EEE339 – Digital System Design with HDL

Assignment 2 Report

Student Name: *Rui Wang*

Student ID: *2033922*

Submission date: *16/12/2023*

Content

1 Basic Operation

1.1 Code

X	Y	Z	T
7	1	2	3

Table 1 Corresponding value of X, Y, Z, T

MIPS Instruction	[address] Machine Code (hexadecimal)
lw $7, 7$0	[0] 0x8C070007
lw $1, 1$0	[4] 0x8C010001
add $2, $7, $1	[8] 0x00E11020
sw $2, 2$0	[C] 0xAC020002
lw $3, 2$0	[10] 0x8C030002

Table 2 MIPS and corresponding machine code for Basic Operation.

Instruction				Instruction Mapping					
lw	$7,	7	$0	31-26	25-21	20-16	15-0		
opcode	rt	offset	rs	opcode	rs	rt	offset		
				100011	00000	00111	0000 0000 0000 0111		
lw	$1,	1	$0	31-26	25-21	20-16	15-0		
opcode	rt	offset	rs	opcode	rs	rt	offset		
				100011	00000	00001	0000 0000 0000 0001		
add	$2,	$7,	$1	31-26	25-21	20-16	15-11	10-6	5-0
funct	rd,	rs,	rt	opcode	rs	rt	rd	shamt	funct
				000000	00111	00001	00010	00000	100000
sw	$2,	2	$0	31-26	25-21	20-16	15-0		
opcode	rt	offset	rs	opcode	rs	rt	offset		
				101011	00000	00010	0000 0000 0000 0010		
lw	$3	2	$0	31-26	25-21	20-16	15-0		
opcode	rt	offset	rs	opcode	rs	rt	offset		
				100011	00000	00011	0000 0000 0000 0010		

Table 3 Instruction Mapping for Basic Operation.

1.2 Simulation

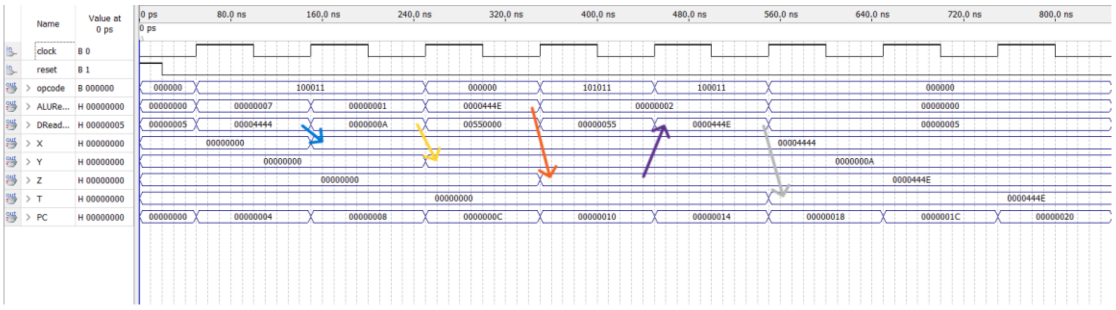

Name
Value at 0 ps
0 ps
80.0 ns
160.0 ns
240.0 ns
320.0 ns
400.0 ns
480.0 ns
560.0 ns
640.0 ns
720.0 ns
800.0 ns
clock B 0
reset B 1
opcode B 000000: 000000, 100011, 000000, 101011, 100011, 000000
ALURe... H 00000000: 00000000, 00000007, 00000001, 0000444E, 00000002, 00000000
DRead... H 00000005: 00000005, 00004444, 0000000A, 00550000, 00000055, 0000444E, 00000005
X H 00000000: 00000000, 00004444
Y H 00000000: 00000000, 0000000A
Z H 00000000: 00000000, 0000444E
T H 00000000: 00000000, 0000444E
PC H 00000000: 00000000, 00000004, 00000008, 0000000C, 00000010, 00000014, 00000018, 0000001C, 00000020

Figure 1 Simulation waveform for Basic Operation.

When first rising edge of clock is coming, the data at Memory [7] is loaded into DReadData bus. Then the second rising edge of clock is coming, and the data at DReadData bus is loaded into Register 7, namely Register X. At the same time of second rising edge of clock, the data at Memory [1] is loaded into DReadData bus. When the third rising edge of clock is coming, the data at DReadData bus is loaded into Register 1, namely Register Y. At the same time, ALU sums up the data at Register 7 and Register 1 and sends the calculated result into ALUResultOut bus. As the data at Register 7 is 0x00004444 and Register 1 is 0x0000000A, the result ought to be 0x0000444E, which is just the result shown at ALUResultOut bus and indicates the accuracy of previous stages and operation of processor.

When the fourth rising edge of clock is coming, the calculated result, 0x0000444E, is loaded into Register 2, namely Register Z. Then at the fifth rising edge of clock, the data at Register 2 is stored into Memory [2]. At the last rising edge of clock, the data at Memory 2 is loaded into Register 3, namely Register T. The whole process is correct as the values of opcode and PC mutually correspond. However, since loading will be activated at the next rising edge of clock, the entire process needs cost six rising edges of clocks rather than 5 clocks.

2 BEQ

2.1 Modified code

MIPS	[address] Machine Code (hexadecimal)
lw $7, 7$0	[0] 0x8C070007
lw $1, 1$0	[4] 0x8C010001
add $2, $7, $1	[8] 0x00E11020
sw $2, 2$0	[C] 0xAC020002
lw $3, 2$0	[10] 0x8C030002
beq $2, $3, branch	[14] 0x10620001
add $2, $2, $3	[18] 0x00431020
branch: add $2, $2, $3	[1C] 0x00431020

Table 4 MIPS and corresponding machine code to test BEQ operation.

Instruction				Instruction Mapping			
beq	$2,	$3,	branch	31-26	25-21	20-16	15-0
opcode	rt	rs	offset	opcode	rs	rt	offset (1)
				000100	00011	00010	0000 0000 0000 0001

Table 5 Instruction Mapping for BEQ Operation

2.2 Simulation

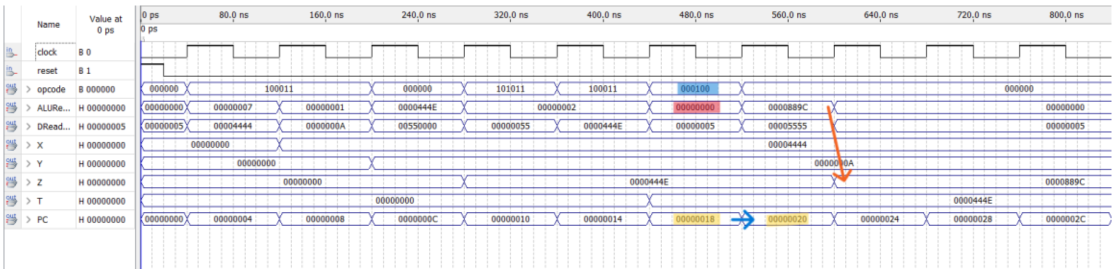
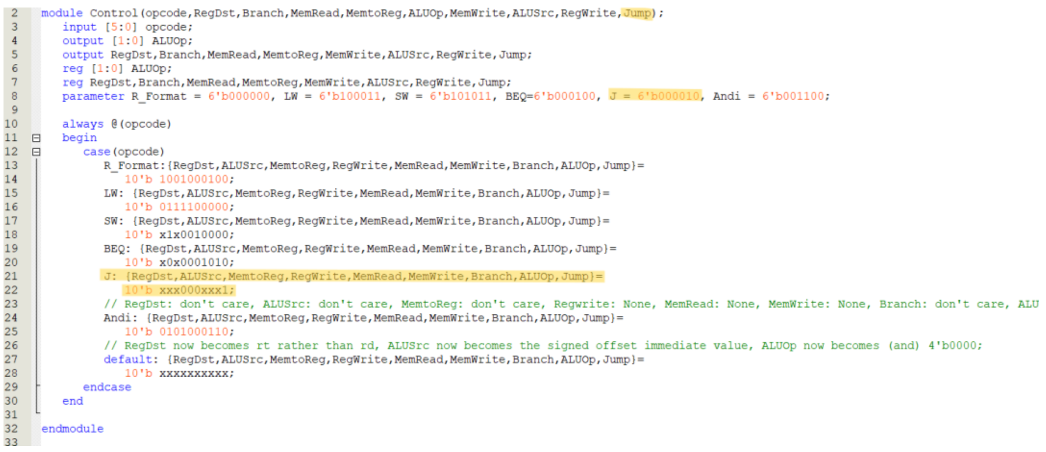

Name
Value at 0 ps
0 ps
80.0 ns
160.0 ns
240.0 ns
320.0 ns
400.0 ns
480.0 ns
560.0 ns
640.0 ns
720.0 ns
800.0 ns
clock B 0
reset B 1
opcode B 000000 | 000000 | 100011 | 000000 | 101011 | 100011 | 000100 | 000000
ALURe... H 00000000 | 00000000 | 00000007 | 00000001 | 0000444E | 00000002 | 00000000 | 0000889C | 00000000
DRead... H 00000005 | 00000005 | 00004444 | 0000000A | 00550000 | 00000055 | 0000444E | 00000005 | 00005555 | 00000005
X H 00000000 | 00000000 | 00004444
Y H 00000000 | 00000000 | 0000000A
Z H 00000000 | 00000000 | 0000444E | 0000889C
T H 00000000 | 00000000 | 0000444E
PC H 00000000 | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | 00000014 | 00000018 | 00000020 | 00000024 | 00000028 | 0000002C

Figure 2 Simulation waveform for BEQ Operation

At the sixth clock, the opcode is 0b000100, indicating that this instruction is a BEQ operation. In addition, the data at ALUResultOut bus is 0x00000000, which suggests the operands of rs and rd have the same value since their difference is 0x00000000. This corresponds to the fact that Register 2 and Register 3 hold the same data 0x0000444E. At the next rising edge of clock, PC changes from 0x00000018 to 0x00000020, and skips the address 0x0000001C. Because PC is 1 clock advanced of instruction execution, the instruction at address 0x00000018 is actually skipped. This means that only the addition at address 0x0000001C will be operated. As a result, at the seventh clock, the ALU calculates the sum of Register 2 and Register 3, namely 0x0000899C, and loads it into ALUResultOut bus. This calculation is the last instruction, so there will be no additional opcode. There still needs an additional rising edge of clock to load the calculated data into Register 2, the eighth rising edge of clock.

3 Jump

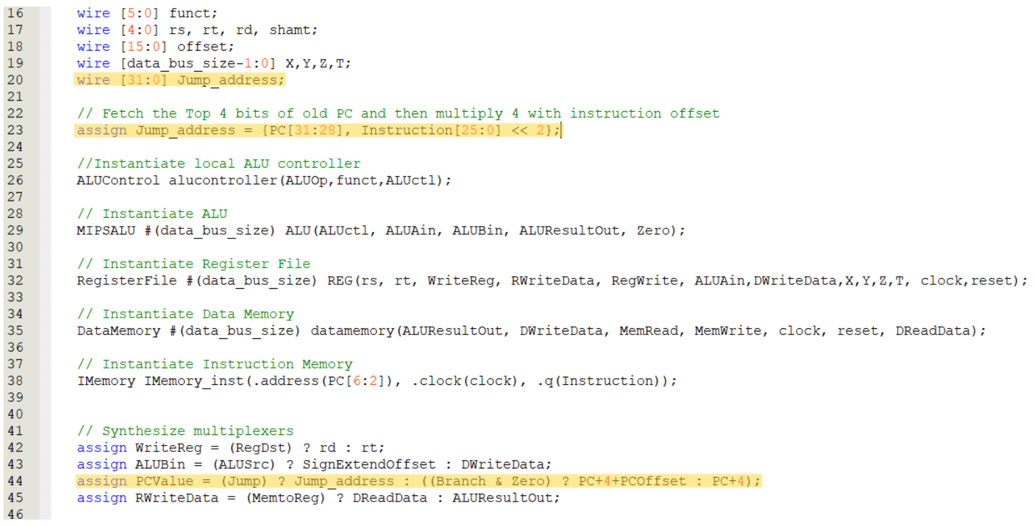
3.1 Modified Processor



```
module Control(opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite,Jump);
   input [5:0] opcode;
   output [1:0] ALUOp;
   output RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump;
   reg [1:0] ALUOp;
   reg RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump;
   parameter R_Format = 6'b000000, LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J = 6'b000010, Andi = 6'b001100;

   always @(opcode)
   begin
      case(opcode)
         R_Format:{RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b 1001000100;
         LW: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b 0111100000;
         SW: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b x1x0010000;
         BEQ: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b x0x0001010;
         J: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b xxx000xxx1;
         // RegDst: don't care, ALUSrc: don't care, MemtoReg: don't care, Regwrite: None, MemRead: None, MemWrite: None, Branch: don't care, ALU
         Andi: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b 0101000110;
         // RegDst now becomes rt rather than rd, ALUSrc now becomes the signed offset immediate value, ALUOp now becomes (and) 4'b0000;
         default: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b xxxxxxxxxx;
      endcase
   end

endmodule
```

Figure 3 Modified module of Control for Jump Operation

There must be an additional signal, Jump, which should be prior to branch, to control the incrementation of PC. In the implement the Jump operation, the register file and data memory should keep unaltered, thus the signal of RegWrite, MemRead and MemWrite ought to be zero. Other control signals can be don’t care and are represented by x. The signal of Jump should be high.



```
	wire [5:0] funct;
	wire [4:0] rs, rt, rd, shamt;
	wire [15:0] offset;
	wire [data_bus_size-1:0] X,Y,Z,T;
	wire [31:0] Jump_address;

	// Fetch the Top 4 bits of old PC and then multiply 4 with instruction offset
	assign Jump_address = {PC[31:28], Instruction[25:0] << 2};

	//Instantiate local ALU controller
	ALUControl alucontroller(ALUOp,funct,ALUctl);

	// Instantiate ALU
	MIPSALU #(data_bus_size) ALU(ALUctl, ALUAin, ALUBin, ALUResultOut, Zero);

	// Instantiate Register File
	RegisterFile #(data_bus_size) REG(rs, rt, WriteReg, RWriteData, RegWrite, ALUAin,DWriteData,X,Y,Z,T, clock,reset);

	// Instantiate Data Memory
	DataMemory #(data_bus_size) datamemory(ALUResultOut, DWriteData, MemRead, MemWrite, clock, reset, DReadData);

	// Instantiate Instruction Memory
	IMemory IMemory_inst(.address(PC[6:2]), .clock(clock), .q(Instruction));


	// Synthesize multiplexers
	assign WriteReg = (RegDst) ? rd : rt;
	assign ALUBin = (ALUSrc) ? SignExtendOffset : DWriteData;
	assign PCValue = (Jump) ? Jump_address : ((Branch & Zero) ? PC+4+PCOffset : PC+4);
	assign RWriteData = (MemtoReg) ? DReadData : ALUResultOut;
```

Figure 4 Modified module of Datapath for Jump Operation

The Jump address should be the top 4 bits of old address, concatenated with the shifted-left-2-bit instruction section from 25 to 0 bit. This will make the PC value automatically multiply with 4. As the previous mention, the Jump signal should be prior to the signal of Branch and Zero.

3.2 Modified Code

MIPS	[address] Machine Code (hexadecimal)
lw $7, 7$0	[0] 0x8C070007
lw $1, 1$0	[4] 0x8C010001
add $2, $7, $1	[8] 0x00E11020
sw $2, 2$0	[C] 0xAC020002
lw $3, 2$0	[10] 0x8C030002
jump: beq $2, $3, branch	[14] 0x10430001
add $2, $2, $3	[18] 0x00431020
branch: add $2, $2, $3	[1C] 0x00431020
j jump	[20] 0x08000005

Table 6 MIPS and corresponding machine code to test Jump operation.

Instruction		Instruction Mapping	
j	jump	31-26	25-0
opcode	address (5)	opcode	address (5)
		000010	00 0000 0000 0000 0000 0000 0101

Table 7 Instruction Mapping for Jump Operation

3.3 Simulation

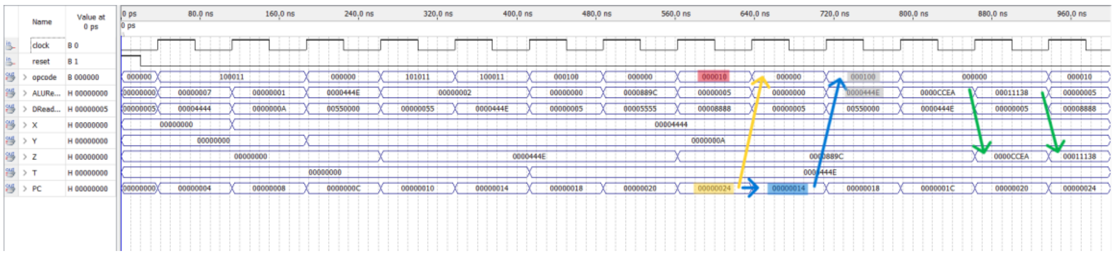
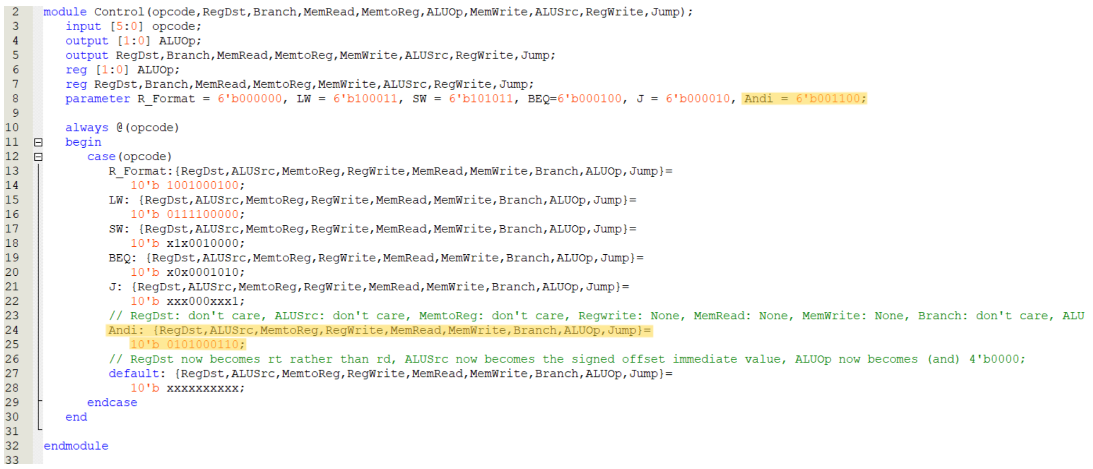


Figure 5 Simulation waveform for Jump Operation

At the eighth clock, the opcode is 0b000010, which indicates this instruction is a Jump operation. When the nineth rising edge of clock is coming, the PCValue changes from address 0x00000024 to 0x0000014, which corresponds the four times of the address section in the instruction. This proves that the jump operation works correctly. In the instruction address 0x00000014, the opcode 0x000100 suggests that BEQ operation runs again, but this time the values at Register 2 and Register 3 are no longer equal, since the value at Register 2 is 0x0000444E larger than Register 2. Therefore, the eventual value at Register 2 will be fourfold of the original value 0x0000444E, namely 0x00011138, rather than twofold of the signal value, 0x0000889C.

4 Andi

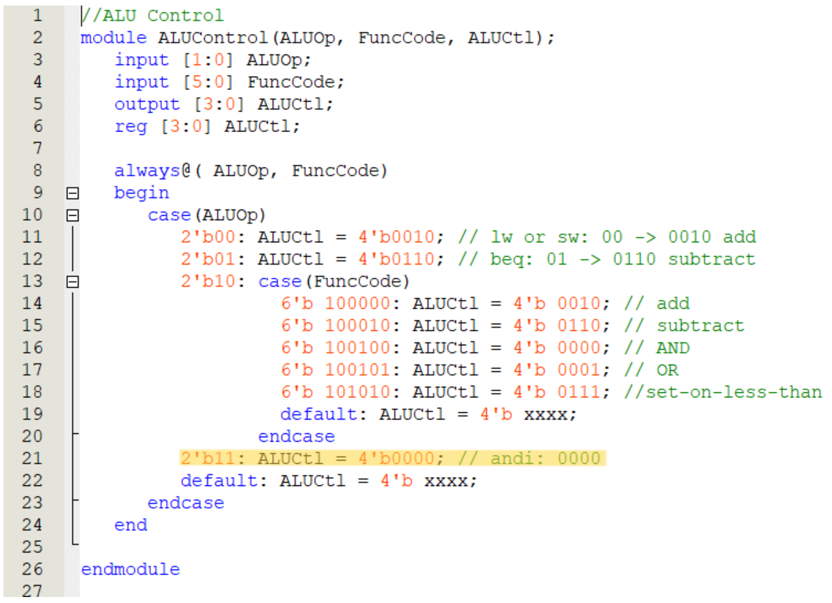
4.1 Modified Processor



```
module Control(opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite,Jump);
   input [5:0] opcode;
   output [1:0] ALUOp;
   output RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump;
   reg [1:0] ALUOp;
   reg RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump;
   parameter R_Format = 6'b000000, LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J = 6'b000010, Andi = 6'b001100;

   always @(opcode)
   begin
      case(opcode)
         R_Format:{RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b 1001000100;
         LW: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b 0111100000;
         SW: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b x1x0010000;
         BEQ: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b x0x0001010;
         J: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b xxx000xxx1;
         // RegDst: don't care, ALUSrc: don't care, MemtoReg: don't care, Regwrite: None, MemRead: None, MemWrite: None, Branch: don't care, ALU
         Andi: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b 0101000110;
         // RegDst now becomes rt rather than rd, ALUSrc now becomes the signed offset immediate value, ALUOp now becomes (and) 4'b0000;
         default: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,ALUOp,Jump}=
            10'b xxxxxxxxxx;
      endcase
   end

endmodule
```

Figure 6 Modified module of Control for Andi Operation

The RegDst now is 0 because there are only rs and rt in instruction section. The destination of ALUResultOut should be pointed to rt rather than rd. This is why RegDst should be zero for Andi operation. In addition, the ALUSrc is 1 as a result of the section of offset, which ranges from 15 to 0 of the instruction. The RegWrite is 1 since register rt will be overwritten. The MemtoReg, MemRead, MemWrite, Branch, and Jump are 0 because those functions don't work for Andi operation. The ALUOp is 11 to facilitate ALU to immediately conduct addition of rs and immediate value.



```
//ALU Control
module ALUControl(ALUOp, FuncCode, ALUCtl);
   input [1:0] ALUOp;
   input [5:0] FuncCode;
   output [3:0] ALUCtl;
   reg [3:0] ALUCtl;

   always@( ALUOp, FuncCode)
   begin
      case(ALUOp)
         2'b00: ALUCtl = 4'b0010; // lw or sw: 00 -> 0010 add
         2'b01: ALUCtl = 4'b0110; // beq: 01 -> 0110 subtract
         2'b10: case(FuncCode)
                  6'b 100000: ALUCtl = 4'b 0010; // add
                  6'b 100010: ALUCtl = 4'b 0110; // subtract
                  6'b 100100: ALUCtl = 4'b 0000; // AND
                  6'b 100101: ALUCtl = 4'b 0001; // OR
                  6'b 101010: ALUCtl = 4'b 0111; //set-on-less-than
                  default: ALUCtl = 4'b xxxx;
               endcase
         2'b11: ALUCtl = 4'b0000; // andi: 0000
         default: ALUCtl = 4'b xxxx;
      endcase
   end

endmodule
```

Figure 7 Modified module of ALUControl for Andi Operation

Add a case of ALUOp, 2'b11, so that the ALU can immediately sum up the value at Register rs and immediate value.

4.2 Modified Code

MIPS	[address] Machine Code (hexadecimal)
lw $7, 7$0	[0] 0x8C070007
andi $1, $7, 0x4A	[4] 0x30E1004A
sw $1, 1$0	[8] 0xAC010001
lw $2, 1$0	[C] 0x8C020001

Table 8 MIPS and corresponding machine code to test Andi operation.

Instruction				Instruction Mapping			
andi	$1,	$7,	0x4A	31-26	25-21	20-16	15-0
opcode	rt	rs	Imme	opcode	rs	rt	Imme (0x4A)
				001100	00111	00001	0000 0000 0100 1010

Table 9 Instruction Mapping for Andi Operation

4.3 Simulation

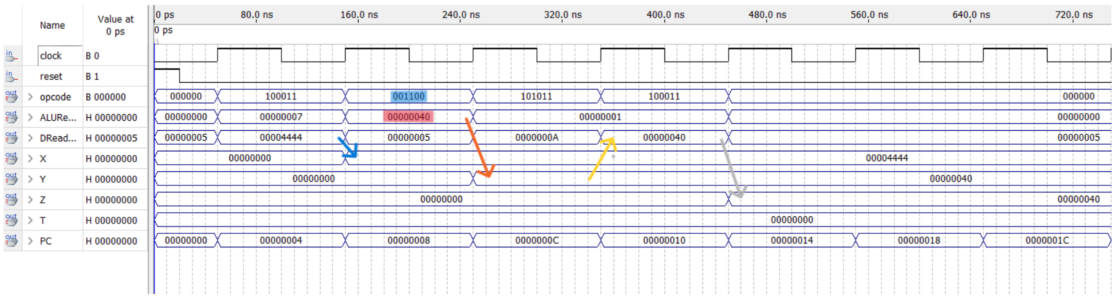
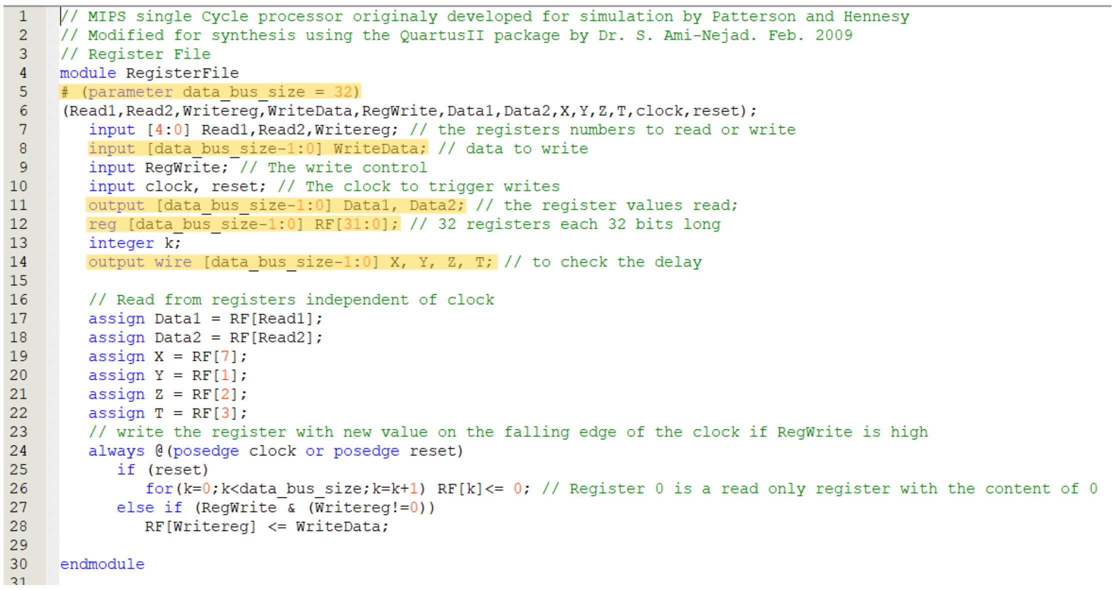


Figure 8 Simulation waveform for Andi Operation

At the first clock the data at Memory [7] is loaded into DReadData bus. At the second rising edge of clock, the data at DreadData bus is loaded into Register 7. At the same time, this data at register 7, 0x00004444, is And with the immediate value 0x0000004A at ALU. The result is 0x00000040 and loaded into ALRResultOut bus. Then the data loaded into Register 1 at the third rising edge of clock.

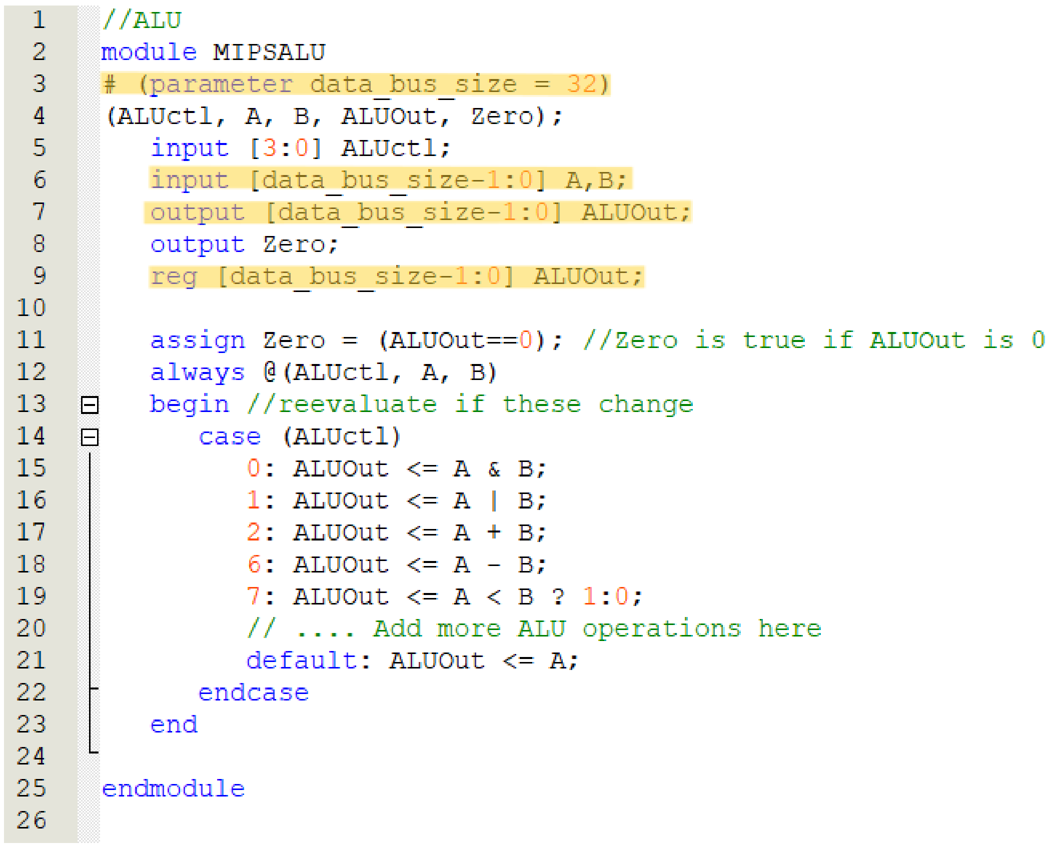
5 Data bus size

5.1 Modified Processor



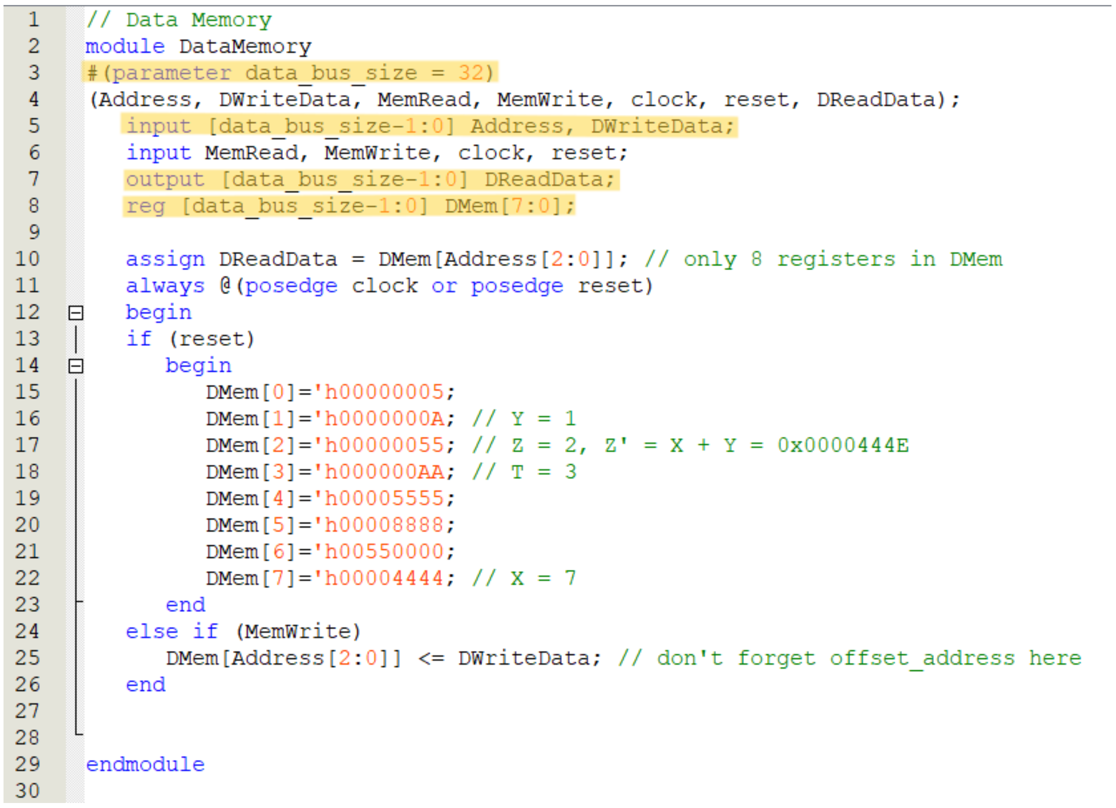
```
// MIPS single Cycle processor originaly developed for simulation by Patterson and Hennesy
// Modified for synthesis using the QuartusII package by Dr. S. Ami-Nejad. Feb. 2009
// Register File
module RegisterFile
# (parameter data_bus_size = 32)
(Read1,Read2,Writereg,WriteData,RegWrite,Data1,Data2,X,Y,Z,T,clock,reset);
   input [4:0] Read1,Read2,Writereg; // the registers numbers to read or write
   input [data_bus_size-1:0] WriteData; // data to write
   input RegWrite; // The write control
   input clock, reset; // The clock to trigger writes
   output [data_bus_size-1:0] Data1, Data2; // the register values read;
   reg [data_bus_size-1:0] RF[31:0]; // 32 registers each 32 bits long
   integer k;
   output wire [data_bus_size-1:0] X, Y, Z, T; // to check the delay

   // Read from registers independent of clock
   assign Data1 = RF[Read1];
   assign Data2 = RF[Read2];
   assign X = RF[7];
   assign Y = RF[1];
   assign Z = RF[2];
   assign T = RF[3];
   // write the register with new value on the falling edge of the clock if RegWrite is high
   always @(posedge clock or posedge reset)
      if (reset)
         for(k=0;k<data_bus_size;k=k+1) RF[k]<= 0; // Register 0 is a read only register with the content of 0
      else if (RegWrite & (Writereg!=0))
         RF[Writereg] <= WriteData;

endmodule
```

Figure 9 Modified module of RegisterFile for Data bus size



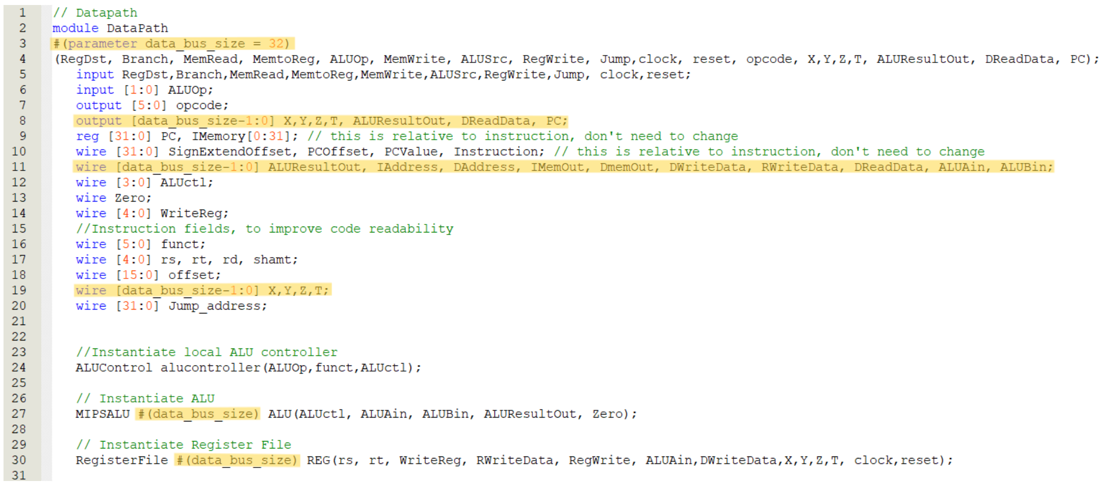
```
//ALU
module MIPSALU
# (parameter data_bus_size = 32)
(ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [data_bus_size-1:0] A,B;
   output [data_bus_size-1:0] ALUOut;
   output Zero;
   reg [data_bus_size-1:0] ALUOut;

   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
   always @(ALUctl, A, B)
   begin //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         // .... Add more ALU operations here
         default: ALUOut <= A;
      endcase
   end

endmodule
```

Figure 10 Modified module of MIPSALU for Data bus size



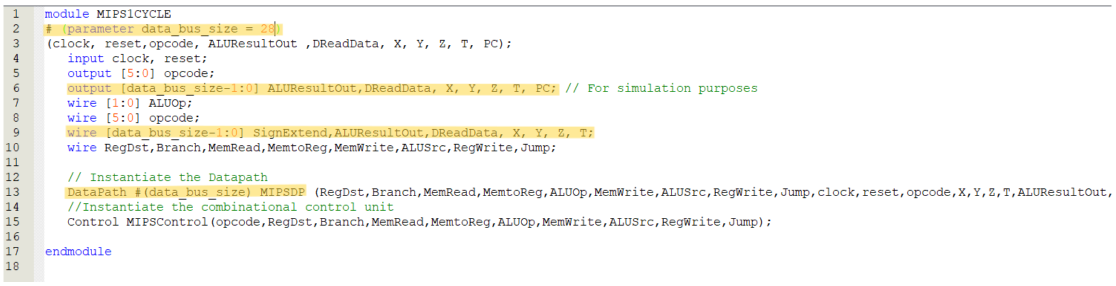
```verilog
// Data Memory
module DataMemory
#(parameter data_bus_size = 32)
(Address, DWriteData, MemRead, MemWrite, clock, reset, DReadData);
   input [data_bus_size-1:0] Address, DWriteData;
   input MemRead, MemWrite, clock, reset;
   output [data_bus_size-1:0] DReadData;
   reg [data_bus_size-1:0] DMem[7:0];

   assign DReadData = DMem[Address[2:0]]; // only 8 registers in DMem
   always @(posedge clock or posedge reset)
   begin
   if (reset)
      begin
         DMem[0]='h00000005;
         DMem[1]='h0000000A; // Y = 1
         DMem[2]='h00000055; // Z = 2, Z' = X + Y = 0x0000444E
         DMem[3]='h000000AA; // T = 3
         DMem[4]='h00005555;
         DMem[5]='h00008888;
         DMem[6]='h00550000;
         DMem[7]='h00004444; // X = 7
      end
   else if (MemWrite)
      DMem[Address[2:0]] <= DWriteData; // don't forget offset_address here
   end


endmodule
```

Figure 11 Modified module of DataMemory for Data bus size



```verilog
// Datapath
module DataPath
#(parameter data_bus_size = 32)
(RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite, Jump,clock, reset, opcode, X,Y,Z,T, ALUResultOut, DReadData, PC);
   input RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump, clock,reset;
   input [1:0] ALUOp;
   output [5:0] opcode;
   output [data_bus_size-1:0] X,Y,Z,T, ALUResultOut, DReadData, PC;
   reg [31:0] PC, IMemory[0:31]; // this is relative to instruction, don't need to change
   wire [31:0] SignExtendOffset, PCOffset, PCValue, Instruction; // this is relative to instruction, don't need to change
   wire [data_bus_size-1:0] ALUResultOut, IAddress, DAddress, IMemOut, DmemOut, DWriteData, RWriteData, DReadData, ALUAin, ALUBin;
   wire [3:0] ALUctl;
   wire Zero;
   wire [4:0] WriteReg;
   //Instruction fields, to improve code readability
   wire [5:0] funct;
   wire [4:0] rs, rt, rd, shamt;
   wire [15:0] offset;
   wire [data_bus_size-1:0] X,Y,Z,T;
   wire [31:0] Jump_address;


   //Instantiate local ALU controller
   ALUControl alucontroller(ALUOp,funct,ALUctl);

   // Instantiate ALU
   MIPSALU #(data_bus_size) ALU(ALUctl, ALUAin, ALUBin, ALUResultOut, Zero);

   // Instantiate Register File
   RegisterFile #(data_bus_size) REG(rs, rt, WriteReg, RWriteData, RegWrite, ALUAin,DWriteData,X,Y,Z,T, clock,reset);

```

Figure 12 Modified module of DataPath for Data bus size



```verilog
module MIPS1CYCLE
# (parameter data_bus_size = 28)
(clock, reset,opcode, ALUResultOut ,DReadData, X, Y, Z, T, PC);
   input clock, reset;
   output [5:0] opcode;
   output [data_bus_size-1:0] ALUResultOut,DReadData, X, Y, Z, T, PC; // For simulation purposes
   wire [1:0] ALUOp;
   wire [5:0] opcode;
   wire [data_bus_size-1:0] SignExtend,ALUResultOut,DReadData, X, Y, Z, T;
   wire RegDst,Branch,MemRead,MemtoReg,MemWrite,ALUSrc,RegWrite,Jump;

   // Instantiate the Datapath
   DataPath #(data_bus_size) MIPSDP (RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite,Jump,clock,reset,opcode,X,Y,Z,T,ALUResultOut,
   //Instantiate the combinational control unit
   Control MIPSControl(opcode,RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite,Jump);

endmodule

```

Figure 13 Modified module of MIPS1CYCLE for Data bus size

To change the data bus for memory data, register file, ALU calculation along with DataPath, the method of exterior parameter for the module is implemented. In this method, we only need to change the parameter at the outmost module, MIPS1CYCLE, into 28, then the data bus size for memory data, register file, ALU and DataPath will automatically change to 28.

5.2 Simulation

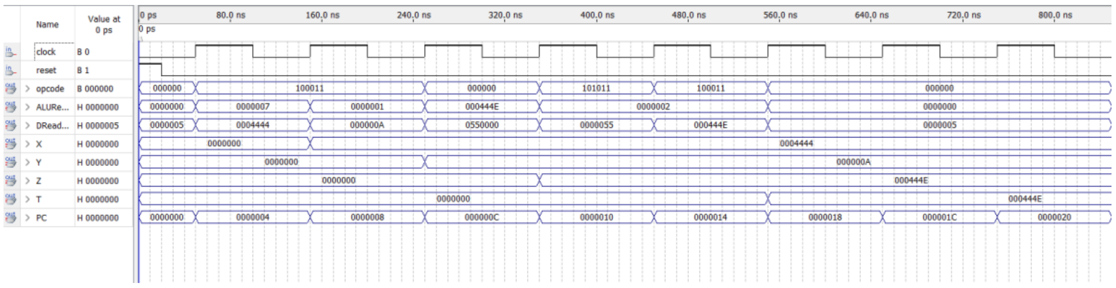

Name
Value at 0 ps
0 ps
80.0 ns
160.0 ns
240.0 ns
320.0 ns
400.0 ns
480.0 ns
560.0 ns
640.0 ns
720.0 ns
800.0 ns
clock B 0
reset B 1
opcode B 000000: 000000, 100011, 000000, 101011, 100011, 000000
ALURe... H 0000000: 0000000, 0000007, 0000001, 000444E, 0000002, 0000000
DRead... H 0000005: 0000005, 0004444, 000000A, 0550000, 0000055, 000444E, 0000005
X H 0000000: 0000000, 0004444
Y H 0000000: 0000000, 000000A
Z H 0000000: 0000000, 000444E
T H 0000000: 0000000, 000444E
PC H 0000000: 0000000, 0000004, 0000008, 000000C, 0000010, 0000014, 0000018, 000001C, 0000020

Figure 14 Simulation waveform for Data bus size of 28

From the above figure, it can be easily found that the data bus sizes of ALUReadOut, DReadOut, X, Y, Z, and T change to 28 since there are only 7 hexadecimal bits.