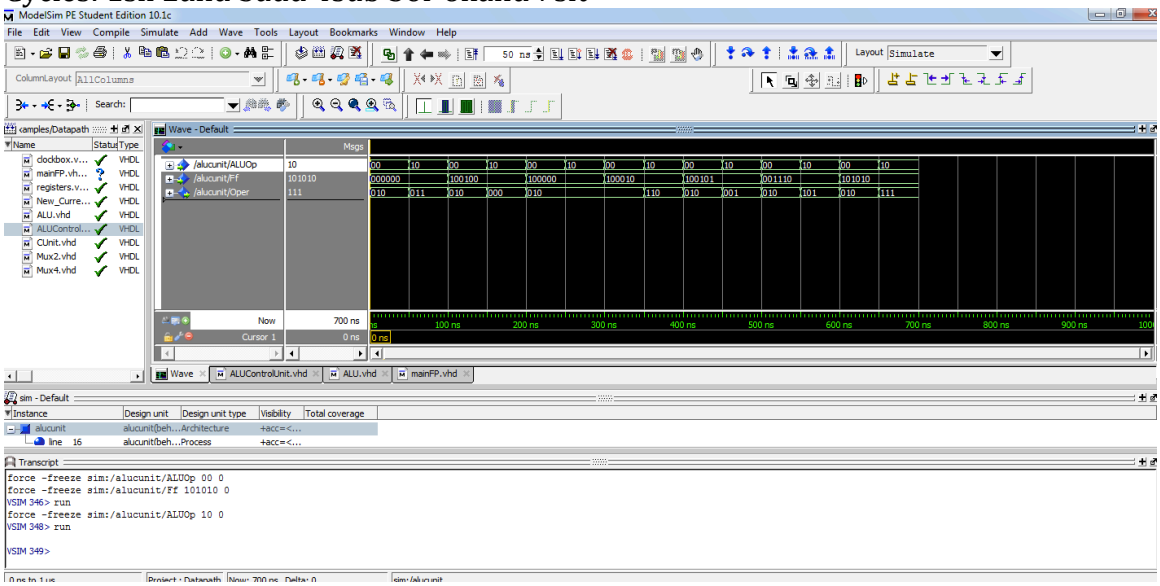


- Registers

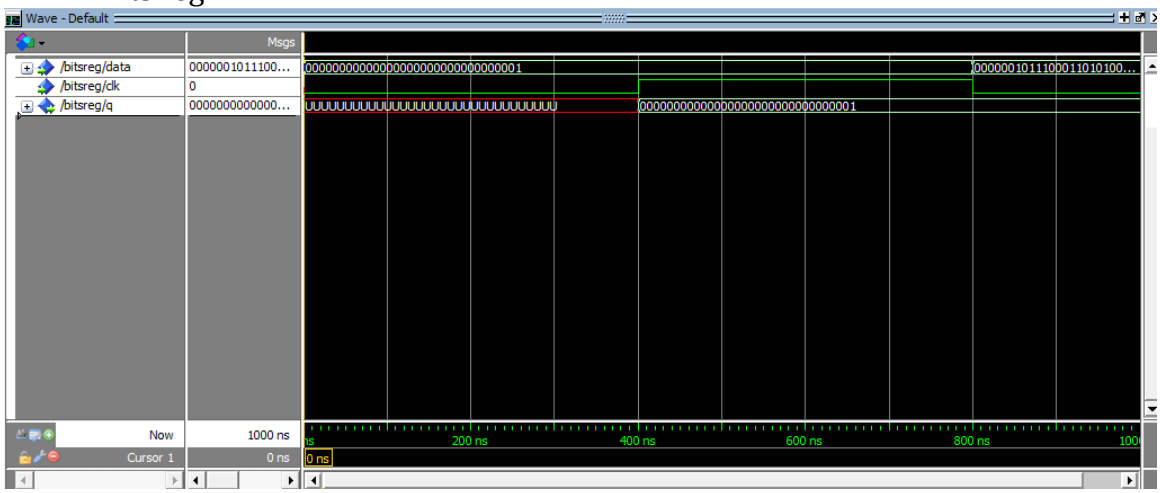


- ALU

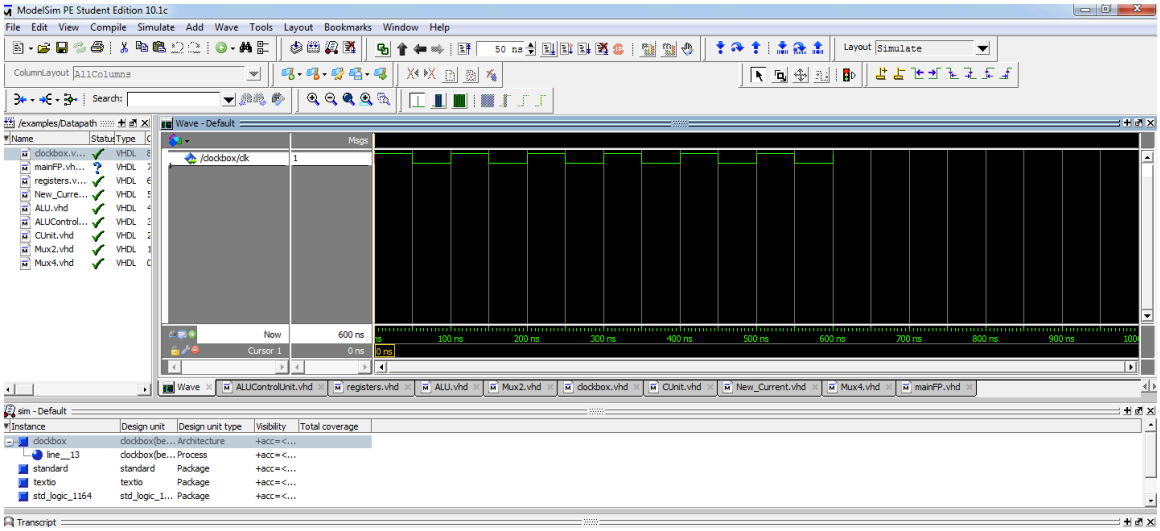
Cycles: 1sll 2and 3add 4sub 5or 6nand 7slt



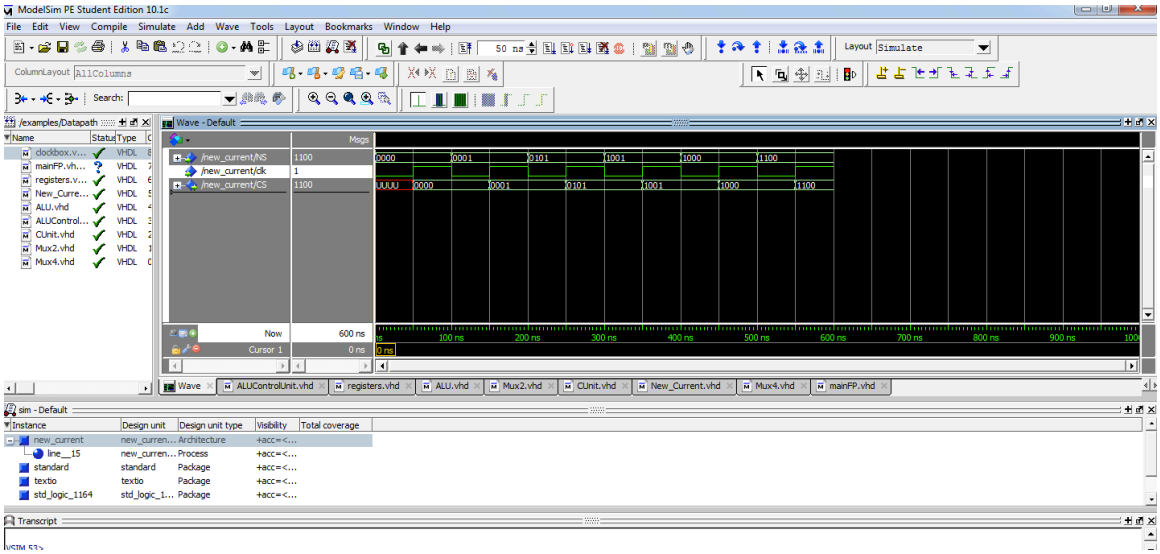
- BitsReg



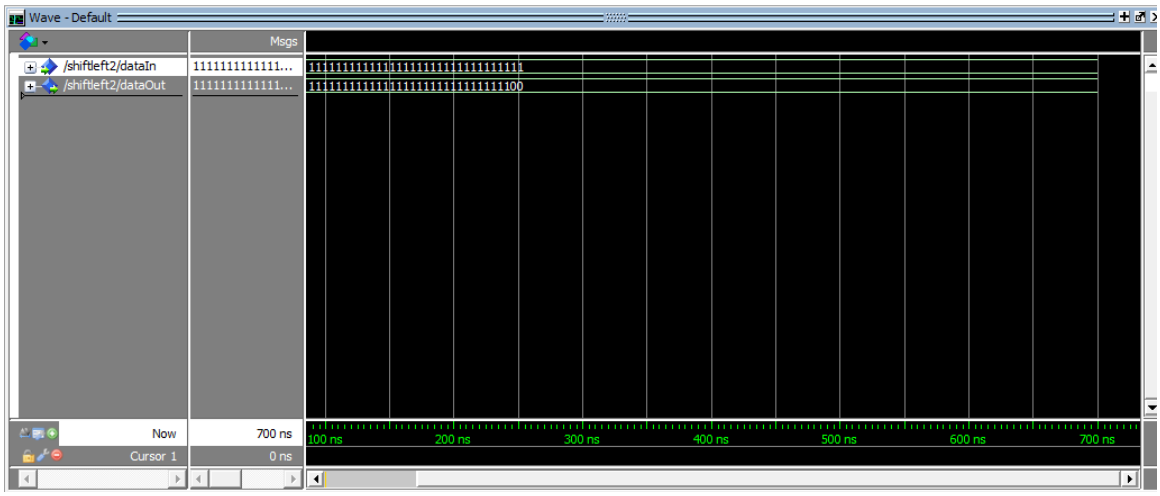
- Clockbox



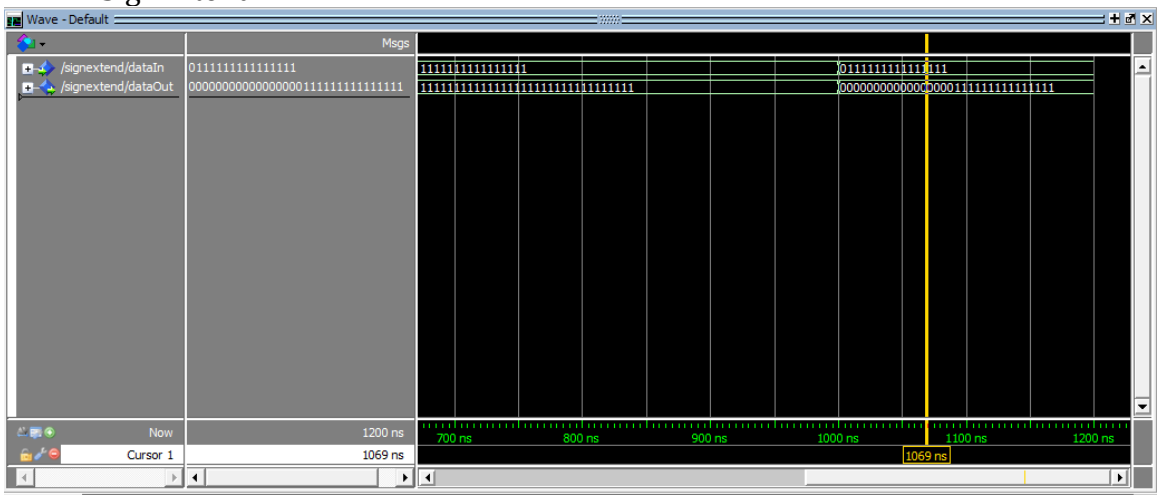
- New-Current



- ShiftLeft2

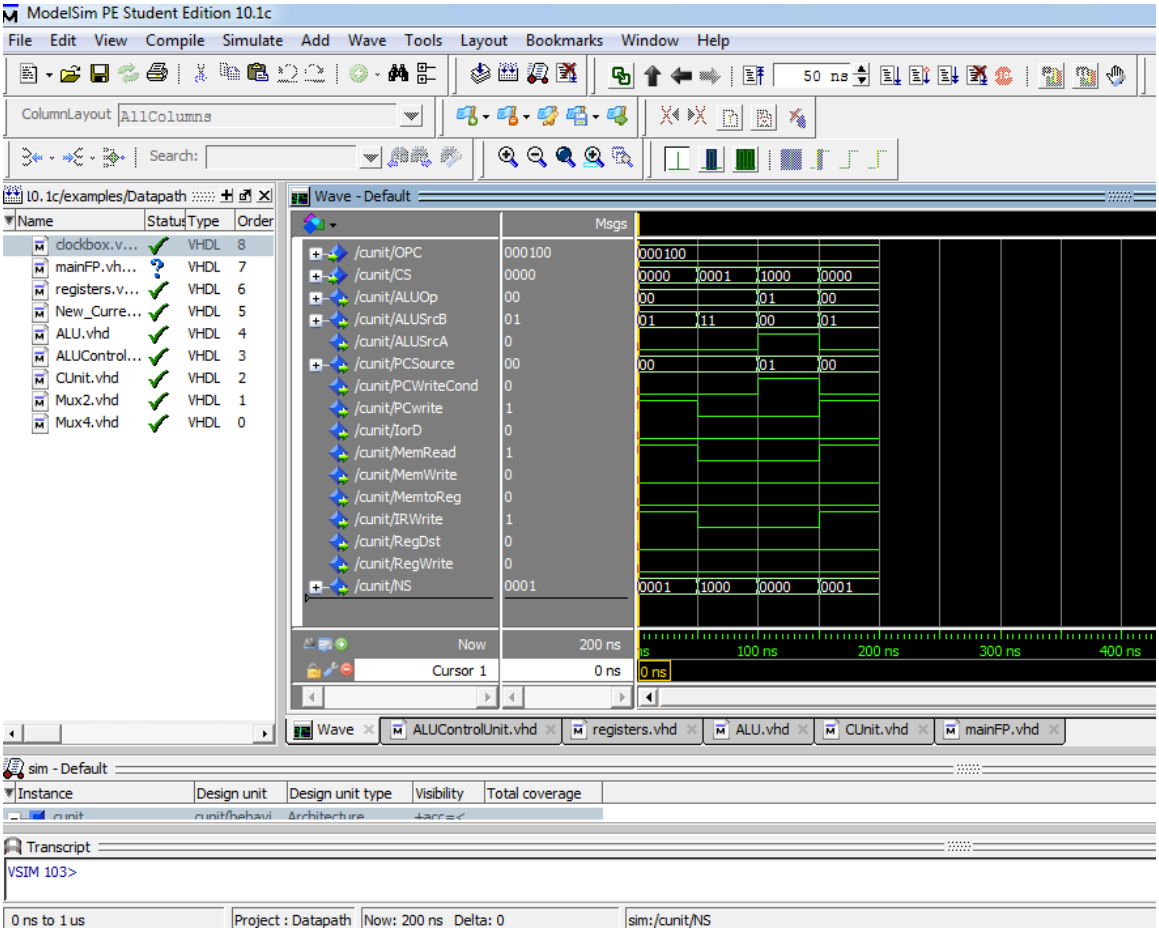


- SignExtend

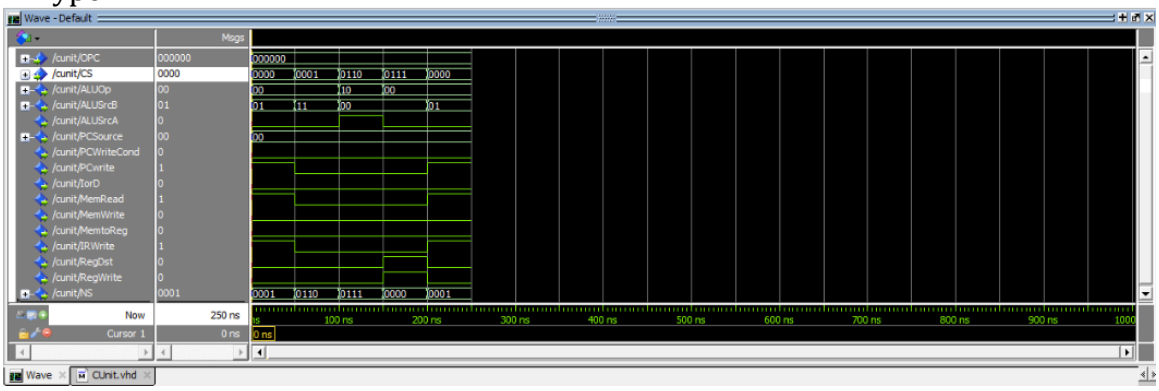


- CUnit

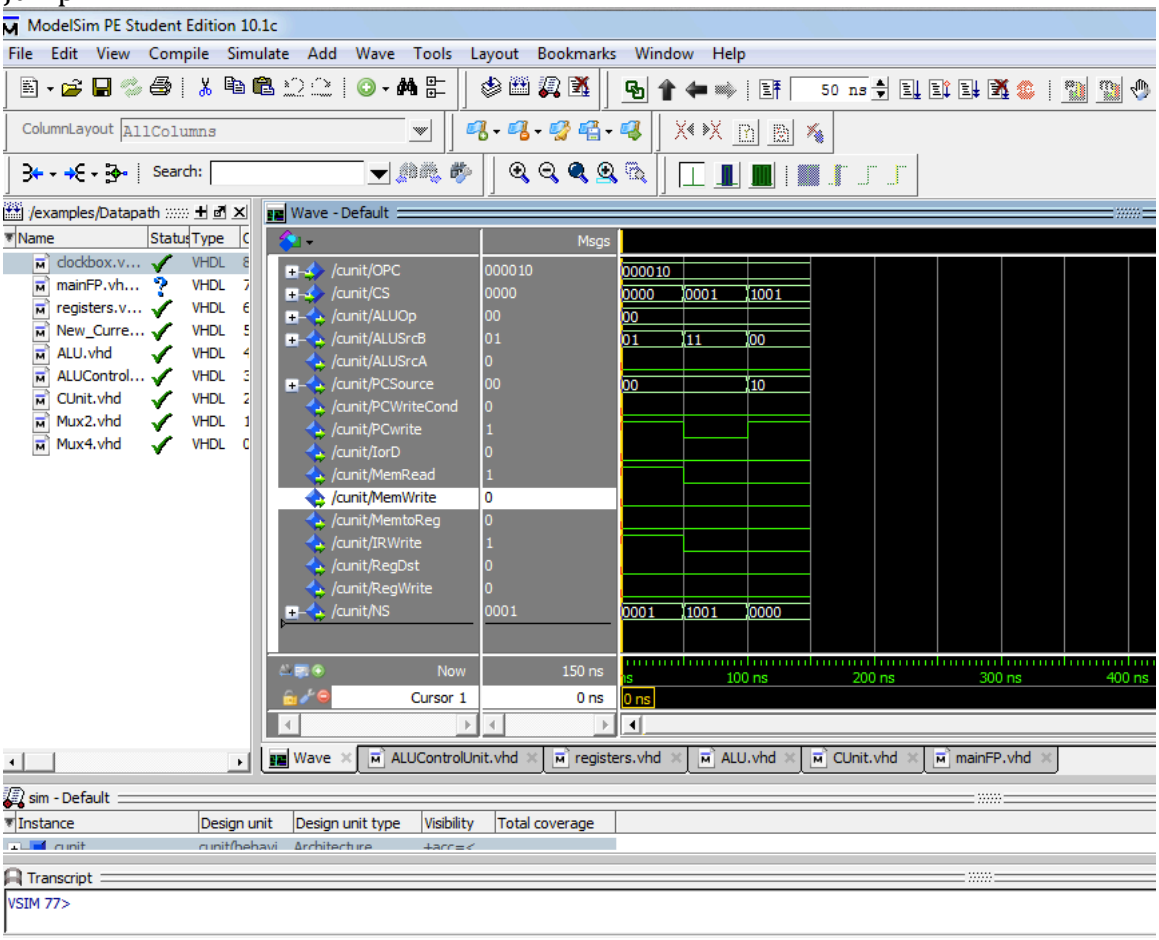
Branch Instruction



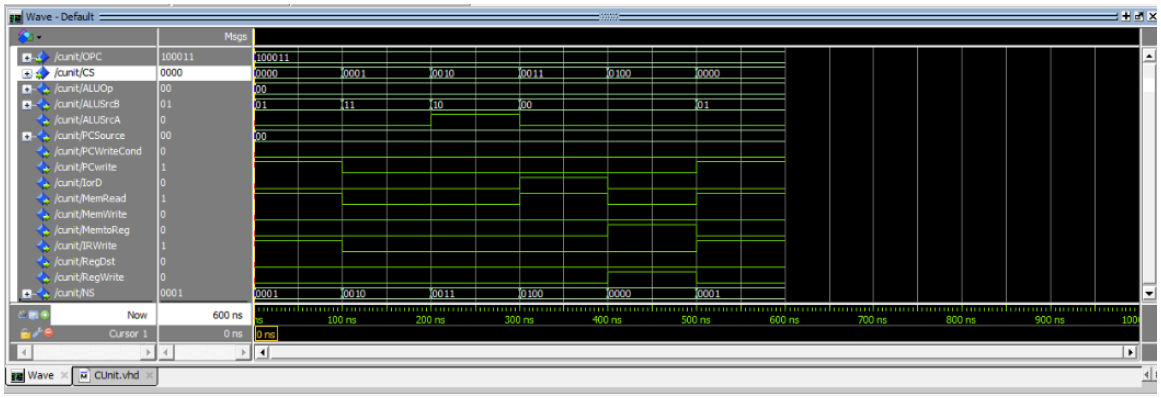
R-Type



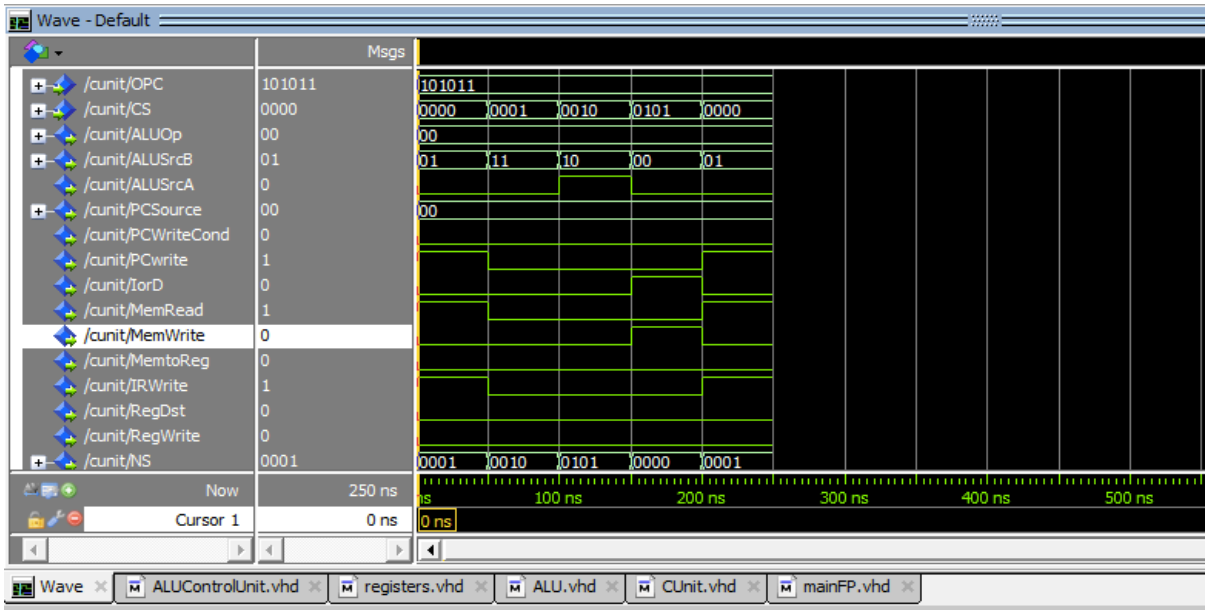
Jump


ModelSim PE Student Edition 10.1c
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
/examples/Datapath
Wave - Default
Msgs
/cunit/OPC 000010
/cunit/CS 0000
/cunit/ALUOp 00
/cunit/ALUSrcB 01
/cunit/ALUSrcA 0
/cunit/PCSource 00
/cunit/PCWriteCond 0
/cunit/PCwrite 1
/cunit/IorD 0
/cunit/MemRead 1
/cunit/MemWrite 0
/cunit/MemtoReg 0
/cunit/IRWrite 1
/cunit/RegDst 0
/cunit/RegWrite 0
/cunit/NS 0001
Now 150 ns
Cursor 1 0 ns
100 ns 200 ns 300 ns 400 ns
Wave ALUControlUnit.vhd registers.vhd ALU.vhd CUnit.vhd mainFP.vhd
sim - Default
Instance Design unit Design unit type Visibility Total coverage
Transcript
VSIM 77>

Load


Wave - Default
Msgs
/cunit/OPC 100011
/cunit/CS 0000
/cunit/ALUOp 00
/cunit/ALUSrcB 01
/cunit/ALUSrcA 0
/cunit/PCSource 00
/cunit/PCWriteCond 0
/cunit/PCwrite 1
/cunit/IorD 0
/cunit/MemRead 1
/cunit/MemWrite 0
/cunit/MemtoReg 0
/cunit/IRWrite 1
/cunit/RegDst 0
/cunit/RegWrite 0
/cunit/NS 0001
Now 600 ns
Cursor 1 0 ns
100 ns 200 ns 300 ns 400 ns 500 ns 600 ns 700 ns 800 ns 900 ns
Wave CUnit.vhd

Store



Appendix 2

- Program counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pc is
   port(
      writeEnable : in  std_logic;
      addrInput   : in  std_logic_vector(31 downto 0);
      addrOutput        : out std_logic_vector(31 downto 0)
   );
end entity;

architecture behavior of pc is

   signal addr_Output : std_logic_vector(31 downto 0);

begin

         addrOutput <= addr_Output;

   process (writeEnable, addrInput) begin
```

```
      if( writeEnable = '1' ) then
         addr_Output <= addrInput after 10 ns;
      end if;
          end process;
end architecture;
```

- Memory

```
library ieee;
 use ieee.std_logic_1164.all;
 use ieee.std_logic_unsigned.all;

entity Memory is generic ( DATA_WIDTH :integer := 32; ADDR_WIDTH :integer := 10 );
 port(
      addressIn : in std_logic_vector(31 downto 0); --address Input
      data1 : out std_logic_vector(DATA_WIDTH-1 downto 0); --data Output rs
      data3 : in std_logic_vector(DATA_WIDTH-1 downto 0); --data Input
      MemRead : in std_logic;
      MemWrite : in std_logic
      );
 end entity;

architecture rtl of Memory is
--Internal Variables--
 signal data_out1 : std_logic_vector(DATA_WIDTH-1 downto 0);
 signal address1 : std_logic_vector(ADDR_WIDTH-1 downto 0);
 constant RAM_DEPTH :integer := 2**(ADDR_WIDTH);
   type RAM is array (integer range<>) of std_logic_vector(DATA_WIDTH-1 downto 0);
 signal mem : RAM (0 to RAM_DEPTH-1);


begin

address1 <= addressIn(ADDR_WIDTH-1 downto 0);
data1 <= data_out1;

--Memory Write Block
MEM_WRITE:
   process (addressIn, data3, MemWrite)
    begin
      if( MemWrite='1') then
        mem( conv_integer(address1)) <= data3;
      end if;
   end process;

MEM_READ:
```

```
  process (addressIn, MemRead)
   begin
     if(MemRead='1') then
       data_out1 <= mem(conv_integer(address1));
     end if;
  end process;

end architecture;
```

- Instruction register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity instructionRegister is
  port(
            IRWrite    : in  std_logic;
            instrucInput : in  std_logic_vector(31 downto 0);
                  opCode     : out std_logic_vector(5 downto 0);
            regRs           : out std_logic_vector(4 downto 0);
                  regRt          : out std_logic_vector(4 downto 0);
                  regRd          : out std_logic_vector(4 downto 0);
                  imm       : out std_logic_vector(15 downto 0);
                  jumpAddr   : out std_logic_vector(25 downto 0);
                  funcCode   : out std_logic_vector(5 downto 0)
      );
end entity;

architecture behavior of instructionRegister is

  signal addr_Output1 : std_logic_vector(5 downto 0);
        signal addr_Output2 : std_logic_vector(4 downto 0);
          signal addr_Output3 : std_logic_vector(4 downto 0);
          signal addr_Output4 : std_logic_vector(4 downto 0);
        signal addr_Output5 : std_logic_vector(15 downto 0);
          signal addr_Output6 : std_logic_vector(25 downto 0);
          signal addr_Output7 : std_logic_vector(5 downto 0);

begin

        opCode  <= addr_Output1;
        regRs   <= addr_Output2;
        regRt   <= addr_Output3;
        regRd   <= addr_Output4;
        imm     <= addr_Output5;
```

```
 jumpAddr <= addr_Output6;
        funcCode <= addr_Output7;

   process (IRWrite , instrucInput) begin
      if( IRWrite  = '1' ) then
         addr_Output1 <= instrucInput(31 downto 26);
                              addr_Output2 <= instrucInput(25 downto 21);
                              addr_Output3 <= instrucInput(20 downto 16);
                              addr_Output4 <= instrucInput(15 downto 11);
                              addr_Output5 <= instrucInput(15 downto 0);
                              addr_Output6 <= instrucInput(25 downto 0);
                              addr_Output7 <= instrucInput(5 downto 0);
      end if;
          end process;

end architecture;
```

- Registers

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity registers is generic ( DATA_WIDTH :integer := 32; ADDR_WIDTH :integer := 5 );

   port(
      address1 : in std_logic_vector(ADDR_WIDTH-1 downto 0); --address Input read
register rs
      address2 : in std_logic_vector(ADDR_WIDTH-1 downto 0); --address Input read
register rt
      address3 : in std_logic_vector(ADDR_WIDTH-1 downto 0); --address Input write
resister rd
      data1    : out std_logic_vector(DATA_WIDTH-1 downto 0); --data Output rs
      data2    : out std_logic_vector(DATA_WIDTH-1 downto 0); --data output rt
      data3    : in std_logic_vector(DATA_WIDTH-1 downto 0); --data Input
      RegWrite      : in std_logic
      );
end entity;

architecture rtl of registers is

   --Internal Variables--
   constant RAM_DEPTH :integer := 2**ADDR_WIDTH;

   signal data_out1 : std_logic_vector(DATA_WIDTH-1 downto 0);
   signal data_out2 : std_logic_vector(DATA_WIDTH-1 downto 0);
```

```
   type RAM is array (integer range<>)of std_logic_vector(DATA_WIDTH-1 downto 0);
   signal mem : RAM (0 to RAM_DEPTH-1);

begin

   data1 <= data_out1;
   data2 <= data_out2;

   --Memory Write Block
   MEM_WRITE:
   process (address3, data3, RegWrite) begin
      if(RegWrite ='1') then
         mem( conv_integer(address3))<=data3;
      end if;
   end process;

   MEM_READ:
   process (address1, address2, RegWrite) begin
      data_out1 <= mem( conv_integer(address1));
      data_out2 <= mem( conv_integer(address2));
   end process;

end architecture;
```

- ALU

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;

entity ALU is
 generic ( delay: time:=2 ns );
 port ( RegA: in std_logic_vector (31 downto 0);
       RegB: in std_logic_vector (31 downto 0);
       RegC: in std_logic_vector (4 downto 0);--register for the shamt; for sll instruction
       Oper: in std_logic_vector (2 downto 0);
       Result: out std_logic_vector (31 downto 0);
       Zero: out std_logic);
   end ALU;

architecture behavioral of ALU is

signal vectorsll : std_logic_vector(31 downto 0);
```

```
   begin
    process ( RegA, RegB, Oper)
     begin
      case Oper is
      when "010" =>-- instruccion add
       Result<= RegA+RegB;
      when "110" => --instruccion sub/subi
       Result<= RegA+(not(RegB))+1;
       if ((RegA+(not(RegB))+1)="00000000000000000000000000000000") then
Zero<='1';
       end if;
      when "000" => --instruccion and
       Result<= ((RegA)and(RegB));
      when "001" => --instruccion or
       Result<= ((RegA)or(RegB));
      when "011" => --instruccion sll
         vectorsll<="00000000000000000000000000000000";
         vectorsll(31 downto conv_integer(RegC)) <= RegA((31-conv_integer(RegC))
downto 0);
         Result <= vectorsll;
      when "111" =>-- instruccion slt 111
       if (RegA<RegB) then Result <= "11111111111111111111111111111111"; else
       Result <= "00000000000000000000000000000000";
       end if;
      when others => -- instruccion nand "101"
       Result <= not(RegA and RegB);

      end case;
     --falta implementar Zero
    end process;

end behavioral;
```

- Bits5Reg

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bits5reg is
   port (
            data5  :in  std_logic_vector(4 downto 0);
     clk   :in  bit;
     q5    :out std_logic_vector(4 downto 0)
        );
end entity;
```

```
architecture behavior of bits5reg is

begin
   process (clk) begin
            if (clk='1' and clk'event) then
                     q5 <= data5;
                  end if;
         end process;

end architecture;
```

- BitsReg

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bitsReg is
   port (
              data  :in  std_logic_vector(31 downto 0);
      clk   :in  std_logic;
      q     :out std_logic_vector(31 downto 0)
          );
end entity;

architecture behavior of bitsReg is

begin
   process (clk) begin
            if (rising_edge(clk)) then
                     q <= data;
                  end if;
         end process;

end architecture;
```

- Clockbox

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;

entity clockbox is
```

```
   port (clk: inout bit);
   end clockbox;

architecture behavioral of clockbox is
   begin
      process
        begin
          clk <= not(clk);
          wait for 100 ns;
      end process;

end behavioral;
```

- Gate

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity gate2 is
   port (
            a :in  std_logic;
                  b :in  std_logic;
                   y :out std_logic
          );
end entity;

architecture behavior of gate2 is

begin
   process(a,b) begin
             y <= ((a) and (b));
          end process;

end architecture;
```

- Mux2_1

```
library ieee;
    use ieee.std_logic_1164.all;

entity mux2_1 is
 generic ( delay: time:=2 ns );
 port  (e0,e1: in std_logic;
       C: in std_logic;
```

```
        s: out std_logic);
end mux2_1;

architecture behavioral of mux2_1 is
  begin
    with (C) select

      s <= e0 when '0',
          e1 when others;

end behavioral;
```

- Or2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity or2 is
   port (
            a :in  std_logic_vector(31 downto 0);
                    b :in  std_logic_vector(31 downto 0);
                    y :out std_logic_vector(31 downto 0)
          );
end entity;

architecture behaviour of or2 is

begin
   process(a,b) begin
            y <= a or b;
          end process;

end architecture;
```

- Mux2_5b

```
library ieee;
     use ieee.std_logic_1164.all;

entity mux2_5b is
  port  (e0,e1: in std_logic_vector (4 downto 0);
        C: in std_logic;
        s: out std_logic_vector (4 downto 0));
end mux2_5b;
```

```
architecture behavioral of mux2_5b is
  begin
   with (C) select

      s <= e0 when '0',
          e1 when others;

end behavioral;
```

- Mux2

```
library ieee;
     use ieee.std_logic_1164.all;

entity mux2 is
  generic ( delay: time:=2 ns );
  port  (e0,e1: in std_logic_vector (31 downto 0);
         C: in std_logic;
         s: out std_logic_vector (31 downto 0));
end mux2;

architecture behavioral of mux2 is
  begin
   with (C) select

      s <= e0 when '0',
          e1 when others;

end behavioral;
```

- Mux4

```
library ieee;
     use ieee.std_logic_1164.all;

entity mux4 is
  generic ( delay: time:=2 ns );
  port  (e0,e1,e2,e3: in std_logic_vector (31 downto 0);
         C: in std_logic_vector (1 downto 0);
         s: out std_logic_vector (31 downto 0));
end mux4;

architecture behavioral of mux4 is
  begin
```

```
   with (C) select
     s <= e0 when "00",
          e1 when "01",
          e2 when "10",
          e3 when others;

end behavioral;
```

- New-Current

```
library ieee;
     use ieee.std_logic_1164.all;
     use ieee.std_logic_unsigned.all;
     use ieee.std_logic_arith.all;

entity New_Current is
  port (NewS: in std_logic_vector (3 downto 0);--New states taht are going to be load
       clk: in bit;
       CS: out std_logic_vector (3 downto 0));--output of the block

   end New_Current;

architecture behavioral of New_Current is
  begin
     process (NewS, clk)
       begin
        --clock rising edge
        if (clk='1' and clk'event) then
           CS <= NewS;
         end if;

       end process;
end behavioral;
```

- ShiftLeft2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity shiftLeft2 is
   port(
            dataIn  :in  std_logic_vector(31 downto 0);
                    dataOut :out std_logic_vector(31 downto 0)
   );
```

```
end entity;

architecture behavior of shiftLeft2 is

        begin
      process(dataIn) begin
                    dataOut(31 downto 2) <= dataIn(29 downto 0);
                    dataOut(1 downto 0) <= "00";
                end process;
end architecture;
```

- ShiftLeft22

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity shiftleft22 is
   port(
       dataIn :in std_logic_vector(25 downto 0);
       dataOut :out std_logic_vector(27 downto 0)
      );
end entity;

architecture behavior of shiftleft22 is
  begin
   process(dataIn)
     begin
      dataOut(27 downto 2) <= dataIn(25 downto 0) ;
      dataOut(1 downto 0) <= "00";
   end process;

end architecture;
```

- SignExtend

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity signExtend is
   port(
            dataIn  :in  std_logic_vector(15 downto 0);
                  dataOut :out std_logic_vector(31 downto 0)
   );
```

```
end entity;

architecture behavior of signExtend is

        begin
    process(dataIn) begin
                if( dataIn(15)='1') then
                    dataOut(15 downto 0)  <= dataIn(15 downto 0);
                        dataOut(31 downto 16) <= "1111111111111111";
                else
                    dataOut(15 downto 0)  <= dataIn(15 downto 0);
                        dataOut(31 downto 16) <= "0000000000000000";
                end if;
            end process;
end architecture;
```

- CUnit

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;

entity CUnit is
 generic ( delay: time:=2 ns );
 port  (OPC: in std_logic_vector (5 downto 0);--input of the Operation code
       CS: in std_logic_vector (3 downto 0);-- input of the current state
       ALUOp: out std_logic_vector (1 downto 0);--output to the ALUControl
       ALUSrcB: out std_logic_vector (1 downto 0);--output to the mux4 of the ALU
       ALUSrcA: out std_logic;-- output to mux2 of the ALU
       PCSource: out std_logic_vector (1 downto 0); --goes to mux4 before of the PC
       PCWriteCond: out std_logic; --goes to and gate with the zero signal from the ALU
       PCwrite: out std_logic; --to allow to write in PC (the jump, branch PC+4)
       IorD: out std_logic;--load/store
       MemRead: out std_logic;--read on memory
       MemWrite: out std_logic; --write in memory
       MemtoReg: out std_logic; --to the mux2 to write in the register of memory
       IRWrite: out std_logic; --it allows to load in the register the intruction
       RegDst: out std_logic; --to mux2 to write in mem register
       RegWrite: out std_logic; --to write in the memory registers
       NewS: out std_logic_vector (3 downto 0);
       Branchsignal: out std_logic);--new states


end CUnit;
```

```
architecture behavioral of CUnit is
   begin
    process (OPC, CS)
      begin
       PCwrite <=
((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(not(CS(0))))or((CS(3))and(not(CS(
2)))and(not(CS(1)))and(CS(0)));
       PCWriteCond <= ((CS(3))and(not(CS(2)))and(not(CS(1)))and(not(CS(0))));
       MemRead <= (((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(not(CS(0)))) or
((not(CS(3)))and(not(CS(2)))and(CS(1))and(CS(0))));
       MemWrite <= ((not(CS(3)))and(CS(2))and(not(CS(1)))and(CS(0)));
       IorD<= (((not(CS(3)))and(CS(2))and(not(CS(1)))and(CS(0))) or
((not(CS(3)))and(not(CS(2)))and(CS(1))and(CS(0))));
       IRWrite <= ((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(not(CS(0))));
       MemtoReg <= ((not(CS(3)))and(CS(2))and(not(CS(1)))and(not(CS(0))));
       PCSource(1)<= ((CS(3))and(not(CS(2)))and(not(CS(1)))and(CS(0)));
       PCSource(0)<= ((CS(3))and(not(CS(2)))and(not(CS(1)))and(not(CS(0))));
       ALUOp(1) <= ((not(CS(3)))and(CS(2))and(CS(1))and(not(CS(0))));
       ALUOp(0) <= ((CS(3))and(not(CS(2)))and(not(CS(1)))and(not(CS(0))));
       ALUSrcB(1) <= (((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))) or
((not(CS(3)))and(not(CS(2)))and(CS(1))and(not(CS(0)))));
       ALUSrcB(0) <= (((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))) or
((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(not(CS(0)))));
       ALUSrcA <= (((not(CS(3)))and(not(CS(2)))and(CS(1))and(not(CS(0)))) or
(((not(CS(3)))and(CS(2))and(CS(1))and(not(CS(0))))) or
((CS(3))and(not(CS(2)))and(not(CS(1)))and(not(CS(0)))));
       RegWrite <= (((not(CS(3)))and(CS(2))and(not(CS(1)))and(not(CS(0)))) or
(((not(CS(3)))and(CS(2))and(CS(1))and(CS(0)))));
       RegDst <= ((not(CS(3)))and(CS(2))and(CS(1))and(CS(0)));
      case (OPC) is
       when "000100" => Branchsignal <='0';
       when others => Branchsignal <='1';
      end case;

       --Next States:

       NewS(0) <= (((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(not(CS(0)))) or
            ((not(CS(3)))and(not(CS(2)))and(CS(1))and(not(CS(0))) and
(OPC(5))and(not(OPC(4)))and(not(OPC(3)))and(not(OPC(2)))and(OPC(1))and(OPC(0
))) or
            ((not(CS(3)))and(not(CS(2)))and(CS(1))and(not(CS(0))) and
(OPC(5))and(not(OPC(4)))and(OPC(3))and(not(OPC(2)))and(OPC(1))and(OPC(0)))
or
            ((not(CS(3)))and(CS(2))and(CS(1))and(not(CS(0)))) or
```

```
((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and(not(OPC(5)))and(not(OP
C(4)))and(not(OPC(3)))and(not(OPC(2)))and(OPC(1))and(not(OPC(0)))) ;
        NewS(1) <=
(((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and((OPC(5))and(not(OPC(
4)))and(OPC(3))and(not(OPC(2)))and(OPC(1))and(OPC(0)) ) )or

(((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and((OPC(5))and(not(OPC(
4)))and(not(OPC(3)))and(not(OPC(2)))and(OPC(1))and(OPC(0)) ) )or

((not(CS(3)))and(not(CS(2)))and(CS(1))and(not(CS(0)))and(OPC(5))and(not(OPC(4)
))and(not(OPC(3)))and(not(OPC(2)))and(OPC(1))and(OPC(0)))or

((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and(not(OPC(5)))and(not(OP
C(4)))and(not(OPC(3)))and(not(OPC(2)))and(not(OPC(1)))and(not(OPC(0))))
            or ((not(CS(3)))and(CS(2))and(CS(1))and(not(CS(0))))));
        NewS(2) <= (((not(CS(3)))and(not(CS(2)))and(CS(1))and(CS(0))) or

((not(CS(3)))and(not(CS(2)))and(CS(1))and(not(CS(0)))and(OPC(5))and(not(OPC(4)
))and(OPC(3))and(not(OPC(2)))and(OPC(1))and(OPC(0)))or

((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and(not(OPC(5)))and(not(OP
C(4)))and(not(OPC(3)))and(not(OPC(2)))and(not(OPC(1)))and(not(OPC(0)))) or
            ((not(CS(3)))and(CS(2))and(CS(1))and(not(CS(0))) ) );
        NewS(3) <=
(((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and(not(OPC(5)))and(not(O
PC(4)))and(not(OPC(3)))and(OPC(2))and(not(OPC(1)))and(not(OPC(0))))
            or
((not(CS(3)))and(not(CS(2)))and(not(CS(1)))and(CS(0))and(not(OPC(5)))and(not(OP
C(4)))and(not(OPC(3)))and(not(OPC(2)))and(OPC(1))and(not(OPC(0)))));

    end process;

end behavioral;
```

- MainFP

```
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.std_logic_unsigned.all;
    use ieee.std_logic_arith.all;

entity mainFP is
  end mainFP;

architecture behavioral of mainFP is
```

```
component New_Current
 port (NewS: in std_logic_vector (3 downto 0);
      clk: in bit;
      CS: out std_logic_vector (3 downto 0));
    end component;

component ALU
   port (RegA: in std_logic_vector (31 downto 0);
        RegB: in std_logic_vector (31 downto 0);
        RegC: in std_logic_vector (4 downto 0);
        Oper: in std_logic_vector (2 downto 0);
        Result: out std_logic_vector (31 downto 0);
        Zero: out std_logic);
 end component;

component CUnit
   port  (OPC: in std_logic_vector (5 downto 0);
         CS: in std_logic_vector (3 downto 0);
         ALUOp: out std_logic_vector (1 downto 0);
         ALUSrcB: out std_logic_vector (1 downto 0);
         ALUSrcA: out std_logic;
         PCSource: out std_logic_vector (1 downto 0);
         PCWriteCond: out std_logic;
         PCwrite: out std_logic;
         IorD: out std_logic;
         MemRead: out std_logic;
         MemWrite: out std_logic;
         MemtoReg: out std_logic;
         IRWrite: out std_logic;
         RegDst: out std_logic;
         RegWrite: out std_logic;
         NewS: out std_logic_vector (3 downto 0);
         Branchsignal: out std_logic);
  end component;

 component ALUCUnit
    port( ALUOp: in std_logic_vector(1 downto 0);
         Ff:    in std_logic_vector(5 downto 0);
         Oper:  out std_logic_vector(2 downto 0));
  end component;

 component Mux2
      port  (e0,e1: in std_logic_vector (31 downto 0);
           C: in std_logic;
           s: out std_logic_vector (31 downto 0));
```

```
    end component;

   component Mux4
       port  (e0,e1,e2,e3: in std_logic_vector (31 downto 0);
            C: in std_logic_vector (1 downto 0);
            s: out std_logic_vector (31 downto 0));
    end component;

   component registers2 generic ( DATA_WIDTH :integer := 32; ADDR_WIDTH :integer
:= 5 );
         port( address1 : in std_logic_vector(ADDR_WIDTH-1 downto 0); --address Input
read register rs
            address2 : in std_logic_vector(ADDR_WIDTH-1 downto 0); --address Input
read register rt
            address3 : in std_logic_vector(ADDR_WIDTH-1 downto 0); --address Input
write resister rd
            data1    : out std_logic_vector(DATA_WIDTH-1 downto 0); --data Output rs
            data2    : out std_logic_vector(DATA_WIDTH-1 downto 0); --data output rt
            data3    : in std_logic_vector(DATA_WIDTH-1 downto 0); --data Input
            RegWrite : in std_logic);
    end component;

    component clockbox
       port (clk: inout bit);
    end component;

    component bitsReg
        port (datareg  :in  std_logic_vector(31 downto 0);
            clk   :in  bit;
            qreg     :out std_logic_vector(31 downto 0));
    end component;

    component instructionRegister
        port( IRWrite     : in  std_logic;
            instrucInput : in  std_logic_vector(31 downto 0);
                        opCode      : out std_logic_vector(5 downto 0);
            regRs              : out std_logic_vector(4 downto 0);
                           regRt              : out std_logic_vector(4 downto 0);
                           regRd              : out std_logic_vector(4 downto 0);
                           imm        : out std_logic_vector(15 downto 0);
                           jumpAddr    : out std_logic_vector(25 downto 0);
                           funcCode    : out std_logic_vector(5 downto 0));
                    end component;

     component Memory
        port( addressIn : in std_logic_vector(31 downto 0); --address Input
```

```
        data1 : out std_logic_vector(31 downto 0); --data Output rs
        data3 : in std_logic_vector(31 downto 0); --data Input
        MemRead : in std_logic;
        MemWrite : in std_logic);
end component;

component or2
    port ( a :in  std_logic;
                    b :in  std_logic;
                    y :out std_logic);
            end component;

            component pc
    port( writeEnable : in  std_logic;
        addrInput   : in  std_logic_vector(31 downto 0);
        addrOutput          : out std_logic_vector(31 downto 0));
end component;

component shiftLeft2
    port( dataIn  :in  std_logic_vector(31 downto 0);
                    dataOut :out std_logic_vector(31 downto 0));
end component;

component signExtend
    port(dataIn  :in  std_logic_vector(15 downto 0);
            dataOut :out std_logic_vector(31 downto 0));
end component;

component  gate2
    port ( a :in  std_logic;
                    b :in  std_logic;
                    y :out std_logic);
            end component;

            component bits5reg
              port ( data5  :in  std_logic_vector(4 downto 0);
      clk   :in  bit;
      q5    :out std_logic_vector(4 downto 0));
end component;

component shiftleft22
  port( dataIn :in std_logic_vector(25 downto 0);
      dataOut :out std_logic_vector(27 downto 0));
end component;

component mux2_5b
```

```
        port  (e0,e1: in std_logic_vector (4 downto 0);
              C: in std_logic;
              s: out std_logic_vector (4 downto 0));
      end component;

      component mux2_1
         port  (e0,e1: in std_logic;
               C: in std_logic;
               s: out std_logic);
      end component;

   --places where the architecture of our components:
      for States: New_Current use entity work.New_Current (behavioral);
      for ALUUnit: ALU use entity work.ALU (behavioral);
      for ControlUnit: CUnit use entity work.CUnit (behavioral);
      for ALUControlUnit: ALUCUnit use entity work.ALUCUnit (behave);
      for clock: clockbox use entity work.clockbox (behavioral);
      for registers_mem: registers2 use entity work.registers (rtl);
      for muxB, muxPC: Mux4 use entity work.Mux4 (behavioral);
      for muxA, muxMem, muxWrite: mux2 use entity work.mux2 (behavioral);
      for PCunit: pc use entity work.pc (behavior);
      for Memoryunit: Memory use entity work.Memory (rtl);
      for InstRegister: instructionRegister use entity work.instructionRegister (behavior);
      for SignExt: signExtend use entity work.signExtend (behavior);
      for ShifL2_1: shiftLeft2 use entity work.shiftLeft2 (behavior);
      for RegisterA, RegisterB, RegisterALUOut, RegisterMemData: bitsReg use entity
work.bitsReg (behavior);
      for RegisterC: bits5reg use entity work.bits5reg (behavior);
      for Or2gate: or2 use entity work.or2 (behavior);
      for And2: gate2 use entity work.gate2 (behavior);
      for ShiftL2_2: shiftleft22  use entity work.shiftleft22  (behavior);
      for muxRD : mux2_5b use entity work.mux2_5b  (behavioral);
      for muxBranch : mux2_1 use entity work.mux2_1 (behavioral);

     --signals that are necessary:
     signal CS, NewS : std_logic_vector (3 downto 0);
     signal OPC,Ff : std_logic_vector (5 downto 0);
     signal ALUOp, ALUSrcB, PCSource : std_logic_vector (1 downto 0);
     signal
ALUSrcA,PCWriteCond,PCwrite,IorD,MemRead,MemWrite,MemtoReg,IRWrite,RegDst,
RegWrite,Zero,Branchsignal,NZero,Zerofinal,FinalPC,ZPC: std_logic;
     signal
RegA,RegB,Result,data1,data2,data3,PCout,SEx,SL2,addressmem,Memdata,Writedata,
MDR,ALUO,RB,RA,PCint,JAdd: std_logic_vector (31 downto 0);
     signal Oper : std_logic_vector (2 downto 0);
     signal RS,RT,RD,RegC,WriteRegister : std_logic_vector(4 downto 0);
```

```
   signal clk : bit;
   signal IMM : std_logic_vector (15 downto 0);
   signal JA : std_logic_vector (25 downto 0);
   signal JA_1 : std_logic_vector (27 downto 0);



   begin
    JAdd (27 downto 0) <= JA_1(27 downto 0);
    JAdd (31 downto 28) <= PCout(31 downto 28);
    NZero <= not(Zero);

--connections between blocks:
    clock : clockbox PORT MAP (clk);
    States : New_Current PORT MAP (NewS,clk,CS);
    ControlUnit : CUnit PORT MAP
(OPC,CS,ALUOp,ALUSrcB,ALUSrcA,PCSource,PCWriteCond,PCWrite,IorD,MemRead,Me
mWrite,MemtoReg,IRWrite,RegDst,RegWrite,NewS,Branchsignal);
    ALUUnit : ALU PORT MAP (RegA,RegB,RegC,Oper,Result,Zero);
    ALUControlUnit : ALUCUnit PORT MAP (ALUOp, Ff, Oper);
    registers_mem : registers2 PORT MAP
(RS,RT,WriteRegister,data1,data2,data3,RegWrite);
    Memoryunit: Memory PORT MAP
(addressmem,Memdata,Writedata,MemRead,MemWrite);
    InstRegister: instructionRegister PORT MAP
(IRWrite,Memdata,OPC,RS,RT,RD,IMM,JA,Ff);
    muxRD : Mux2_5b PORT MAP ( RT, RD, RegDst, WriteRegister);
    muxWrite : Mux2 PORT MAP ( ALUO, MDR, MemtoReg, data3);
    muxMem : Mux2 PORT MAP ( PCout, ALUO, IorD, addressmem);
    muxB : Mux4 PORT MAP ( RB, "00000000000000000000000000000100", SEx, SL2,
ALUSrcB, RegB);
    muxA : Mux2 PORT MAP (PCout, RA, ALUSrcA, RegA);
    muxPC : Mux4 PORT MAP (Result, ALUO,
JAdd,"00000000000000000000000000000000",PCSource, PCint);
    muxBranch : mux2_1 PORT MAP (Zero,NZero,Branchsignal,Zerofinal);
    PCunit : pc PORT MAP (FinalPC, PCint, PCout);
    ShiftL2_2 : shiftleft22 PORT MAP (JA,JA_1);
    ShifL2_1 : shiftLeft2 PORT MAP (SEx, SL2);
    SignExt : signExtend PORT MAP (IMM,SEx);
    RegisterC : bits5reg PORT MAP (RD,clk,RegC);
    RegisterA : bitsReg PORT MAP (data1, clk, RA);
    RegisterB : bitsReg PORT MAP (data2, clk, RB);
    RegisterALUOut : bitsReg PORT MAP (Result,clk,ALUO);
    RegisterMemData: bitsReg PORT MAP (Memdata,clk,MDR);
    And2 : gate2 PORT MAP (Zerofinal, PCWriteCond, ZPC);
    Or2gate : or2 PORT MAP (ZPC,PCWrite,FinalPC);
```

```
  tb : process
     begin
       wait for 100 ns;
         MemWrite <= '1';
         MemRead <= '0';
         addressmem <= "00000000000000000000000001100100";
         Writedata <= "11111111111111111111111111111111";
         RegWrite <= '1';
         WriteRegister <= "01010";
         data3 <= "00000000000000000000000000000000";
       wait for 100 ns;
         MemWrite <= '0';
         RegWrite <= '0';
       wait for 100 ns;
         MemWrite <= '1';
         MemRead <= '0';
         addressmem <= "00000000000000000000000000000000";
         Writedata <= "10001101010110010000000001100100";
       wait for 100 ns;
         MemWrite <= '0';
         MemRead <= '0';
         PCout <= "00000000000000000000000000000000";
         CS <= "0000";
       wait for 1 ns;

  end process;

end behavioral;
```