# MIPS in Verilog

Nathan St. Amour & Colt Kiser

Ohio University, Athens, OH 45701

**Abstract.** This report describes an implementation of a subset of the MIPS instruction architecture as described in class and the book[1]. The implementation is simulated using the architecture modeling language Verilog. Included in our implementation is a language of instructions that is broken down into three types: The memory-reference instructions(lw and sw), the arithmetic logic instructions(add, sub, AND, OR, , and branch instructions(j and beq). These instructions can be used to implement a simple language. It includes several fully functional logic units that are brought together to define the semantics of each instruction on simulated hardware. We will describe the modules used to realize this design and the methodology that allows MIPS be a successfull implementation and any problems that have arisen while implementing the processor.

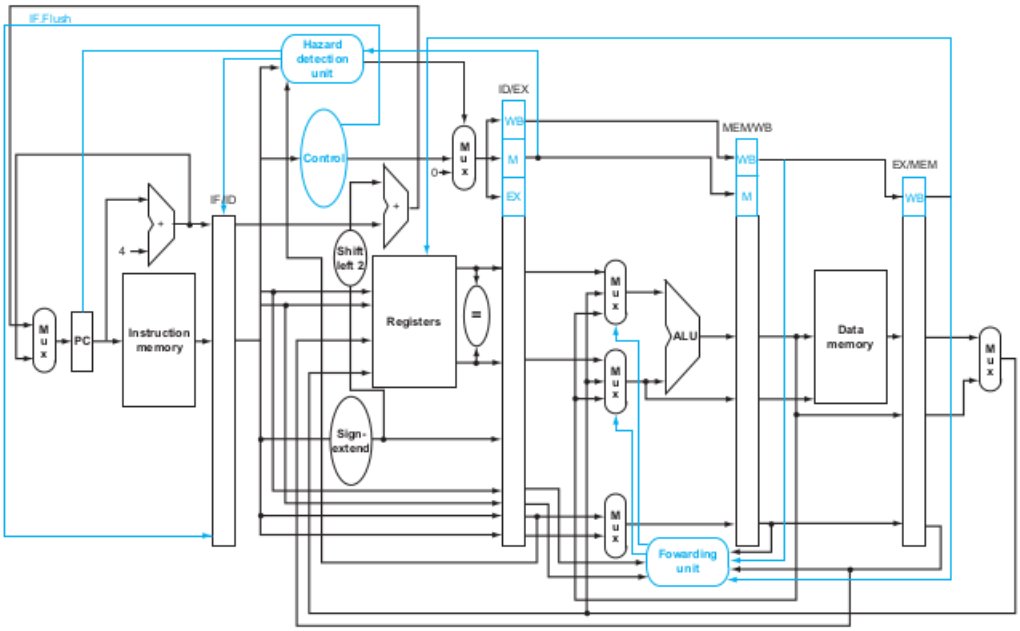



**FIGURE 4.65 The final datapath and control for this chapter.** Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.57 and the multiplexor controls from Figure 4.51.

[1]

# Table of Contents

# 1 Introduction and MIPS Design

## 1.1 MIPS

MIPS. is a reduced instruction set(RISC), Our implementation is going to be pipelined vs multicycle. It can be used for embedded devices, microcontrollers or for a wide range of other styles of computing. MIPS is used in embedded systems such as: windows ce devices, gateways and some video game implementations.

# 2 Design Methodology

The MIPS instruction set is designed with the ideal of a simple instruction set that includes instructions that largely use the same set of logical operations to carry out their task. Our design methodology stays true to that, especially with the small instruction set. We implement all of the basic features of a MIPS processor. This design includes an Instruction memory, Registers, Data memory, ALU operations etc... We also looked into implementing Forwarding(6.22) to solve harzards but did not have enought time to get it to work properly with the final MIPS design.

## 2.1 Mips Data Storage

We have split up the Instuction memory and Data memory in order to prevent to the hazard that would result from trying to fetch an instruction and read or write to the Data memory at the same time. So the memory that stores instructions ends up being separated from the main Data memory. There is also a register file which allows for functionality of registers. Registers provide quick lookup to data for operations. This allows the slow main memory to be used for writing and lookup less frequently. The Data memory include 128 words for long term storage of data. This Data memory is used by instructions sw and lw.

The Data memory module is described in greater detail in the section further into the document 6.7

The Instruction memory stores the value of the PC counter and the different types of instructions that we have implemented in our MIPS implementation. 6.8 [1]

# 3 Instruction Set

The MIPS instruction set includes three types of commands: R-type, I-type, J-type. All three of the command types share have some regularities. This allows for the use of a reduced number of features in hardware to implement the instructions. [1]

# 4 Processor Purpose/Functions

The processor can be used to do anything that can be imagined by the small set of instruction and state which are implemented in the code corresponding with this report. The 5 stages of our pipelined MIPS processor are implemented here: IF, ID, EX, MEM, WB. The processor could be further extended to allow for forwarding and other hazard mitigation techniques. The beq instruction does not work without forwarding so without that functionality we cannot use the branch equal command.

## 4.1 Modules

The module system in Verilog provides an essential feature of abstraction. A module is written to perform a specific task that may include several instructions which is then used to implement more general features by passing inputs and outputs to the module and using the affected outputs.

**4.1.1 MIPS** This is our attempt at bring together all of the modules and stages in order to have a fully functional processor. We have implemented three instructions with opcode shown in the test bench. 6.1 We learned a lot during this stage but failed to completely tie together all of the peices into a fully functional processor.

**4.1.2 IF** The IF module that we implemented will take care of the Instruction Fetch stage of the MIPS instruction pipeline. The purpose of the module is to fetch the instruction opcode, which identifies the instruction is being executed, from the instruction memory. The other responsibility of the IF module is to increment the Program Counter (PC) each time an instruction is executed. Our implementation increments the counter by a value of 1 each time an instruction is implemented.6.2

**4.1.3 ID** The purpose of the ID module is to complete the Instruction Decode stage of the MIPS instruction pipeline. The main purpose of this stage of the pipeline is to simply figure out what the instruction returned by the IF stage of the pipeline is supposed to do. The ID module figures out what the instruction is from the opcode and breaks apart the instruction code into its different parts. This step also decides what registers are required and can make this decision before the instruction is fully decoded due to the simple instruction format it is given. 6.3

**4.1.4 EX** The main purpose of the EX module to do the calculations that are required for the connection. This includes doing the arithmetic calculations required in mathematical instruction, like add or sub, and calculating memory references or adding up base and offset values. Our implementation uses a 32bit ALU that we created to calculate these values.

**4.1.5 MEM** The MEM module in our project handles all the memory access required for MIPS commands. This module is used if an instruction needs to load, store, or access data that is stored in memory. This stage of the pipeline also replaces the PC with a destination address if the instruction a branching instruction. If the instruction does not require either of these functions, then this module does nothing.6.5

**4.1.6 WB** The purpose of the WB module is to handle the Write Back part of the MIPS pipeline. This module has one simple function. This is to place the results of the instruction in the appropriate register, which is designated in the instruction. 6.6

**4.1.7 Data memory** The DataMemory module that we implemented either reads or writes and address from a specific register in memory. You can find the data memory here: 6.7

**4.1.8 Instruction Memory** This is the Instruction memory. All commands use the instructions memory and the PC counter uses it as well.

In our module for the Instruction memory takes as input an instruction address and an instruction to be returned (instruction_address, instruction). The instruction address is declared as a 32bit input and the instruction is a 32bit output. The instructions are then loaded into the 32bit register file instruction_memory_file. Then as needed the instruction memory register, instruction_memory_file is used to lookup the correct instruction address. Find the code to the instruction memory in the appendex. 6.8

**4.1.9 ALU** The arithmetic logic unit of our MIPS implementation is used by all commands that we have implemented except for the jump command. The ALU is 32 bit to correspond with the standard data size of a word that the MIPS design uses. The ALU is made up of many sub modules that are extracted away to make the logic of the Verilog implementation only 2 lines long. The ALU takes 7 reasonable parameters as input.

1. **a, b:** The two 32 bit binary numbers (a,b) that are declared as 32 bit inputs in the Verilog
2. **op:** The op-code that is declared as a 3 bit input
3. **result:** The 32 bit output of the operation
4. **set:** The one bit output result of the most-significant ADDER unit
5. **zero:** The one bit output that is set if the result is 0x00000000
6. **overflow:** One bit output that is 1 if the output of an operation creates an overflow

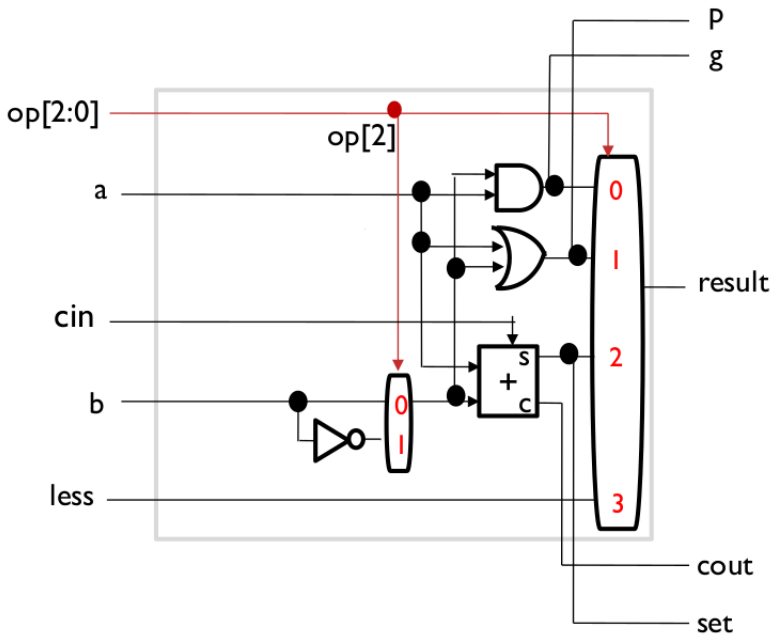
The 32 bit then makes two calls to 16 bit ALUs with some temporary wires to help us with the structure of the Verilog. The ALU is referenced in the appendex here: 6.13

*4.1.9.1 16 Bit ALU* The 16 bit ALU behaves in a similar way to the 32 bit version. However it does make use of the CLA unit, described in 4.1.9.2, and makes use of four 4 bit ALUs, the output of which is likewise combined into one 16 bit half word. The 16 bit ALU is referenced here: 6.14

*4.1.9.2 4 Bit ALU* The four bit adder uses a carry the Carry lookhead adder unit, 4.1.9.2 as well and makes use of four 1 bit ALUs. The 4 bit ALU that is implemented here: 6.15

**4.1.10 CLA** The carry look ahead adder module uses the logic that was described in class and in the book to implement a 4 bit adder. The logic behind it makes use of quickly predicting which bits will preform a carry. You can find the CLA code here: 6.16

**4.1.11 1 Bit ALU** The one bit ALU is implemented per the design given to us on homework three.




## 5 Conclusion

We were able to successfully implement and test all five stages if the MIPS instruction pipeline successfully, however, the MIPS processor that we implemented does not work completely right. We created test benches for the five stages of the pipeline and we are convinced that they are working correctly.

The test benches and waveforms created from the modules are included in the appendix of the report. Given more time to work on the project we are both confident that we could get together an implementation that had a fully functional set of commands.

# 6 Appendex

## 6.1 MIPS

```
`timescale 1ns / 1ps
module MIPSProcessor(Clk, Out_PC, Out_Instr, Out_BranchPC, Out_JumpPC,
    Out_PCSrc, Out_Jump, Out_DataA, Out_DataB, Out_SE, Out_Funct,
    Out_WriteAddr, Out_ALUResult, Out_ALUData, Out_Rs, Out_Rt, Out_Rd,
    Out_MEMAddress, Out_MEMData, Out_Data, Out_RegWrite, Out_EXControl1,
    Out_MEMControl1, Out_WBControl1, Out_MEMControl2, Out_WBControl2,
    Out_WBControl3);
input Clk;
output [31:0] Out_PC;
output [31:0] Out_Instr;
output [31:0] Out_Data;
output Out_RegWrite;
output [2:0] Out_Funct;
output [4:0] Out_Rs, Out_Rt, Out_Rd;
output [3:0] Out_EXControl1;
output [1:0] Out_MEMControl1, Out_WBControl1, Out_MEMControl2,
    Out_WBControl2;
output [4:0] Out_WriteAddr;
output [31:0] Out_ALUResult, Out_ALUData;
output [1:0] Out_WBControl3;
output [31:0] Out_MEMAddress, Out_MEMData;
output Out_PCSrc, Out_Jump;
output [31:0] Out_BranchPC, Out_JumpPC;
output [31:0] Out_DataA, Out_DataB, Out_SE;


reg [31:0] In_PC;

wire [31:0] Out_PC;
wire [31:0] Out_Instr;

wire [4:0] Out_WriteAddr;
wire [31:0] Out_Data;
wire Out_RegWrite, Out_PCSrc, Out_Jump;
wire [31:0] Out_BranchPC, Out_JumpPC;
wire [31:0] Out_DataA, Out_DataB, Out_SE;
wire [2:0] Out_Funct;
wire [4:0] Out_Rs, Out_Rt, Out_Rd;
wire [3:0] Out_EXControl1;
```

```
wire [1:0] Out_MEMControl1, Out_WBControl1;
wire [31:0] Out_ALUResult, Out_ALUData;
wire [1:0] Out_MEMControl2, Out_WBControl2;
wire [1:0] Out_WBControl3;
wire [31:0] Out_MEMAddress, Out_MEMData;

initial begin
        In_PC = 32'd0;
end

always @(negedge Clk) begin
    if(Out_PCSrc==1) begin
                In_PC=Out_BranchPC;
        end
        else if(Out_PCSrc==0) begin
                In_PC=Out_PC;
        end

end

IF myIF(In_PC, Clk, Out_PC, Out_Instr);
ID myID(Out_PC, Out_Instr, Out_WriteAddr, Out_Data, Out_RegWrite,
    Out_BranchPC, Out_JumpPC, Out_PCSrc, Out_Jump, Out_DataA, Out_DataB,
    Out_SE, Out_Funct, Out_Rs, Out_Rt, Out_Rd, Out_EXControl1,
    Out_MEMControl1, Out_WBControl1, Clk);
EX myEX(Out_DataA, Out_DataB, Out_SE, Out_Rt, Out_Rd, Out_EXControl1,
    Out_MEMControl1, Out_WBControl1, Out_Funct, Out_ALUResult, Out_ALUData,
     Out_WriteAddr, Out_MEMControl2, Out_WBControl2, Clk);
MEM myMEM(Out_ALUResult, Out_ALUData, Out_MEMControl2, Out_WBControl2,
    Out_MEMAddress, Out_MEMData, Out_WBControl3, Clk);
WB myWB(Out_MEMAddress, Out_MEMData, Out_WBControl3, Out_Data, Out_RegWrite
    , Clk);

endmodule
```

#### 6.1.1 TB/WF

```
`timescale 1ns / 1ps

module MIPSProcessorb();
reg Clk;
wire [31:0] Out_DataB;
wire [31:0] Out_SE;
wire [2:0] Out_Funct;
wire [4:0] Out_WriteAddr;
wire [31:0] Out_ALUResult, Out_ALUData;
wire [4:0] Out_Rs, Out_Rt, Out_Rd;
wire [31:0] Out_MEMAddress, Out_MEMData;
wire [31:0] Out_Data;
```

```
wire [31:0] Out_PC;
wire [31:0] Out_Instr;
wire [31:0] Out_Branch_PC, Out_Jump_PC;
wire Out_PCSrc, Out_Jump;
wire [31:0] Out_DataA;
wire Out_RegWrite;
wire [3:0] Out_EXControl1;
wire [1:0] Out_MEMControl1, Out_WBControl1, Out_MEMControl2, Out_WBControl2
    , Out_WBControl3;

MIPSProcessor myMIPSProcessor(Clk, Out_PC, Out_Instr, Out_Branch_PC,
    Out_Jump_PC, Out_PCSrc, Out_Jump, Out_DataA, Out_DataB, Out_SE,
    Out_Funct, Out_WriteAddr, Out_ALUResult, Out_ALUData, Out_Rs, Out_Rt,
    Out_Rd, Out_MEMAddress, Out_MEMData, Out_Data, Out_RegWrite,
    Out_EXControl1, Out_MEMControl1, Out_WBControl1, Out_MEMControl2,
    Out_WBControl2, Out_WBControl3);

initial begin
        Clk<=1'b0;
        #1 $display("Clk=%d, Out_PC=%d, Out_Instr=%d, Out_Branch_PC=%d,
            Out_PCSrc=%d, Out_DataA=%d, Out_DataB=%d, Out_SE=%d, Out_Funct
            =%d", Clk, Out_PC, Out_Instr, Out_Branch_PC, Out_PCSrc,
            Out_DataA, Out_DataB, Out_SE, Out_Funct);
        Clk<=1'b1;
        #1 $display("Clk=%d, Out_PC=%d, Out_Instr=%d, Out_Branch_PC=%d,
            Out_PCSrc=%d, Out_DataA=%d, Out_DataB=%d, Out_SE=%d, Out_Funct
            =%d", Clk, Out_PC, Out_Instr, Out_Branch_PC, Out_PCSrc,
            Out_DataA, Out_DataB, Out_SE, Out_Funct);
        Clk<=1'b0;
        #1 $display("Clk=%d, Out_PC=%d, Out_Instr=%d, Out_Branch_PC=%d,
            Out_PCSrc=%d, Out_DataA=%d, Out_DataB=%d, Out_SE=%d, Out_Funct
            =%d", Clk, Out_PC, Out_Instr, Out_Branch_PC, Out_PCSrc,
            Out_DataA, Out_DataB, Out_SE, Out_Funct);
end

endmodule
```

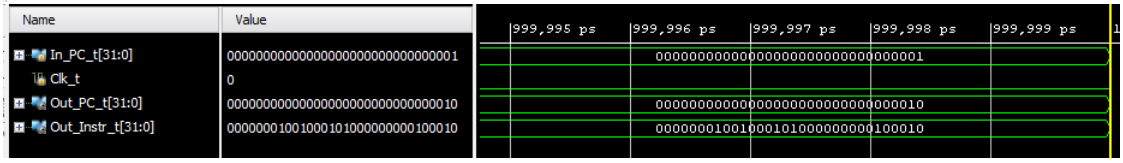
```
Clk=0, Out_PC=         1, Out_Instr=   4737056, Out_Branch_PC=     73857, Out_PCSrc=0, Out_DataA=         x, Out_DataB=         x, Ou
       x, Out_Funct=x
Clk=1, Out_PC=         1, Out_Instr=   4737056, Out_Branch_PC=     73857, Out_PCSrc=0, Out_DataA=         x, Out_DataB=         x, Ou
       x, Out_Funct=x
Clk=0, Out_PC=         2, Out_Instr=   4737056, Out_Branch_PC=     73858, Out_PCSrc=0, Out_DataA=         0, Out_DataB=4294967295, Ou
   18464, Out_Funct=2
```

## 6.2 IF

```
`timescale 1ns / 1ps
module IF(PC_In,Clk,PC_Out,Instr_Out);
output [31:0] PC_Out;
output [31:0] Instr_Out;
input [31:0] PC_In;
input Clk;
InstructionMemory IM(PC_In,Clk,Instr_Out);
assign PC_Out=PC_In+32'd 1;

endmodule
```

### 6.2.1 WF/TB

```
`timescale 1ns / 1ps
module IF_tb();
reg [31:0] In_PC_t;
reg Clk_t;
wire [31:0] Out_PC_t;
wire [31:0] Out_Instr_t;

IF myIF(In_PC_t,Clk_t,Out_PC_t,Out_Instr_t);

initial begin
In_PC_t<=32'd 1; Clk_t<=1'b0;
#1 $display("In_PC = %b, Clk = %d, Out_PC = %b, Out_Instr = %b", In_PC_t, Clk_t, Out_PC_t, Out_Instr_t);
end

endmodule
```

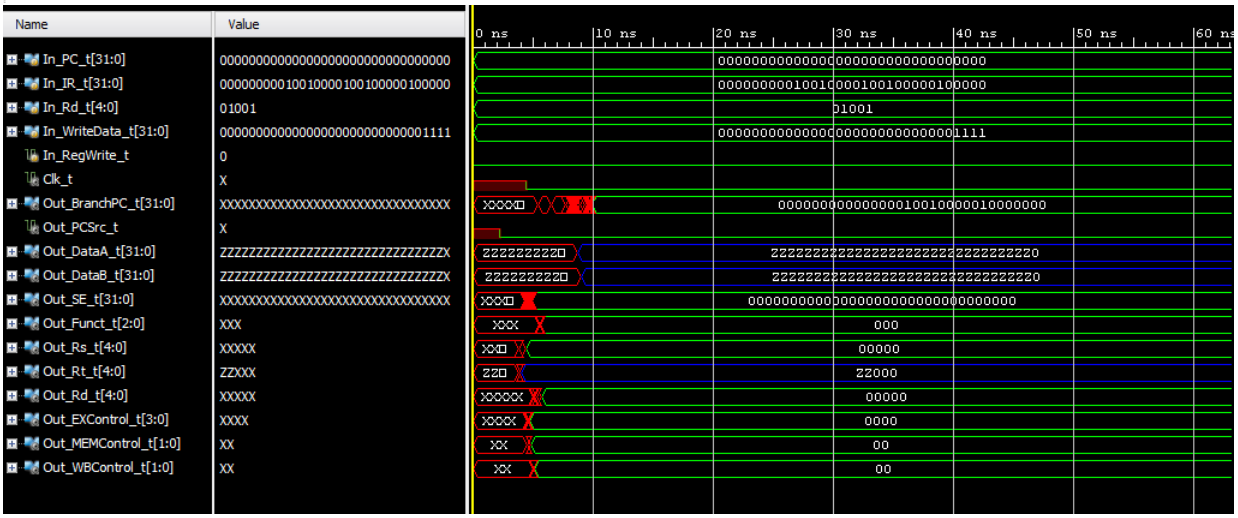


## 6.3 ID

```
wire [31:0] DataA, DataB, SE;
wire PCSrc;
wire ALUSrc;
wire [1:0] ALUOp;
wire RegDst, MemWrite, MemRead, Beq, Bne, MemToReg, RegWrite;
wire [2:0] Funct;
wire [3:0] EXControl;
wire [1:0] MEMControl, WBControl;

RegisterFile RF(In_IR[25:21],In_IR[20:16],In_Rd,In_WriteData,In_RegWrite,Clk,DataA,DataB);
//module Control(opcode, funct, ALUSrc, RegDst, MemWrite, MemRead, Beq, Bne, Jump, MemToReg, RegWrite, ALUControl);
Control Ctrl(In_IR[31:26],In_IR[5:0],ALUSrc,ALUOp,RegDst,MemWrite,MemRead,Beq, Bne, Jump, MemToReg,RegWrite,Funct);
SignExtension SE1(In_IR[15:0],SE);

assign EXControl={ALUSrc,ALUOp[1:0],RegDst};
assign MEMControl={MemWrite,MemRead};
assign WBControl={MemToReg,RegWrite};


assign Out_BranchPC=In_PC+(SE<<2);  // Branch target address computation
assign Out_JumpPC={In_PC[31:28],(In_IR[25:0]<<2)};

assign Out_PCSrc=(((Beq==1)&(DataA==DataB))|((Bne==1)&(DataA!=DataB)));
assign Out_Jump=Jump;

// Assign the outputs at the negative edge of the clock.
// You do not want the outputs to settle after the positive edge.
always@(negedge Clk) begin
      Out_DataA=DataA;
      Out_DataB=DataB;

      Out_SE=SE;
      Out_Rs=In_IR[25:21];
      Out_Rt=In_IR[20:16];
      Out_Rd=In_IR[15:11];

      Out_EXControl=EXControl;
      Out_MEMControl=MEMControl;
      Out_WBControl=WBControl;
      Out_Funct=Funct;
end
endmodule
```

### 6.3.1 WF/TB

```
wire [31:0] Out_DataA_t; // Read Register A
wire [31:0] Out_DataB_t; // Read Register B
wire [31:0] Out_SE_t; // Sign extended value
wire [2:0] Out_Funct_t; // 6 bit funct field
wire [31:0] Out_BranchPC_t; // Branch target address or jump address

wire [4:0] Out_Rt_t; // Rt register address In_IR[20:16];
wire [4:0] Out_Rd_t; // Rd register address In_IR[15:11];
wire [4:0] Out_Rs_t; // Rs register address In_IR[25:21]; for use in forwarding logic


wire [3:0] Out_EXControl_t; // Control signals for EX stage
// ([3] ALUSrc, [2:1] ALUOp, [0] RegDst);
wire [1:0] Out_MEMControl_t; // Control signals for MEM stage
// ([1] MemWrite, [0] MemRead);
wire [1:0] Out_WBControl_t; // Control signals for WB stage.
// ([1] MemToReg, [0] RegWrite)

ID myID(In_PC_t, In_IR_t, In_Rd_t, In_WriteData_t, In_RegWrite_t,
Out_BranchPC_t, Out_PCSrc_t, Out_DataA_t, Out_DataB_t, Out_SE_t,
Out_Funct_t, Out_Rs_t, Out_Rt_t, Out_Rd_t, Out_EXControl_t,
Out_MEMControl_t, Out_WBControl_t, Clk_t);

initial begin

In_PC_t <= 32'h 00000000; // Program counter
In_IR_t <= 32'b 00000000010010000100100000100000; // Instruction register.
In_Rd_t <= 5'b 01001; // Address for writing
In_WriteData_t <= 32'd 15; // Data to be written
In_RegWrite_t <= 1'b 0; // Control line for writing into register
Clk_t <= 1'b 0;

#1 $display( "Out_DataA = %b, Out_DataB = %b,  In_IR[25:21] = %b, In_IR_t[20:16] = %b, In_Rd_t = %b, In_WriteData_t = %b, In_RegWrite_t

end
endmodule
```

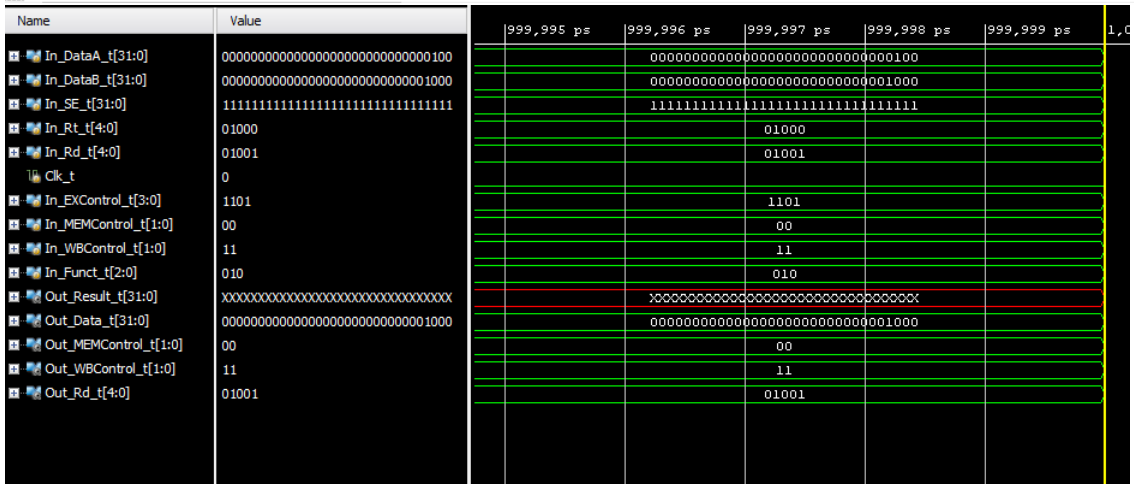


## 6.4 EX

```
module EX(DataA_In, DataB_In, SE_In, Rt_In, Rd_In, EXControl_In, MEMControl_In, WBControl_In,
Funct_In, Result_Out, Data_Out, Rd_Out, MEMControl_Out, WBControl_Out, Clk);

    input [3:0] EXControl_In;   // Control signals for EX stage
                                // ([3] ALUSrc, [2:1] ALUOp, [0] RegDst);
    input [1:0] MEMControl_In; // Control signals for MEM stage
                                // ([1] MemRead, [0] MemWrite);
    input [1:0] WBControl_In;   // Control signals for WB stage.
                                // ([1] MemToReg, [0] RegWrite)
    input [2:0] Funct_In;       // 3-bit ALU control
    input Clk;
    input [31:0] DataA_In;      // Data A input
    input [31:0] DataB_In;      // Data B input
    input [31:0] SE_In;         // value (Sign extended)
    input [4:0] Rt_In;
    input [4:0] Rd_In;



    output [31:0] Result_Out;   // Result from ALU; output for address for MEM
    output [31:0] Data_Out;     // Output for write data for MEM
    output [1:0] MEMControl_Out;    // Control signals for MEM stage
    output [1:0] WBControl_Out; // Control signals for WB stage
    output [4:0] Rd_Out;

    reg [31:0] DataB;               // Data to go into second input of ALU (either In_DataB or In_SE)
    reg [31:0] Data_Out;
    reg [31:0] Result_Out;
    reg [1:0] MEMControl_Out;
    reg [1:0] WBControl_Out;
    reg [4:0] Rd_Out;

    wire set, zero, overflow;
    wire [31:0] result;

    always @(EXControl_In) begin
        if (EXControl_In[0]==0) begin
                Rd_Out=Rt_In;
            end
            else if (EXControl_In[0]==1) begin
                 Rd_Out=Rd_In;
            end
        if (EXControl_In[3]==0) begin
             DataB=DataB_In;
        end
        else if (EXControl_In[3]==1) begin
             DataB=SE_In;
        end
    end
    always@(negedge Clk) begin
        Result_Out=result;
         Data_Out=DataB_In;
         MEMControl_Out=MEMControl_In;
         WBControl_Out=WBControl_In;
    end
    ALU32Bit myALU32Bit(In_DataA,DataB,In_Funct,result,set,zero,overflow);
endmodule
```

### 6.4.1 TB/WF

```
EX myEX(In_DataA_t, In_DataB_t, In_SE_t, In_Rt_t, In_Rd_t, In_EXControl_t, In_MEMControl_t, In_WBControl_t, In_Funct_t, Out_Result_t,
 Out_Data_t, Out_Rd_t, Out_MEMControl_t, Out_WBControl_t, Clk_t);

initial begin
    In_DataA_t<=32'd 4;
    In_DataB_t<=32'd 8;
    In_SE_t<=32'h ffffffff;
    In_Rt_t<=5'd8;
    In_Rd_t<=5'd9;
    In_EXControl_t<=4'b1101;
    In_MEMControl_t<=2'b00;
    In_WBControl_t<=2'b11;
    In_Funct_t<=3'b010;
    Clk_t<=1'b0;

    #1 $display( "In_DataA = %b, In_DataB = %b, In_SE = %b, In_Rt = %d, In_Rd = %d, Clk = %b, In_EXControl = %b, In_MEMControl = %b, In_
end
endmodule
```

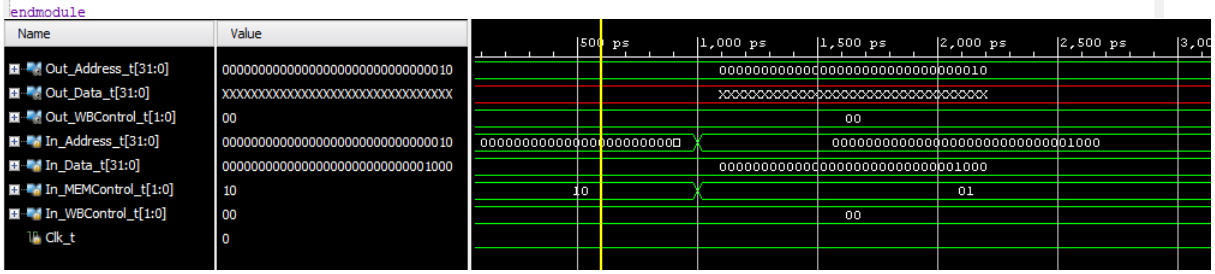


## 6.5 MEM

```
`timescale 1ns / 1ps
module MEM(Address_In, Data_In, MEMControl_In, WBControl_In, Address_Out, Data_Out, WBControl_Out, Clk);
input [31:0] Address_In;
input [31:0] Data_In;
input [1:0] WBControl_In;
input Clk;
input [1:0] MEMControl_In; // Control signals for MEM stage ([1] MemRead, [0] MemWrite);
output [31:0] Address_Out;
output [31:0] Data_Out;
output [1:0] WBControl_Out;

reg [31:0] Address_Out;
reg [31:0] Data_Out;
reg [1:0] WBControl_Out;

wire [31:0] Data;
wire [6:0] Address;

assign Address=Address_In[6:0];

DataMemory myDataMemory(Address, Data_In, MEMControl_In[1], MEMControl_In[0], Clk, Data);

always@(negedge Clk) begin
     Address_Out=Address_In;
     Data_Out=Data;
     WBControl_Out=WBControl_In;
end

endmodule
```

### 6.5.1 WF/TB

```
`timescale 1ns / 1ps
module MEM_tb();
wire [31:0] Out_Address_t;
wire [31:0] Out_Data_t;
wire [1:0] Out_WBControl_t;
reg [31:0] In_Address_t;
reg [31:0] In_Data_t;
reg [1:0] In_MEMControl_t;
reg [1:0] In_WBControl_t;
reg Clk_t;


MEM myMEM(In_Address_t, In_Data_t, In_MEMControl_t, In_WBControl_t, Out_Address_t, Out_Data_t, Out_WBControl_t, Clk_t);

initial begin
In_MEMControl_t<=2'b10;
In_WBControl_t<=2'b00;
In_Address_t<=32'd2;
In_Data_t<=32'd8;
Clk_t<=1'b0;
#1 $display("In_Address=%b, In_Data=%b, In_MEMControl=%b, In_WBControl=%b, Out_Address=%b, Out_Data=%b, Out_WBControl=%b, Clk=%b", In_Ad

In_Data_t<=32'd8;
In_MEMControl_t<=2'b01;
In_WBControl_t<=2'b00;
In_Address_t<=32'd8;
Clk_t<=1'b0;
#1 $display("In_Address=%b, In_Data=%b, In_MEMControl=%b, In_WBControl=%b, Out_Address=%b, Out_Data=%b, Out_WBControl=%b, Clk=%b", In_Ad
end

endmodule
```

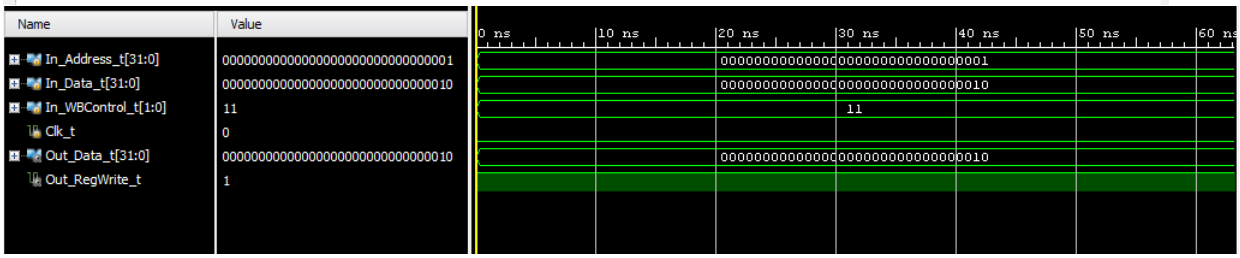


## 6.6 WB

```
`timescale 1ns / 1ps
module WB(Address_In, Data_In, WBControl_In, Data_Out, RegWrite_Out, Clk);
input Clk;
input [31:0] Address_In;
input [1:0] WBControl_In;
input [31:0] Data_In;

output [31:0] Data_Out;
output RegWrite_Out;
reg [31:0] Data_Out;
wire RegWrite;
reg RegWrite_Out;

always @(WBControl_In) begin
    if(WBControl_In[1]==1) begin
         Data_Out=Data_In;
    end
    else if(WBControl_In[1]==0) begin
         Data_Out=Address_In;
    end
end
assign RegWrite=WBControl_In[0];
always @(negedge Clk) begin
     RegWrite_Out=RegWrite;
end

endmodule
```

### 6.6.1 WF/TB

```
`timescale 1ns / 1ps
module WB_tb();
reg [31:0] In_Address_t;
reg [31:0] In_Data_t;
reg [1:0] In_WBControl_t; // Control signals for WB stage.
                          // ([1] MemToReg, [0] RegWrite)
reg Clk_t;

wire [31:0] Out_Data_t;
wire Out_RegWrite_t;

WB myWB(In_Address_t, In_Data_t, In_WBControl_t, Out_Data_t, Out_RegWrite_t, Clk_t);

initial begin
In_Address_t<=32'd1;
In_Data_t<=32'd2;
In_WBControl_t<=2'b11;
Clk_t<=1'b0;

#1 $display("In_Address = %b, In_Data = %b, In_WBControl = %b, Out_Data = %b, Out_RegWrite = %b, Clk = %b", In_Address_t, In_Data_t, In_
end

endmodule
```




### 6.7 Data Memory

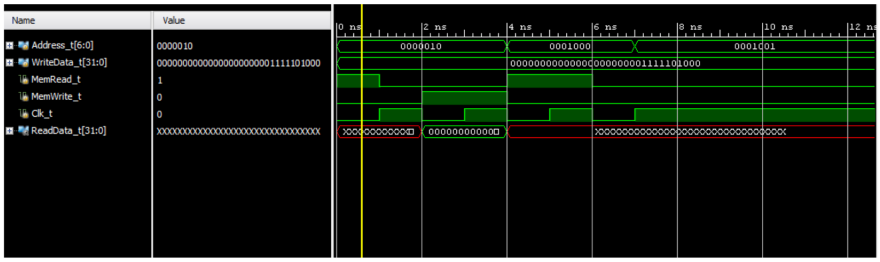
```
module DataMemory(Address, WriteData, MemRead, MemWrite, Clk,ReadData);
input [6:0] Address; // 7-bit address to memory.
input [31:0] WriteData; // Data to be written into WriteRegister
input MemRead; // Data in memory location Address is read if thiscontrol is
    set
input MemWrite; // WriteData is written in Address during positiveclock
    edge if this control is set
output [31:0] ReadData; // Value read from memory location Address
reg [31:0] ReadData;
input Clk;
reg [255:0] memFile[31:0];


initial @(Clk) begin
   memFile[0] <= 10;
end
```

```
always @(*) begin
        if (MemRead) begin
                ReadData = memFile[Address];
        end

        if (MemWrite) begin
                memFile[Address] = WriteData;
        end
end

endmodule
```

### 6.7.1 TB/WF

```
`timescale 1ns / 1ps
module DataMemory_tb();
reg [6:0] Address_t;                    // 7-bit address to memory.
reg [31:0] WriteData_t;  // Data to be written into WriteRegister
reg MemRead_t;                                  // Data in memory location Address
     is read if this control is set
reg MemWrite_t;                         // WriteData is written in Address during
    positive clock edge if this control is set
reg Clk_t;                                              // Clock input
wire [31:0] ReadData_t;  // Value read from memory location Address

DataMemory dmem(Address_t, WriteData_t, MemRead_t, MemWrite_t, Clk_t,
    ReadData_t);

initial
        begin
         Address_t <= 7'd 2; WriteData_t <= 32'd 1000; MemRead_t <= 1;
             MemWrite_t <= 0; Clk_t <= 0;
         #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
             %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
             MemRead_t, MemWrite_t, Clk_t, ReadData_t);

         Address_t <= 7'd 2; WriteData_t <= 32'd 1000; MemRead_t <= 0;
             MemWrite_t <= 0; Clk_t <= 1;
         #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
             %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
             MemRead_t, MemWrite_t, Clk_t, ReadData_t);

         Address_t <= 7'd 2; WriteData_t <= 32'd 1000; MemRead_t <= 0;
             MemWrite_t <= 1; Clk_t <= 0;
         #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
             %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
             MemRead_t, MemWrite_t, Clk_t, ReadData_t);
```

```
        Address_t <= 7'd 2; WriteData_t <= 32'd 1000; MemRead_t <= 0;
            MemWrite_t <= 1; Clk_t <= 1;
        #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
            %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
            MemRead_t, MemWrite_t, Clk_t, ReadData_t);

        Address_t <= 7'd 8; WriteData_t <= 32'd 1000; MemRead_t <= 1;
            MemWrite_t <= 0; Clk_t <= 0;
        #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
            %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
            MemRead_t, MemWrite_t, Clk_t, ReadData_t);

       Address_t <= 7'd 8; WriteData_t <= 32'd 1000; MemRead_t <= 1;
           MemWrite_t <= 0; Clk_t <= 1;
        #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
            %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
            MemRead_t, MemWrite_t, Clk_t, ReadData_t);

       Address_t <= 7'd 8; WriteData_t <= 32'd 1000; MemRead_t <= 0;
           MemWrite_t <= 0; Clk_t <= 0;
        #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
            %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
            MemRead_t, MemWrite_t, Clk_t, ReadData_t);

        Address_t <= 7'd 9; WriteData_t <= 32'd 1000; MemRead_t <= 0;
            MemWrite_t <= 0; Clk_t <= 1;
        #1 $display ("Address = %d, WriteData = %d, MemRead = %d, MemWrite =
            %d, Clk = %d, ReadData = %d", Address_t, WriteData_t,
            MemRead_t, MemWrite_t, Clk_t, ReadData_t);

    end

endmodule
```



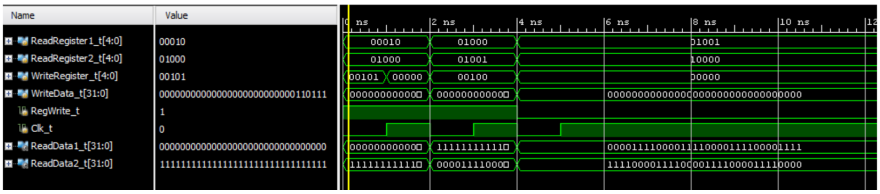

### 6.8 Instruction Memory

```
`timescale 1ns / 1ps
module InstructionMemory(PC_In, Clk, Instr_Out);
input Clk;
```

```
output [31:0] Instr_Out;
reg [31:0] ins[0:65536];
reg [31:0] Instr_Out;
input [31:0] PC_In;



initial begin
        ins[0]=32'b 00000000010010000100100000100000;
        ins[1]=32'b 00000001001000101000000000100010;
end
always@(negedge Clk) begin
         Instr_Out=ins[PC_In];
end

endmodule
```

### 6.8.1 TB/WF

```
`timescale 1ns / 1ps
module InstructionMemory(PC_In,Clk,Instr_Out);
input Clk;
output [31:0] Instr_Out;
reg [31:0] ins[0:65536];
reg [31:0] Instr_Out;
input [31:0] PC_In;



initial begin
        ins[0]=32'b 00000000010010000100100000100000;
        ins[1]=32'b 00000001001000101000000000100010;
end
always@(negedge Clk) begin
         Instr_Out=ins[PC_In];
end

endmodule
```



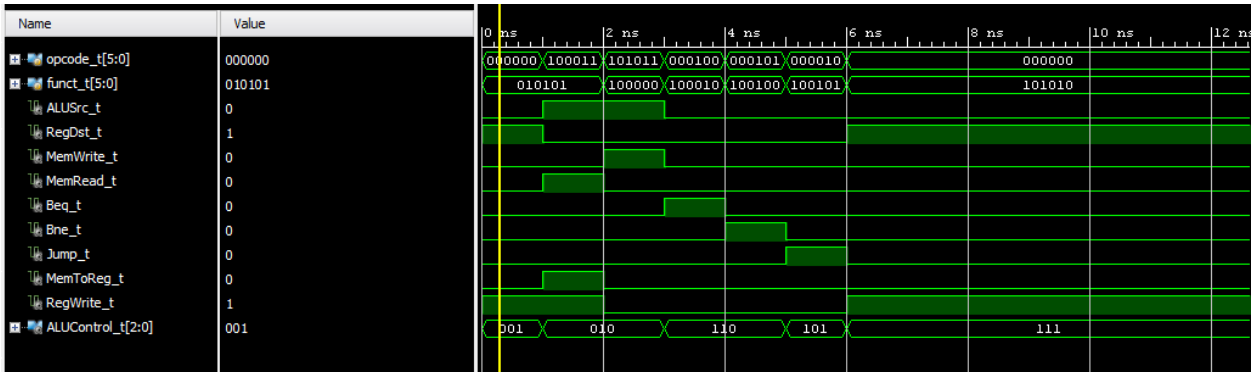

## 6.9 Control

```
`timescale 1ns / 1ps
module Control(opcode, funct, ALUSrc, ALUOp, RegDst, MemWrite, MemRead, Beq, Bne,
    Jump, MemToReg, RegWrite, ALUControl);
input [5:0] opcode;                 // 6-bit operation code
input [5:0] funct;                  // 6-bit function code from the instruction
                                                                    // least
                                                                        significant 6
                                                                         bits of an
                                                                        instruction
output ALUSrc, RegDst, MemWrite, MemRead, Beq, Bne, Jump, MemToReg, RegWrite; //
    Output control lines
output [1:0] ALUOp;
output [2:0] ALUControl; // 3-bit control for the ALU that specifies the operation

//wire [1:0] ALUOp;                 // 2-bit intermediate output for controlling ALU

ControlOld ControlOld(opcode, ALUSrc, ALUOp, RegDst, MemWrite, MemRead, Beq, Bne,
    Jump, MemToReg, RegWrite);
ALUOpToALUControl ALUOpToALUControl1(ALUOp, funct, ALUControl);

endmodule
```

### 6.9.1 TB/WF

```
module Control_tb();
reg [5:0] opcode_t;      // 6-bit operation code
reg [5:0] funct_t;           // 6-bit function code from the instruction
                             // least significant 6 bits of an instruction
wire ALUSrc_t, RegDst_t, MemWrite_t, MemRead_t, Beq_t, Bne_t, Jump_t, MemToReg_t, RegWrite_t; // Output control lines
wire [2:0] ALUControl_t; // 3-bit control for the ALU that specifies the operation
Control Control1(opcode_t, funct_t, ALUSrc_t, RegDst_t, MemWrite_t, MemRead_t, Beq_t, Bne_t, Jump_t, MemToReg_t, RegWrite_t, ALUControl_t);

 initial
 begin
    opcode_t <= 6'b 000000; funct_t <= 6'b 010101;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A

    opcode_t <= 6'b 100011; funct_t <= 6'b 010101;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A


    opcode_t <= 6'b 101011; funct_t <= 6'b 100000;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A


    opcode_t <= 6'b 000100; funct_t <= 6'b 100010;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A


    opcode_t <= 6'b 000101; funct_t <= 6'b 100100;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A


    opcode_t <= 6'b 000010; funct_t <= 6'b 0100101;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A


    opcode_t <= 6'b 000000; funct_t <= 6'b 101010;
    #1 $display ("opcode = %b, funct=%b, ALUSrc=%b, RegDst=%b, MemWrite=%b, MemRead=%b, Beq=%b, Bne=%b, Jump=%b, MemToReg=%b, RegWrite=%b, A
```

```
opcode = 000000, funct=010101, ALUSrc=0, RegDst=1, MemWrite=0, MemRead=0, Beq=0, Bne=0, Jump=0, MemToReg=0, RegWrite=1, ALUControl=001
opcode = 100011, funct=010101, ALUSrc=1, RegDst=0, MemWrite=0, MemRead=1, Beq=0, Bne=0, Jump=0, MemToReg=1, RegWrite=1, ALUControl=010
opcode = 101011, funct=100000, ALUSrc=1, RegDst=0, MemWrite=1, MemRead=0, Beq=0, Bne=0, Jump=0, MemToReg=0, RegWrite=0, ALUControl=010
opcode = 000100, funct=100010, ALUSrc=0, RegDst=0, MemWrite=0, MemRead=0, Beq=1, Bne=0, Jump=0, MemToReg=0, RegWrite=0, ALUControl=110
opcode = 000101, funct=100100, ALUSrc=0, RegDst=0, MemWrite=0, MemRead=0, Beq=0, Bne=1, Jump=0, MemToReg=0, RegWrite=0, ALUControl=110
opcode = 000010, funct=100101, ALUSrc=0, RegDst=0, MemWrite=0, MemRead=0, Beq=0, Bne=0, Jump=1, MemToReg=0, RegWrite=0, ALUControl=101
opcode = 000000, funct=101010, ALUSrc=0, RegDst=1, MemWrite=0, MemRead=0, Beq=0, Bne=0, Jump=0, MemToReg=0, RegWrite=1, ALUControl=111
```



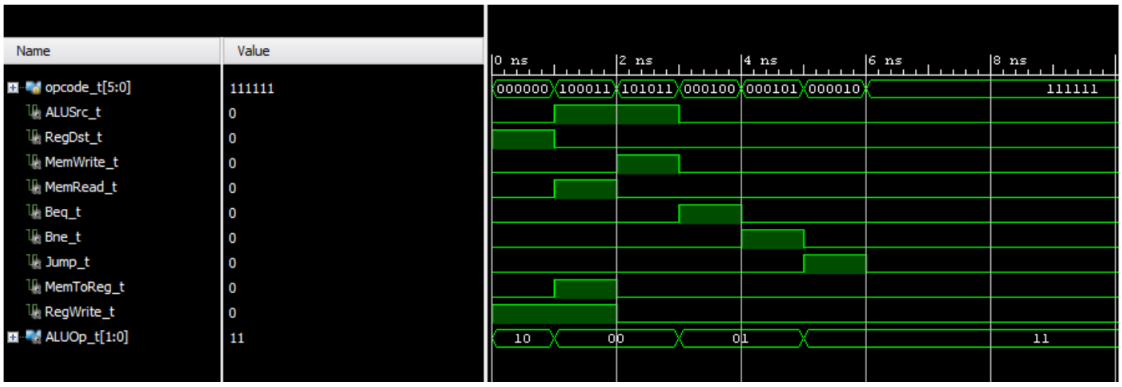
## 6.10 Control Old

```
module OldControl_tb();
reg [5:0] opcode_t; // 6-bit operation code
wire ALUSrc_t, RegDst_t, MemWrite_t, MemRead_t, Beq_t, Bne_t, Jump_t, MemToReg_t, RegWrite_t; // Output control lines
wire [1:0] ALUOp_t; // 2-bit intermediate output for controlling ALU

ControlOld ctlr(opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRead_t, Beq_t,Bne_t, Jump_t, MemToReg_t, RegWrite_t);

initial
begin
// r-format
opcode_t <= 6'b 000000;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

// lw
opcode_t <= 6'b 100011;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

// sw
opcode_t <= 6'b 101011;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

// beq
opcode_t <= 6'b 000100;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

// bne
opcode_t <= 6'b 000101;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

// jump
opcode_t <= 6'b 000010;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

 //invalad input; should produce all zeros
opcode_t <= 6'b 111111;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRe

end

endmodule
```

Output

```
# run 1000ns
opcode = 000000, ALUSrc = 0, ALUOp = 10, RegDst = 1, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 1
opcode = 100011, ALUSrc = 1, ALUOp = 00, RegDst = 0, MemWrite = 0, MemRead = 1, Beq = 0,Bne = 0, Jump = 0, MemToReg = 1, RegWrite = 1
opcode = 101011, ALUSrc = 1, ALUOp = 00, RegDst = 0, MemWrite = 1, MemRead = 0, Beq = 0,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 0
opcode = 000100, ALUSrc = 0, ALUOp = 01, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 1,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 0
opcode = 000101, ALUSrc = 0, ALUOp = 01, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 1, Jump = 0, MemToReg = 0, RegWrite = 0
opcode = 000010, ALUSrc = 0, ALUOp = 11, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 0, Jump = 1, MemToReg = 0, RegWrite = 0
opcode = 111111, ALUSrc = 0, ALUOp = 11, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 0
```

Waveform



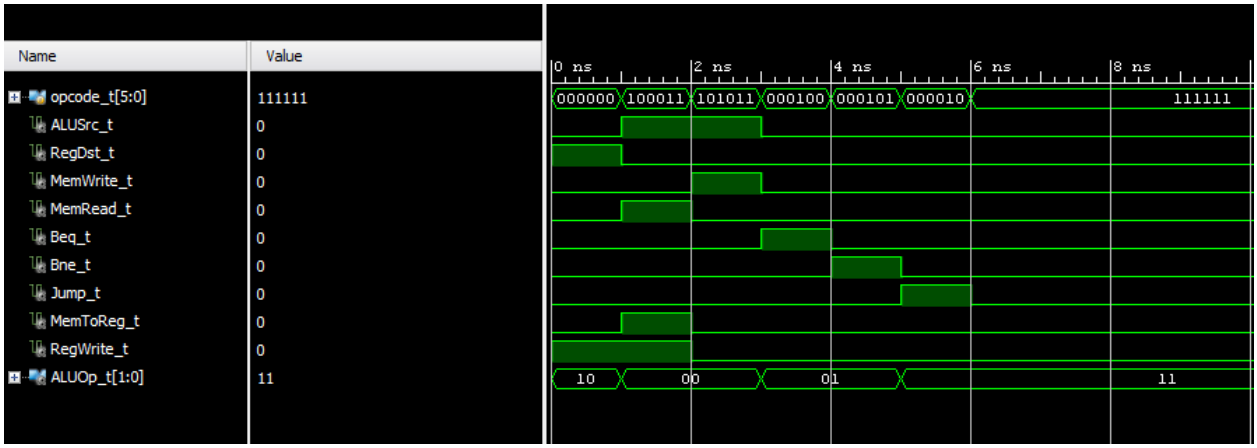

### 6.10.1 TB/WF

```
module OldControl_tb();
reg [5:0] opcode_t; // 6-bit operation code
wire ALUSrc_t, RegDst_t, MemWrite_t, MemRead_t, Beq_t, Bne_t, Jump_t, MemToReg_t, RegWrite_t; // Output control lines
wire [1:0] ALUOp_t; // 2-bit intermediate output for controlling ALU

ControlOld ctlr(opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRead_t, Beq_t,Bne_t, Jump_t, MemToReg_t, RegWrite_t);

initial
begin
// r-format
opcode_t <= 6'b 000000;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

// lw
opcode_t <= 6'b 100011;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

// sw
opcode_t <= 6'b 101011;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

// beq
opcode_t <= 6'b 000100;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

// bne
opcode_t <= 6'b 000101;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

// jump
opcode_t <= 6'b 000010;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

 //invalad input; should produce all zeros
opcode_t <= 6'b 111111;
#1 $display ("opcode = %b, ALUSrc = %b, ALUOp = %b, RegDst = %b, MemWrite = %b, MemRead = %b, Beq = %b,Bne = %b, Jump = %b, MemToReg = %b, RegWrite = %b", opcode_t, ALUSrc_t, ALUOp_t, RegDst_t, MemWrite_t, MemRea

end

endmodule
```

## Output

```
# run 1000ns
opcode = 000000, ALUSrc = 0, ALUOp = 10, RegDst = 1, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 1
opcode = 100011, ALUSrc = 1, ALUOp = 00, RegDst = 0, MemWrite = 0, MemRead = 1, Beq = 0,Bne = 0, Jump = 0, MemToReg = 1, RegWrite = 1
opcode = 101011, ALUSrc = 1, ALUOp = 00, RegDst = 0, MemWrite = 1, MemRead = 0, Beq = 0,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 0
opcode = 000100, ALUSrc = 0, ALUOp = 01, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 1,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 0
opcode = 000101, ALUSrc = 0, ALUOp = 01, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 1, Jump = 0, MemToReg = 0, RegWrite = 0
opcode = 000010, ALUSrc = 0, ALUOp = 11, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 0, Jump = 1, MemToReg = 0, RegWrite = 0
opcode = 111111, ALUSrc = 0, ALUOp = 11, RegDst = 0, MemWrite = 0, MemRead = 0, Beq = 0,Bne = 0, Jump = 0, MemToReg = 0, RegWrite = 0
```

## Waveform



### 6.11 Register File

```
‘timescale 1ns / 1ps
module RegisterFile(ReadRegister1, ReadRegister2, WriteRegister,WriteData,
    RegWrite, Clk, ReadData1, ReadData2);
input [4:0] ReadRegister1, ReadRegister2; // Two registers to be read
input [4:0] WriteRegister; // Register address to write into
input [31:0] WriteData; // Data to be written into WriteRegister
input RegWrite; // RegWrite control signal. Data is written only whenthis signal
    is enabled
output [31:0] ReadData1, ReadData2;
input Clk;
reg [31:0] ReadData1, ReadData2;
reg [31:0] iamReg[0:31];

initial begin
        iamReg[8]=32'h FFFFFFFF;
                iamReg[2]=32'h 00000000;
                iamReg[16]=32'h F0F0F0F0;
                iamReg[9]=32'h 0F0F0F0F;
end

always @(negedge Clk) begin
        if(RegWrite) begin
           iamReg[WriteRegister] = WriteData ;
        end
         ReadData1 = iamReg[ReadRegister1] ;
         ReadData2 = iamReg[ReadRegister2] ;
end




endmodule
```

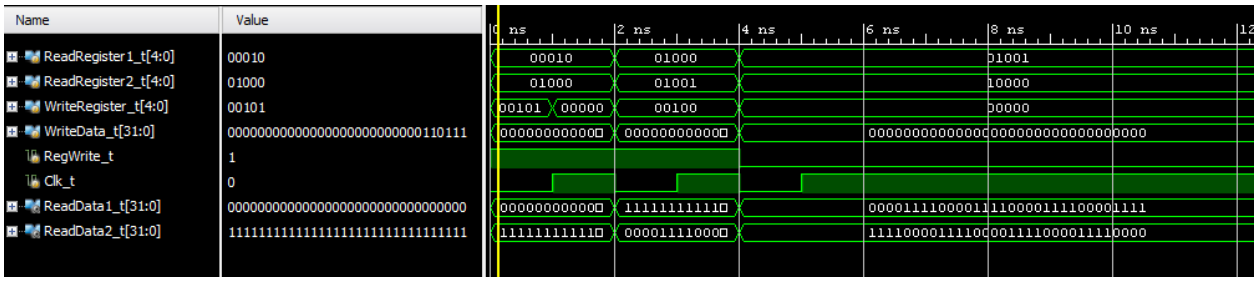
#### 6.11.1 TB/WF

```
module problem_c_tb();
reg [4:0] ReadRegister1_t, ReadRegister2_t;
reg [4:0] WriteRegister_t;                  // Register address to write into
reg [31:0] WriteData_t;                        // Data to be written into WriteRegister
reg RegWrite_t;                                   // RegWrite control signal. Data is written only when this signal is enabled
reg Clk_t;
wire [31:0] ReadData1_t, ReadData2_t;

RegisterFile register(ReadRegister1_t, ReadRegister2_t, WriteRegister_t, WriteData_t, RegWrite_t, Clk_t, ReadData1_t, ReadData2_t);

initial begin
Clk_t<=1'b0; ReadRegister1_t<=5'd2; ReadRegister2_t<=5'd8; WriteRegister_t<=5'd5; WriteData_t<=32'd55; RegWrite_t<=1;
#1 $display("ReadRegister1 = %d, ReadRegister2 = %d, WriteRegister = %d, WriteData = %d, RegWrite =%d, Clk_t = %b, ReadData1 = %d, ReadD

Clk_t<=1'b1; ReadRegister1_t<=5'd2; ReadRegister2_t<=5'd8; WriteRegister_t<=5'd0; WriteData_t<=32'd55; RegWrite_t<=1;
#1 $display("ReadRegister1 = %d, ReadRegister2 = %d, WriteRegister = %d, WriteData = %d, RegWrite =%d, Clk_t = %b, ReadData1 = %d, ReadD

Clk_t<=1'b0; ReadRegister1_t<=5'd8; ReadRegister2_t<=5'd9; WriteRegister_t<=5'd4; WriteData_t<=32'd6444; RegWrite_t<=1;
#1 $display("ReadRegister1 = %d, ReadRegister2 = %d, WriteRegister = %d, WriteData = %d, RegWrite =%d, Clk_t = %b, ReadData1 = %d, ReadD

Clk_t<=1'b1; ReadRegister1_t<=5'd8; ReadRegister2_t<=5'd9; WriteRegister_t<=5'd4; WriteData_t<=32'd6444; RegWrite_t<=1;
#1 $display("ReadRegister1 = %d, ReadRegister2 = %d, WriteRegister = %d, WriteData = %d, RegWrite =%d, Clk_t = %b, ReadData1 = %d, ReadD

Clk_t<=1'b0; ReadRegister1_t<=5'd9; ReadRegister2_t<=5'd16; WriteRegister_t<=5'd0; WriteData_t<=32'd0; RegWrite_t<=0;
#1 $display("ReadRegister1 = %d, ReadRegister2 = %d, WriteRegister = %d, WriteData = %d, RegWrite =%d, Clk_t = %b, ReadData1 = %d, ReadD

Clk_t<=1'b1; ReadRegister1_t<=5'd9; ReadRegister2_t<=5'd16; WriteRegister_t<=5'd0; WriteData_t<=32'd0; RegWrite_t<=0;
#1 $display("ReadRegister1 = %d, ReadRegister2 = %d, WriteRegister = %d, WriteData = %d, RegWrite =%d, Clk_t = %b, ReadData1 = %d, ReadD
```

## Output

```
ReadRegister1 =  2, ReadRegister2 =  8, WriteRegister =  5, WriteData =         55, RegWrite =1, Clk_t = 0, ReadData1 =          0, ReadData2 = 4294967295
ReadRegister1 =  2, ReadRegister2 =  8, WriteRegister =  0, WriteData =         55, RegWrite =1, Clk_t = 1, ReadData1 =          0, ReadData2 = 4294967295
ReadRegister1 =  8, ReadRegister2 =  9, WriteRegister =  4, WriteData =       6444, RegWrite =1, Clk_t = 0, ReadData1 = 4294967295, ReadData2 =  252645135
ReadRegister1 =  8, ReadRegister2 =  9, WriteRegister =  4, WriteData =       6444, RegWrite =1, Clk_t = 1, ReadData1 = 4294967295, ReadData2 =  252645135
ReadRegister1 =  9, ReadRegister2 = 16, WriteRegister =  0, WriteData =          0, RegWrite =0, Clk_t = 0, ReadData1 =  252645135, ReadData2 = 4042322160
ReadRegister1 =  9, ReadRegister2 = 16, WriteRegister =  0, WriteData =          0, RegWrite =0, Clk_t = 1, ReadData1 =  252645135, ReadData2 = 4042322160
```

## Waveform



### 6.12 ALU Op to Control

```
`timescale 1ns / 1ps

module ALUOpToALUControl(ALUOp, Funct, ALUControl);
input [1:0] ALUOp;       // 2-bit intermediate output for controlling ALU
input [5:0] Funct;       // 6-bit function code
output [2:0] ALUControl; // 3-bit output for controlling ALU based on ALUOp and function code

reg [2:0] ALUControl;
always@(ALUOp, Funct) begin
    ALUControl[0]=(ALUOp[1]&(Funct[0]|Funct[3]));
    ALUControl[1]=(~ALUOp[1]|~Funct[2]);
    ALUControl[2]=(ALUOp[0]|(ALUOp[1]&Funct[1]));
end

endmodule
```

Test Bench

```
module AluOpTestBench();
reg [1:0] ALUOp_t;
reg [5:0] Funct_t;
wire [2:0] ALUControl_t;

ALUOpToALUControl my_op(ALUOp_t, Funct_t, ALUControl_t);

 initial
 begin
    ALUOp_t <= 2'b 00; Funct_t <= 6'b 010101;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);

        ALUOp_t <= 2'b 01; Funct_t <= 6'b 010101;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);

    ALUOp_t <= 2'b 10; Funct_t <= 6'b 100000;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);

        ALUOp_t <= 2'b 10; Funct_t <= 6'b 100010;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);

        ALUOp_t <= 2'b 10; Funct_t <= 6'b 100100;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);

        ALUOp_t <= 2'b 10; Funct_t <= 6'b 100101;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);

        ALUOp_t <= 2'b 10; Funct_t <= 6'b 101010;
    #1 $display ( "ALUOp = %b, Funct = %b, ALUControl = %b", ALUOp_t, Funct_t, ALUControl_t);
 end
endmodule
```

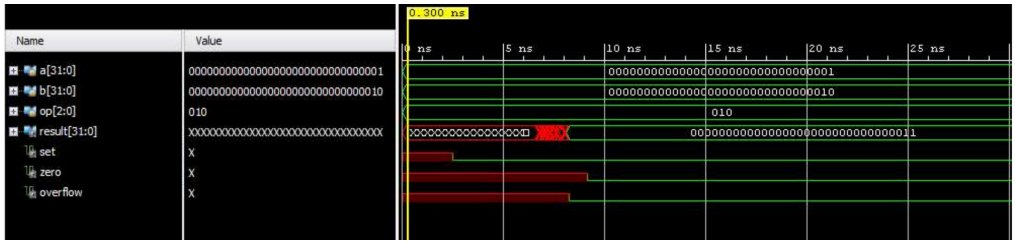

### 6.13 32 Bit ALU

```
module ALU32Bit(a, b, op, result, set, zero, overflow);
input [31:0] a, b;
input [2:0] op; // op[2] is "binv". op[1:0] denotes the 2-bit operation.
output [31:0] result;
output set; // set is the result of the most-significant ADDER unit.
output zero; // zero is 1 if the result is 0x0000. Otherwise, it is 0.
output overflow; // Overflow is 1 if the output is 0x00000000.
wire [7:0] g,p;
wire [1:0] over;

wire s0,s1, over0,over1;
wire [1:0] z;
wire cout0, cout1;
wire zero0, zero1;
//module ALU16Bit(   a,        b, *cin*, *less*, op,     result*cout*, *set*,
    zero,      g,     p, overflow);
ALU16Bit alu0( a[15:0],  b[15:0], op[2], s1 , op, result[15:0], cout0, s0 , zero0,
     g[3:0], p[3:0],  over0);
ALU16Bit alu1(a[31:16], b[31:16],  cout0,  1'b0, op,result[31:16], cout1,  s1,
    zero1, g[7:4], p[7:4],  over1);

assign overflow = cout1;
assign set = s1;
assign zero = zero0 & zero1;
endmodule


/*module ALU32Bit(a, b, op, result, set, zero, overflow);
input [31:0] a, b;
input [2:0] op; // op[2] is "binv". op[1:0] denotes the 2-bit operation.
output [31:0] result;
output set; // set is the result of the most-significant ADDER unit.
output zero; // zero is 1 if the result is 0x0000. Otherwise, it is 0.
output overflow; // Overflow is 1 if the output is 0x00000000.

wire cout;
wire z0, z1;
wire s;
wire dontCare;
```

```
ALU16Bit alu16_1(a[15:0], b[15:0], op[2], s, op, result[15:0], cout, dontCare, z0,
    G, P, overflow);
ALU16Bit alu16_2(a[31:16], b[31:16], cout, 0, op, result[31:16], cout, s, z1, G, P
    , overflow);

assign zero = z0 & z1;



endmodule
*/
```

### 6.13.1 TB/WV

```
module thiry_two_tb();
        reg [31:0] a, b;
        reg [2:0] op;
        wire [31:0] result;
        wire set, zero, overflow;

        ALU32Bit alu32(a, b, op, result, set, zero, overflow);

        initial
                begin
        a <= 32'd 21; b <= 32'd 21;
        op <= 3'b 010;
        #1 $display ("a = %d, b = %d, op = %d, result = %d, set = %d, zero = %d,
            overflow = %d", a, b, op, result, set, zero, overflow);
        end
endmodule
```




## 6.14 16 Bit ALU

```
module ALU16Bit(a, b, cin, less, op, result, cout, set, zero, g, p, overflow);
input [15:0] a, b;
input cin, less;
input [2:0] op;                                         // op[2] is "binv". op
    [1:0] denotes the 2-bit operation.
```

```
output [15:0] result;
output cout, set, zero, g, p, overflow;
// set is the result of the most-significant
// ADDER unit. zero is 1 if the result is 0x0000.
// Otherwise, it is 0.
wire [3:0] c;           // cout from each 4-bit ALU
wire g0, g1, g2,g3, p0, p1, p2, p3;     // G and P from each 4-bit ALU
wire [3:0] setv;        // set from each 4-bit ALU
wire [3:0] ovf; // overflow from each 4-bit ALU
wire [3:0] z;           // zero from each 4-bit ALU
wire C1, C2, C3, C4;


//module    FourBitALU(       a,       b, cin,    less, op,       result,  cout, G
    , P,     set, overflow, zero);
FourBitALU FourBitALU0(  a[3:0], b[3:0],    cin,    less, op,  result[3:0], c[0],
    g0, p0, setv[0], ovf[0], z[0]);
FourBitALU FourBitALU1(  a[7:4],  b[7:4],  C1,    1'b0, op,  result[7:4], c[1], g1
    , p1, setv[1], ovf[1], z[1]);
FourBitALU FourBitALU2( a[11:8], b[11:8],  C2,    1'b0, op, result[11:8], c[2], g2
    , p2, setv[2], ovf[2], z[2]);
FourBitALU FourBitALU3(a[15:12],b[15:12],  C3,    1'b0, op, result[15:12], c[3],
    g3, p3, setv[3], ovf[3], z[3]);

//module CLA(g0, p0, g1, p1, g2, p2, g3, p3, cin, C1, C2, C3, C4, G, P);
CLA CLA0(g0, p0, g1, p1, g2, p2, g3, p3, cin, C1, C2, C3, C4, g, p);

assign cout=C4;
assign set=setv[3];
assign zero=z[0]&z[1]&z[2]&z[3];
assign overflow=c[3];

endmodule
```

### 6.14.1 TB/WF

```
module ALU16Bitb();
    reg [15:0] a, b;
    reg cin, less;
    reg [2:0] op;
    wire [15:0] result;
    wire cout, set, zero, g, p, overflow;

        ALU16Bit  alu(a, b, cin, less, op, result, cout, set, zero, g, p, overflow
            );

        initial
                begin
        //Overflow!
```

```
        a <= 16'b 1111111111111111; b <= 16'b 1111111111111111; cin <= 0; less <=
            0; op <= 3'b010;
        #1 $display ( "Addition: a = %d, b = %d, cin = %d, op = %d, result = %d,
            cout = %d, set = %d, zero = %d, g = %d, p = &b, overflow = %d", a, b,
            cin, op, result, cout, set, zero, g, p, overflow);

        a <= 16'b 0000000001000000; b <= 16'b 0000000000000001; cin <= 0; less <=
            0; op <= 3'b010;
        #1 $display ("Addition: a = %d, b = %d, cin = %d, op = %d, result = %d,
            cout = %d, set = %d, zero = %d, g = %d, p = &b, overflow = %d", a, b,
            cin, op, result, cout, set, zero, g, p, overflow);

        a <= 16'b 0101010101010101; b <= 16'b 1010101010101010; cin <= 0; less <=
            0; op <= 3'b010;
        #1 $display ( "Addition 3 a = %d, b = %d, cin = %d, op = %d, result = %d,
            cout = %d, set = %d, zero = %d, g = %d, p = &b, overflow = %d", a, b,
            cin, op, result, cout, set, zero, g, p, overflow);

        a <= 16'b 100000100000001; b <= 16'b 1101001000000101;  cin <= 0; less <=
            0; op <= 3'b000;
        #1 $display ( "And: a = %b, b = %b, cin = %d, op = %d, result = %b, cout =
            %d, set = %d, zero = %d, g = %d, p = &b, overflow = %d", a, b, cin,
            less, op, result, cout, set, zero, g, p, overflow);


        a <= 16'b 000000000000000; b <= 16'b 0000000000000000;  cin <= 0; less <=
            0; op <= 3'b000;
        #1 $display ( "And: a = %b, b = %b, cin = %d, result = %b, cout = %d, set
            = %d, zero = %d, g = %d, p = &b, overflow = %d", a, b, cin,  result,
            cout, set, zero, g, p, overflow);



        a <= 16'b 1100000000000011; b <= 16'b 0000000000000000; cin <= 0; less <=
            0; op <= 3'b001;
        #1 $display ( "Or a = %b, b = %b, cin = %d, result = %b, cout = %d, set =
            %d, zero = %d, g = %d, p = &b, overflow = %d", a, b, cin, result, cout
            , set, zero, g, p, overflow);


                end
endmodule
```

### 6.15 4 Bit ALU

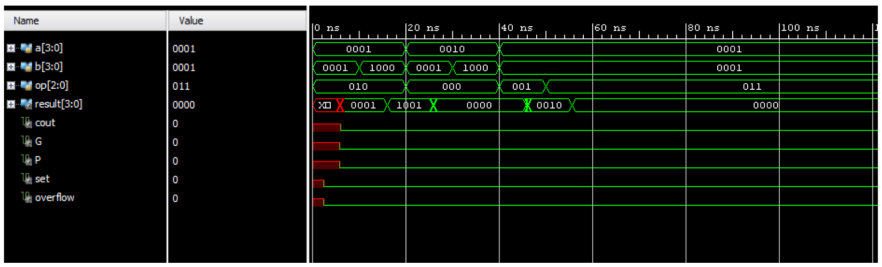
```
module FourBitALU(a, b, cin, less, op, result, cout, G, P, set, overflow, zero);
input [3:0] a, b;       // Inputs to the one-bit ALU.
input cin;                          // Carry-in bit of the ALU.
```

```
input less;
input [2:0] op;                    // 3-bit operation code. op[2] is the "binv". op
    [0] is the
                                                        // least significant bit.
output [3:0] result; // The result of the ALU (depends on the operation that
                                                        // is chosen)
output cout;                       // Carry-out bit of the ALU
output G, P;                       // Block generate and propagate of the four-bit
    ALU
output set;                        // This is the set output of the most significant
    ALU block
output overflow;                   // This bit indicates that an overflow has
    occurred.
                                                        // (Ignores what operation
                                                             is chosen for ALU)
output zero;                       // This bit indicates if the result is zero (1 if
    zero,
                                                        // 0 if non-zero)


wire [3:0] C;
wire [3:0] g,p;
wire [3:0] setv;
wire C1,C2,C3,C4;
CLA CLA0(g[0],p[0],g[1],p[1],g[2]    ,p[2],g[3],p[3],cin,C1,C2,C3,C4,G,P);

//module OneBitALU(     a,    b,cin,    less,op,    result,cout,    g,    p,     set);
OneBitALU onebitALU0(a[0],b[0],cin,    less,op,result[0],C[0],g[0],p[0],setv[0]);
OneBitALU onebitALU1(a[1],b[1], C1,    1'b0,op,result[1],C[1],g[1],p[1],setv[1]);
OneBitALU onebitALU2(a[2],b[2], C2,    1'b0,op,result[2],C[2],g[2],p[2],setv[2]);
OneBitALU onebitALU3(a[3],b[3], C3,1'b0,op,result[3],C[3],g[3],p[3],setv[3]);

OverflowDetection OverflowDetection1(C3,C4,overflow);

assign set=setv[3];
assign cout=C4;
assign zero=~(result[0]|result[1]|result[2]|result[3]);

endmodule
```

#### 6.15.1 TB/WF

```
module fourbit_bench();
 reg [3:0] a;
 reg [3:0] b;
 reg cin;
 reg less;
 reg [2:0] op;
 wire [3:0] result;
```

```
 wire cout, G, P, set, overflow, zero;

 //module FourBitALU (a, b, op, result, cout, G, P, set,
     overflow, zero);
FourBitALU fourbitalu1(a, b, cin, less, op, result, cout, G, P, set, overflow,
    zero);

initial begin
        a <= 4'b 0001; b <= 4'b 0001; op <= 3'b 010; cin <= 1'b0; less <= 1'b0;

        #1 $display("a = %d, b = %d, result = %d, cout = %b, G = %b, P = %b, set =
            %b, overflow = %b",a,b,result,cout,G,P,set,overflow) ;

        a <= 4'b 0100; b <= 4'b 0001; cin <= 1'b0; less <= 1'b0; op <= 3'b 110;
 #1 $display ("a = %d, b = %d, op = %d, result = %d, cout = %d, G = %d, P = %d,
     set = %d, overflow = %d", a, b, op, result, cout, G, P, set, overflow);


end
endmodule
```




### 6.16 CLA

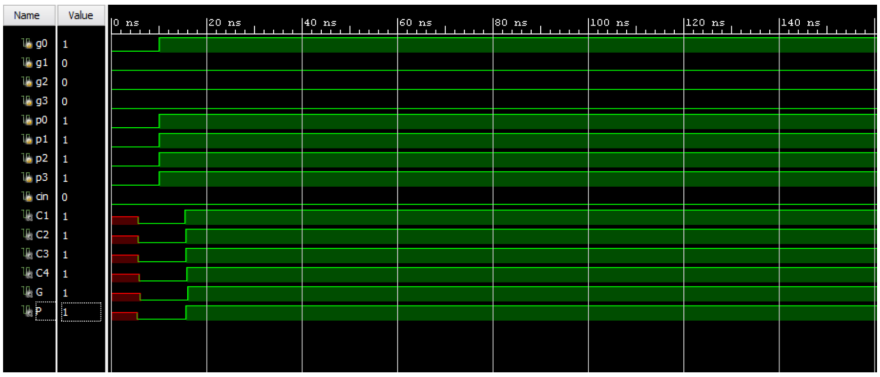
```
module CLA(g0, p0, g1, p1, g2, p2, g3, p3, cin, C1, C2, C3, C4, G, P);
input g0, p0, g1, p1, g2, p2, g3, p3; // Generate and propagate signals









input cin;
    // Carry-in input
```

```
output C1, C2, C3, C4;                                          // Carry bits computed
    by the CLA.
output G, P;                                                                    // Block
     generate and block propagate to be used by CLAs at a









reg C1, C2, C3, C4;
reg G, P;

always @ (g0, g1, g2, g3, p0, p1, p2, p3, cin)
begin
          C1 = (g0 | (p0 & cin));
          C2 = (g1 | (p1 & C1));
          C3 = (g2 | (p2 & C2));
          C4 = (g3 | (p3 & C3));

          G = (g3 | (g2 & p3) | (g1 & p3 & p2) |  (g0 & p3 & p2 & p1));
          P = (p3 & p2) & (p1 & p0);
end

endmodule
```

### 6.16.1 TB/WF

```
module p3a_ttb(); //test bench module
        reg cin_t ,g0_t, g1_t, g2_t, g3_t, p0_t, p1_t, p2_t, p3_t; //input
            registers
   wire G_t, P_t, C1_t, C2_t, C3_t, C4_t;

        CLA CLA_tst(g0_t, p0_t, g1_t, p1_t, g2_t, p2_t, g3_t, p3_t, cin_t,  C1_t,
            C2_t, C3_t, C4_t, G_t, P_t);

        initial
        begin
        // Case : all p's and g's are 1
        g0_t <= 0;
        p0_t <= 0;
        g1_t <= 0;
        p1_t <= 0;
        g2_t <= 0;
        p2_t <= 0;
        g3_t <= 0;
        p3_t <= 0;
```

```
        cin_t <= 0;
        #1 $display("C1_t = %b, C2_t = %b, C3_t = %b, C4_t = %b, G_t = %b, P_t = %b"
            , C1_t, C2_t, C3_t, C4_t, G_t, P_t);

        // Case: all p's and g's are 0
        g0_t <= 1;
        p0_t <= 1;
        g1_t <= 1;
        p1_t <= 1;
        g2_t <= 1;
        p2_t <= 1;
        g3_t <= 1;
        p3_t <= 1;
        cin_t <= 1;
        #1 $display("C1_t = %b, C2_t = %b, C3_t = %b, C4_t = %b, G_t = %b, P_t = %b"
            , C1_t, C2_t, C3_t, C4_t, G_t, P_t);

        // Case: Misc 1
        g0_t <= 1;
        p0_t <= 1;
        g1_t <= 1;
        p1_t <= 1;
        g2_t <= 0;
        p2_t <= 0;
        g3_t <= 0;
        p3_t <= 0;
        cin_t <= 0;
        #1 $display("C1_t = %b, C2_t = %b, C3_t = %b, C4_t = %b, G_t = %b, P_t = %b"
            , C1_t, C2_t, C3_t, C4_t, G_t, P_t);

    // Case: Misc 2
        g0_t <= 0;
        p0_t <= 0;
        g1_t <= 0;
        p1_t <= 0;
        g2_t <= 0;
        p2_t <= 0;
        g3_t <= 1;
        p3_t <= 1;
        cin_t <= 1;
        #1 $display("C1_t = %b, C2_t = %b, C3_t = %b, C4_t = %b, G_t = %b, P_t = %b"
            , C1_t, C2_t, C3_t, C4_t, G_t, P_t);



    end
endmodule
```




### 6.17 1 Bit ALU

```
module OneBitALU(a, b, cin, less, op, result, cout, g, p, set);
input a, b, cin; // Inputs to the one-bit ALU, "cin" is the carry-in bit.
input [2:0] op;  // 3-bit operation code. op[2] is the "binv". op[0] is the
                                        // least significant bit.
input less;      // This input will be set as 0 for all ALUs but the one
                                        //corresponding to the most-significant
                                            bit.
output result;   // The result of the ALU (depends on the operation that is
                                        // chosen)
output cout;      // Carry out bit of the adder
output g, p;      // Generate and propagate signals that are to be used by the
                                        // CLA unit.
output set;       // This is the "sum" output of the full-adder.

reg result;
reg p, g;
reg cout;
reg bcomp;
reg set;
reg sum;

always @ (a or b or op or cin or less)
        begin
                if (op[2] ==1)
                        begin
                                bcomp = ~b;
                        end
                else if (op[2]==0)
                        begin
                                bcomp = b;
                        end
                 sum = (( a & bcomp) & cin) | (( ~a & bcomp) & ~cin) | (( ~a & ~
                     bcomp) & cin) | ((a &~bcomp) & ~cin);
```

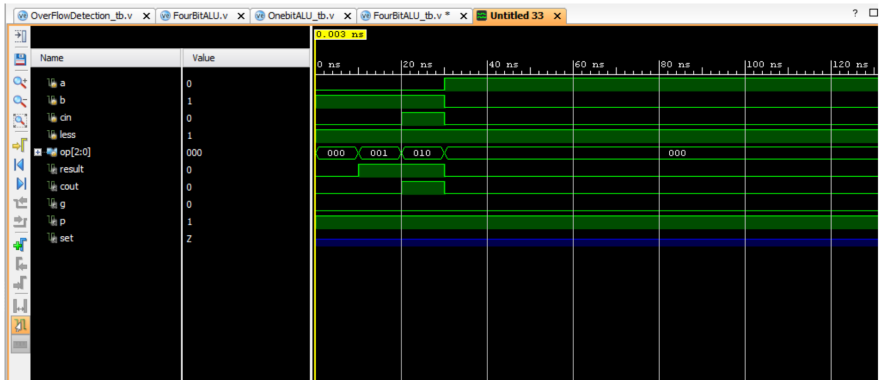
```
                cout = ((a & bcomp)| (a&cin) | (bcomp & cin));
                g = a & bcomp;
                p = a | bcomp;
                set = sum;
              case (op)
                      3'b 000:  result = a & bcomp;
                      3'b 001:  result = a | bcomp;
                      3'b 010:  result = sum;
                      3'b 110:  result = sum;
                      3'b 111:  result = less;
              endcase
        end


endmodule
```

### 6.17.1 TB/WF

```
module alu_tb();
reg a_t, b_t, cin_t, less_t;
reg [2:0] op_t; // 3-bit operation code. op[2] is the "binv". op[0] is the
 // least significant bit.
//reg less_t; // This input will be set as 0 for all ALUs but the one
 //corresponding to the most-significant bit.
wire result_t; // The result of the ALU (depends on the operation that is
 // chosen)
//wire cout_t; // Carry out bit of the adder
wire g_t, p_t, set_t; // Generate and propagate signals that are to be used by the
 // CLA unit.


 OneBitALU alu(a_t, b_t, cin_t, less_t, op_t, result_t, cout_t, g_t, p_t, set_t);
 initial
 begin
 //case 1
 // a = 0
 // b = 0
 // c = 0
 // l = 1
 // op= 000
 a_t <= 0; b_t <= 0; cin_t <=0; op_t <= 3'b 000; less_t <=1;
 #1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
     = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);

  //case 2
 // a = 1
 // b = 0
 // c = 0
 // l = 1
```

```
// op= 000
a_t <= 1; b_t <= 0; cin_t <=0; op_t <= 3'b 000; less_t <=1;
#1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
    = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);

 //case 3
// a = 0
// b = 0
// c = 0
// l = 1
// op= 100
a_t <= 0; b_t <= 0; cin_t <=0; op_t <= 3'b 100; less_t <=1;
#1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
    = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);

 //case 4
// a = 1
// b = 1
// c = 1
// l = 1
// op= 000
a_t <= 1; b_t <= 1; cin_t <=1; op_t <= 3'b 000; less_t <=1;
#1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
    = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);


 //case 5
// a = 1
// b = 1
// c = 1
// l = 1
// op= 001
a_t <= 1; b_t <= 1; cin_t <=1; op_t <= 3'b 001; less_t <=1;
#1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
    = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);


 //case 6
// a = 1
// b = 1
// c = 1
// l = 1
// op= 010
a_t <= 1; b_t <= 1; cin_t <=0; op_t <= 3'b 010; less_t <=1;
#1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
    = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);


 //case 7
// a = 1
```

```
 // b = 1
 // c = 1
 // l = 1
 // op= 011
 a_t <= 1; b_t <= 1; cin_t <=1; op_t <= 3'b 011; less_t <=1;
 #1 $display ("a = %b, b = %b, cin = %b, less= %b, op = %b, g = %b, p = %b, result
     = %b, set = %b", a_t, b_t,cin_t,less_t, op_t, g_t, p_t, result_t, set_t);

 end
endmodule
```



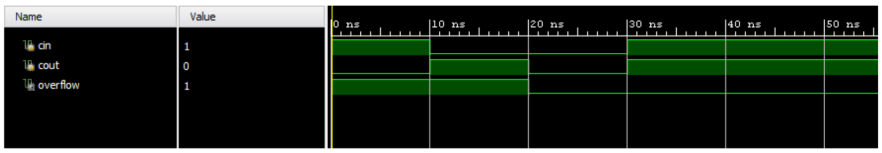

### 6.18 Overflow

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:     10:21:42 11/02/2016
// Design Name:
// Module Name:     OverFlowDection
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module OverflowDetection(a,b,overflow);
input a,b;
```

```
output overflow;


assign overflow=a^b;

endmodule
```

### 6.18.1 TB/WB

```
module overflow_tb();
    reg cin, cout;
    wire overflow;

    OverflowDetection ov(cin,cout,overflow) ;

initial begin
    cin <= 1; cout <= 0;

    #10 $display("overflow = %d",overflow) ;

    cin <= 0; cout <= 1;

    #10 $display("overflow = %d",overflow) ;
    cin <= 0; cout <= 0;

    #10 $display("overflow = %d",overflow) ;

    cin <= 1; cout <= 1;

    #10 $display("overflow = %d",overflow) ;


end
```




[]

## References


1. David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.


### 6.19 Sign Extension

```
`timescale 1ns / 1ps
module SignExtension(a, result);
input [15:0] a;                              // 16-bit input
output [31:0] result;    // 32-bit output
reg [31:0] result;

always@(a)
  begin
  if(a[15]==1'b1)
    begin
    result={16'b1111111111111111,a};
    end
  else
      begin
      result={16'b0000000000000000,a};
      end
  end

endmodule
```

### 6.19.1 TB/WF

```
module SignExtension_tb();
        reg [15:0] a;
        wire [31:0] result;

        SignExtension ext(a, result);
        initial
                begin
                a <= 16'd 0;
                #1 $display ( "a = %b, result = %b",  a, result);

                a <= 16'd 23;
                #1 $display ( "a = %b, result = %b",  a, result);
                a <= 16'b 1111111111111111;
                #1 $display ( "a = %b, result = %b",  a, result);
        end
endmodule
```

```
a = 0000000000000000, result = 00000000000000000000000000000000
a = 0000000000010111, result = 00000000000000000000000000010111
a = 1111111111111111, result = 11111111111111111111111111111111
```

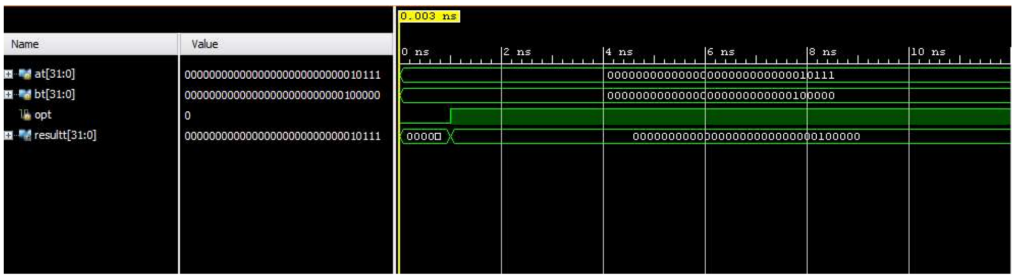
## 6.20 Mux 32 bit 2 to 1

```
module Mux32Bit2To1(a, b, op, result);
input [31:0] a, b;        // 32-bit inputs
```

```
input op;                                    // one-bit selection input
output [31:0] result; // 32-bit output
reg [31:0]result;
        wire [31:0] a;
        wire [31:0] b;
        wire op;

always@(a,b,op)
begin
if(op==1'b0)
  begin
   result=a;
  end
 else
  begin
   result=b;
 end
 end

endmodule
```

### 6.20.1 TB/WF

```
module mux32bit_tb();
reg [31:0] at, bt; // 32-bit inputs
    reg opt; // one-bit selection input
    wire [31:0] resultt; // 32-bit output

        Mux32Bit2To1 mymux(at, bt, opt, resultt);

initial begin

        at <= 32'd 23; bt <= 32'd 32 ; opt <= 0;
        #1 $display ( "a = %d, b = %d, op = %b, result = %d", at, bt, opt,
            resultt);

        at <= 32'd 23; bt <= 32'd 32; opt <= 1;
        #1 $display ( "a = %d, b = %d, op = %b result = %d", at, bt, opt, resultt
            );

end



endmodule
```




Output

```
a =         23, b =         32, op = 0, result =         23
a =         23, b =         32, op = 1 result =         32
```

### 6.21 Mux 5 Bit 2 to 1

```
module Mux5Bit2To1(a, b, op, result);
input [4:0] a, b; // 5-bit inputs
input op; // one-bit selection input
output [4:0] result; // 5-bit output

reg [4:0]result;
        wire [4:0] a;
        wire [4:0] b;
        wire op;

always@(op,a,b)
        begin
                if(op==1'b0)
                        begin
                                result=a;
                        end
                else
                        begin
                                result=b;
                        end
        end

endmodule
```

```
module mux5bitb();
reg [4:0] a, b; // 32-bit inputs
    reg op; // one-bit selection input
    wire [4:0] result; // 32-bit output

        Mux5Bit2To1 mymux(a, b, op, result);
        initial
```

```
        begin

                a <= 5'd 2; b <= 5'd 3; op <= 0;
                #1 $display ( "a = %d, b = %d, op = %b, result = %d", a,
                    b, op, result);

                a <= 5'd 2; b <= 5'd 3; op <= 1;
                #1 $display ( "a = %d, b = %d, op = %b, result = %d", a,
                    b, op, result);

        end



endmodule
```

### 6.22 Forarding

```
module Forward( MEMWB_Rd, EXMEM_Rd, IDEX_Rs, IDEX_Rt, MB_Write, EM_Write, A, B);
  input [4:0] MEMWB_Rd;
  input [4:0] EXMEM_Rd;
  input [4:0] IDEX_Rs;
  input [4:0] IDEX_Rt;
  input MB_Write;
  input EM_Write;
  output reg [1:0] A;
  output reg [1:0] B;

    always @(*) begin
      if ((MB_Write == 1'b 1) && (MEMWB_Rd != 5'b 00000) && (EXMEM_Rd !=IDEX_Rs)
          &&(MEMWB_Rd == IDEX_Rs ))
        A <= 2'b 01;
      else begin
        if ((EM_Write == 1'b 1) && (EXMEM_Rd != 5'b 00000) && (EXMEM_Rd == IDEX_Rs
            ))
          A <= 2'b 10;
        else
          A <= 2'b 00;
      end

      if ((MB_Write == 1'b 1) && (MEMWB_Rd != 5'b 00000) && (EXMEM_Rd != IDEX_Rt)
          && (MEMWB_Rd == IDEX_Rt))
        B <= 2'b 01;
      else begin
        if ((EM_Write == 1'b 1) && (EXMEM_Rd != 5'b 00000) && (EXMEM_Rd == IDEX_Rt
            ))
          B <= 2'b10;
        else
```

```
            B <= 2'b 00;
        end
    end
endmodule
```