Eric Smith; Tyler MacNeil

March 7, 2018

CS385: Progress Report 2

**Team Members:**

Tyler MacNeil: Debugged the problems from the previous progress report. Coded the final parts to the MIPS single cycle implementation which included completing the contoller, the data memory(DMemory), module to support the BEQ and BNE branch controller, and the LW and SW instructions. Created CPU diagram and worked on the progress report document.

Eric Smith: Debugged the problems from the previous progress report. Added the ALUmsb diagram to document. Coded the final parts to the MIPS single cycle implementation which included completing the contoller, the data memory(DMemory), module to support the BEQ and BNE branch controller, and the LW and SW instructions. Worked on report document.

**Instruction Set Architecture of the Current Version of the CPU:**

In this current version of our CPU, our CPU consists of a 16-bits of binary data. Each section correlates to the opcode, registers and/or immediate value depending on the register type. An Example line of an addi instruction code:



The first 4-bits in the code, 0100, correlate to the opcode which is the addi opcode. The next 2-bits are the rs address, 00. Followed by another 2-bits denoting the rt register address, 01. The last 8-bits correspond to the immediate value which in this case is 15, so 0000 1111.

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction Set/Meaning | | | |
| Instruction | Opcode | Format | Meaning |
| add | 0000 | R-type instruction | Adds register rs and rt and places the result in rd |
| sub | 0001 | R-type instruction | Subtracts rt from rs and places result into rd |
| and | 0010 | R-type instruction | Performs the ‘and’ operation between rt and rs and places result into register rd |
| or | 0011 | R-type instruction | Performs the ‘or’ operation between rt and rs and places the result in register rd |
| slt | 0111 | R-type instruction | If rs is less than rt, place 1 in register rd |
| addi | 0100 | I-type instruction | Adds the immediate value to register rt and places the result in rs |
| lw | 0101 | I-type instruction | A word is loaded from rs+address from the DMemory to rt. |
| sw | 0110 | I-type instruction | A word is stored in rs+address in the DMemory from rt. |
| beq | 1000 | I-type instruction | Branches if rs and rt are equal and increases the PC by the value in the address/value |
| bne | 1001 | I-type instruction | Branches if rs and rt do not equal and increases the PC by the value in address/value. |

Instruction format key:

R-format (add, sub, and, or, slt)

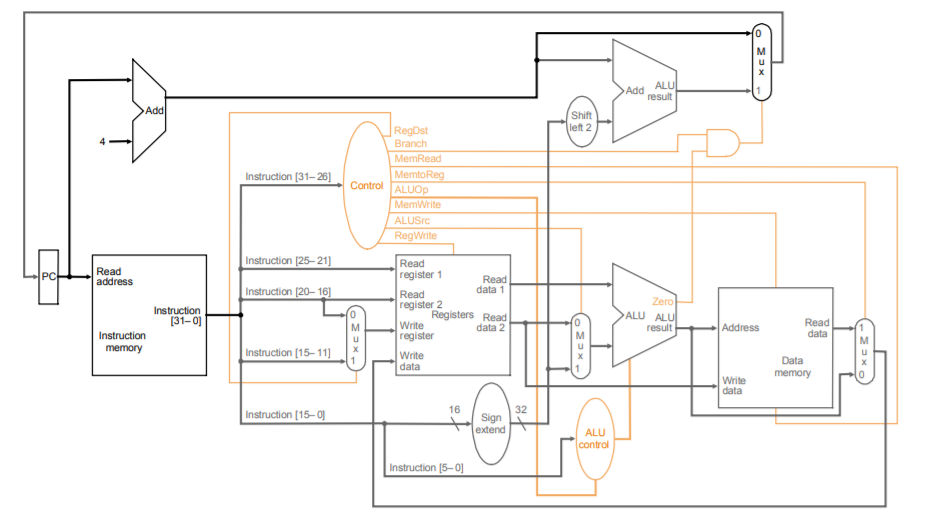
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| op | rs | rt | rd | unused |
| 4 | 2 | 2 | 2 | 6 |

I-format (addi, lw, sw, beq, bne)

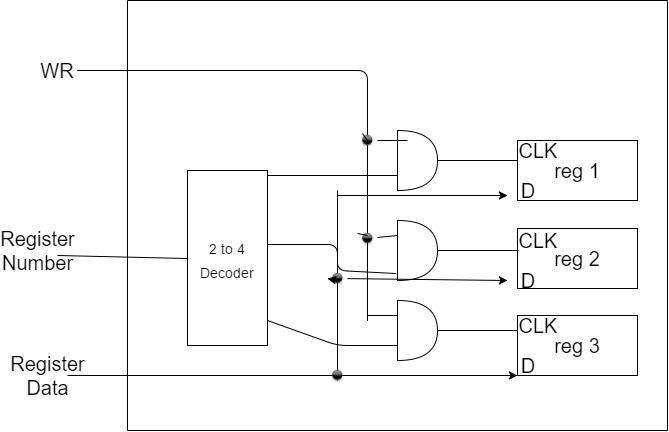
|  |  |  |  |
| --- | --- | --- | --- |
| op | rs | rt | address/value |
| 4 | 2 | 2 | 8 |

**Logic Diagram and Truth Tables:**

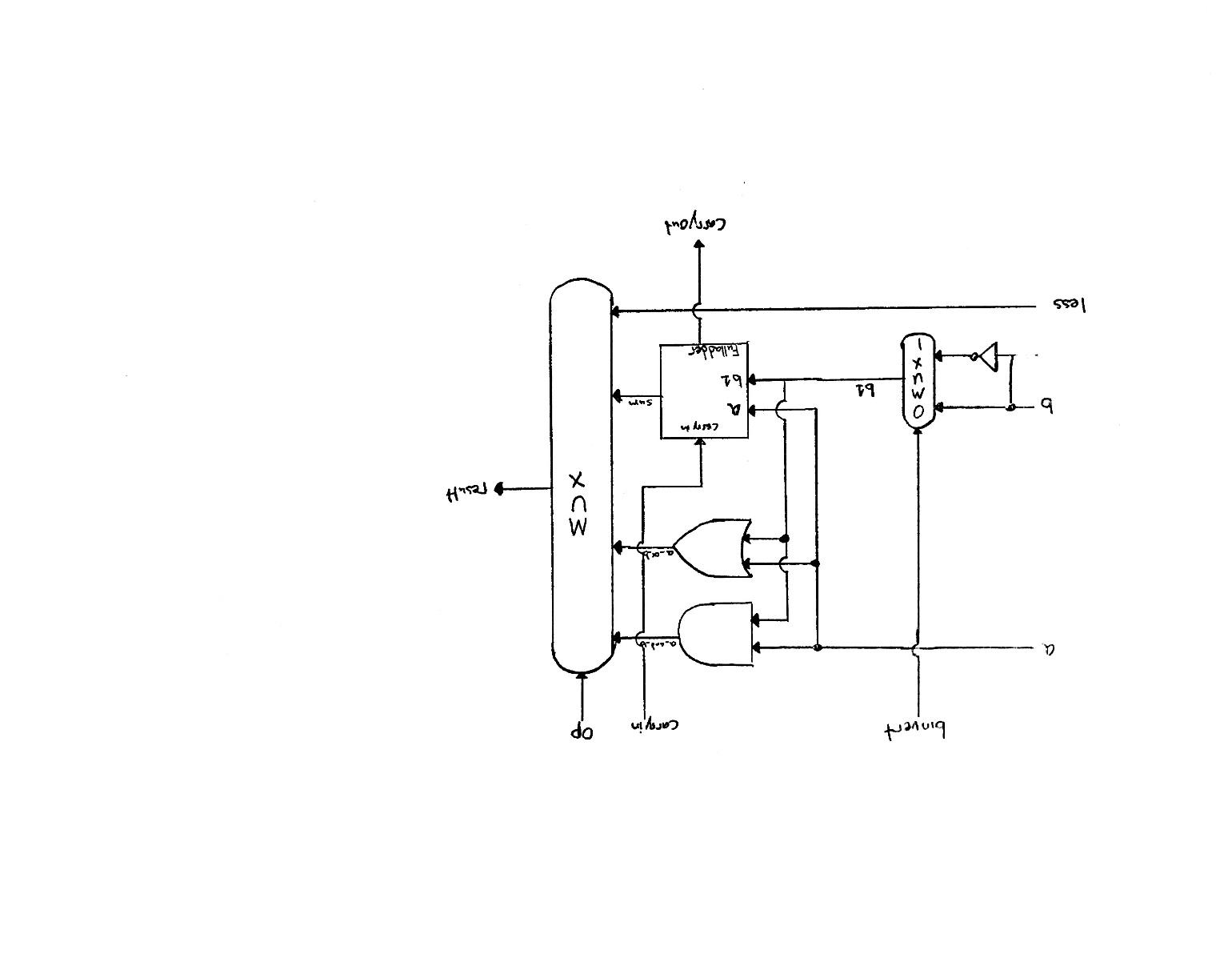
CPU (Single Cycle data path):



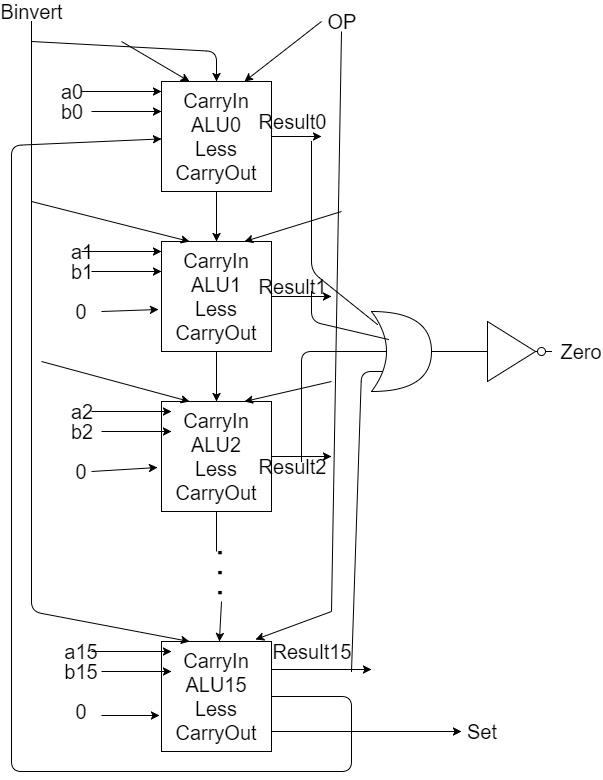
Reg File:

****

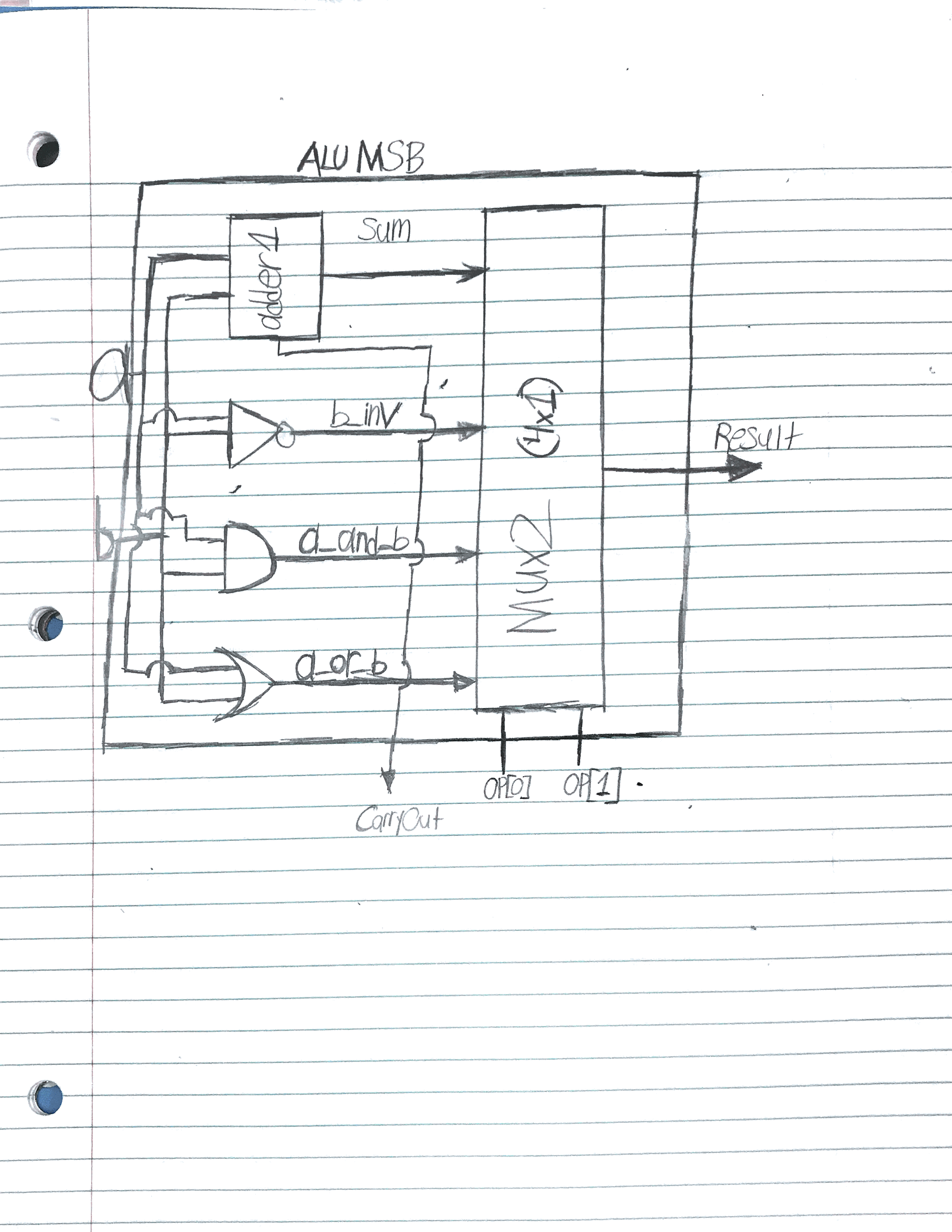
ALU1 1-bit:



ALU 16-bit:



ALU MSB:

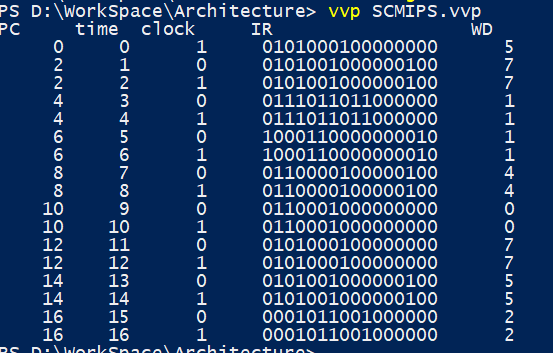


**Source Code:**

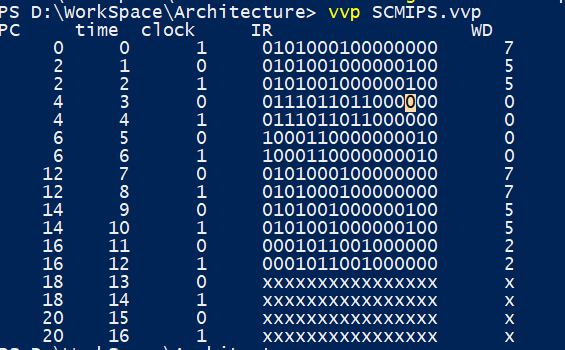
|  |
| --- |
| // Structural model of MIPS - single cycle implementation full implementation  // authors: Tyler MacNeil, Eric Smith  // data: March 30, 2016  // Iteration: 2  module reg\_file (rr1,rr2,wr,wd,regwrite,rd1,rd2,clock);    input [1:0] rr1,rr2,wr;  input [15:0] wd;  input regwrite,clock;  output [15:0] rd1,rd2;  wire [15:0] q1,q2,q3;    // Registers  register reg1(wd,c1,q1);  register reg2(wd,c2,q2);  register reg3(wd,c3,q3);  // Output Port  mux4x1\_16bit mux1 (16'b0,q1,q2,q3,rr1,rd1),  mux2 (16'b0,q1,q2,q3,rr2,rd2);  // Input Port  decoder dec(wr[1],wr[0],w3,w2,w1,w0);    and and0 (regwrite\_and\_clock,regwrite,clock),  and1 (c1,regwrite\_and\_clock,w1),  and2 (c2,regwrite\_and\_clock,w2),  and3 (c3,regwrite\_and\_clock,w3);  endmodule  // Registers build  module register(D,CLK,Q);  input [15:0] D;  input CLK;  output [15:0] Q;    D\_flip\_flop r1 (D[0],CLK,Q[0]);  D\_flip\_flop r2 (D[1],CLK,Q[1]);  D\_flip\_flop r3 (D[2],CLK,Q[2]);  D\_flip\_flop r4 (D[3],CLK,Q[3]);  D\_flip\_flop r5 (D[4],CLK,Q[4]);  D\_flip\_flop r6 (D[5],CLK,Q[5]);  D\_flip\_flop r7 (D[6],CLK,Q[6]);  D\_flip\_flop r8 (D[7],CLK,Q[7]);  D\_flip\_flop r9 (D[8],CLK,Q[8]);  D\_flip\_flop r10 (D[9],CLK,Q[9]);  D\_flip\_flop r11 (D[10],CLK,Q[10]);  D\_flip\_flop r12 (D[11],CLK,Q[11]);  D\_flip\_flop r13 (D[12],CLK,Q[12]);  D\_flip\_flop r14 (D[13],CLK,Q[13]);  D\_flip\_flop r15 (D[14],CLK,Q[14]);  D\_flip\_flop r16 (D[15],CLK,Q[15]);  endmodule  // MIPS ALU 16-bit in Verilog  module ALU(op,a,b,result,zero);  input [15:0] a,b;  input [2:0] op;  output [15:0] result;  output zero;  wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15;    ALU1 alu0 (a[0], b[0], op[2],op[1:0],set, op[2],c1, result[0]),  alu1 (a[1], b[1], op[2],op[1:0],1'b0, c1, c2, result[1]),  alu2 (a[2], b[2], op[2],op[1:0],1'b0, c2, c3, result[2]),  alu3 (a[3], b[3], op[2],op[1:0],1'b0, c3, c4, result[3]),  alu4 (a[4], b[4], op[2],op[1:0],1'b0, c4, c5, result[4]),  alu5 (a[5], b[5], op[2],op[1:0],1'b0, c5, c6, result[5]),  alu6 (a[6], b[6], op[2],op[1:0],1'b0, c6, c7, result[6]),  alu7 (a[7], b[7], op[2],op[1:0],1'b0, c7, c8, result[7]),  alu8 (a[8], b[8], op[2],op[1:0],1'b0, c8, c9, result[8]),  alu9 (a[9], b[9], op[2],op[1:0],1'b0, c9, c10,result[9]),  alu10 (a[10],b[10],op[2],op[1:0],1'b0, c10, c11,result[10]),  alu11 (a[11],b[11],op[2],op[1:0],1'b0, c11, c12,result[11]),  alu12 (a[12],b[12],op[2],op[1:0],1'b0, c12, c13,result[12]),  alu13 (a[13],b[13],op[2],op[1:0],1'b0, c13, c14,result[13]),  alu14 (a[14],b[14],op[2],op[1:0],1'b0, c14, c15,result[14]);    ALUmsb alu15 (a[15],b[15],op[2],op[1:0],1'b0, c15, c16,result[15],set);  or or1(or01, result[0],result[1]);  or or2(or23, result[2],result[3]);  nor nor1(zero,or01,or23);  endmodule  module ALU1(a,b,binvert,op,less,carryin,carryout,result);  input a,b,carryin,binvert;  input less;  input [1:0] op;  output carryout,result;  wire sum,a\_and\_b,a\_or\_b,b\_inv;    not not1(b\_inv, b);  mux2x1 mux1(b,b\_inv,binvert,b1);  and and1(a\_and\_b, a, b);  or or1(a\_or\_b, a, b);  fulladder adder1(sum,carryout,a,b1,carryin);  mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);  endmodule  module ALUmsb(a,b,binvert,op,less,carryin,carryout,result,sum);  input a,b,carryin,binvert;  input less;  input [1:0] op;  output carryout,result;  output sum;  wire sum,a\_and\_b,a\_or\_b,b\_inv;    not not1(b\_inv, b);  mux2x1 mux1(b,b\_inv,binvert,b1);  and and1(a\_and\_b, a, b);  or or1(a\_or\_b, a, b);  fulladder adder1(sum,carryout,a,b1,carryin);  mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);  endmodule  // MIPS MainControl  module MainControl(Op,Control);  input [3:0] Op;  output reg [10:0] Control;    always @(Op) case (Op)  4'b0000: Control <= 11'b10010000010; //add  4'b0001: Control <= 11'b10010000110; //sub  4'b0010: Control <= 11'b10010000000; //and  4'b0011: Control <= 11'b10010000001; //or  4'b0111: Control <= 11'b10010000111; //slt  4'b0100: Control <= 11'b01010000010; //addi  4'b0101: Control <= 11'b01111000010; //LW  4'b0110: Control <= 11'b01000100010; //SW  4'b1000: Control <= 11'b00000001110; //BEQ  4'b1001: Control <= 11'b00000010110; //BNE  endcase  endmodule  //Branch Controller for BEQ and BNE  module BranchControl(bne,beq,Zero,BranchOut);    input bne,beq;  input Zero;  output BranchOut;  wire ZeroInvert,i0,i1;    not not1(ZeroInvert,Zero);  and and1(i0,bne,ZeroInvert),  and2(i1,beq,Zero);  or or1(BranchOut,i0,i1);    endmodule  // MIPS CPU & Test Program  module CPU(clock,WD,IR,PC);  input clock;  output [15:0] WD,IR,PC;  reg [15:0] PC;  reg [15:0] IMemory[0:1023], DMemory[0:1023];  wire [15:0] IR,NextPC,A,B,ALUOut,RD2,SignExtend,PCplus4,Target;  wire [2:0] ALUctl;  wire [2:0] ALUOp;  wire [1:0] WR;  wire RegDst;  wire branchcontrol;    // Test Program  initial begin  //R-types are op=2bit rs=2bit rt=2bit rd=2bit unused=6bit  //16'b0100 00 00 00 000000  //I-types are op=4bit rs=2bit rt=2bit address=8bit  //16'b0000 00 00 00000000    IMemory[0] = 16'b0101000100000000; // lw $1, 0($0)  IMemory[1] = 16'b0101001000000100; // lw $2, 4($0)  IMemory[2] = 16'b0111011011000000; // slt $3, $1, $2  //IMemory[3] = 16'b1000110000000010; // beq $3, $0, 2  IMemory[3] = 16'b1001110000000010; // bne $3, $0, 2  IMemory[4] = 16'b0110000100000100; // sw $1, 4($0)  IMemory[5] = 16'b0110001000000000; // sw $2, 0($0)  IMemory[6] = 16'b0101000100000000; // lw $1, 0($0)  IMemory[7] = 16'b0101001000000100; // lw $2, 4($0)  IMemory[8] = 16'b0001011001000000; // sub $1, $1, $2    DMemory[0] =16'h5;  DMemory[1] =16'h7;  end    initial PC = 0;    assign IR = IMemory [PC>>1];    mux2x1\_2bit mux1(IR[9:8],IR[7:6],RegDst,WR); //RegDst    mux2x1\_16bit mux2(RD2,SignExtend,ALUSrc,B); //ALUSrc    mux2x1\_16bit mux3(ALUOut,DMemory[ALUOut>>2],MemToReg,WD); //MemtoReg    mux2x1\_16bit mux4(PCplus4,Target,BranchConOut,NextPC);    assign SignExtend = {{8{IR[7]}},IR[7:0]}; // sign extension unit    reg\_file rf(IR[11:10],IR[9:8],WR,WD,RegWrite,A,RD2,clock);    ALU fetch (3'b010,PC,16'h2,PCplus4,Unused1);    ALU ex (ALUctl,A,B,ALUOut,Zero);    ALU branch (3'b010,SignExtend<<1,PCplus4,Target,Unused2);    MainControl MainCtr (IR[15:12],{RegDst,ALUSrc,MemToReg,RegWrite,MemRead,MemWrite,bne,beq,ALUctl[2:0]});    BranchControl branch1(bne,beq,Zero,BranchConOut);    always @(negedge clock) begin  PC <= NextPC;  if(MemWrite) DMemory[ALUOut>>2] <= RD2;  end    endmodule  // Test Module  module test();    reg clock;  wire signed [15:0] WD,IR,PC;    CPU test\_cpu(clock,WD,IR,PC);    always #1 clock = ~clock;    initial begin  $display ("PC time clock IR WD");  $monitor ("%d %2d %b %b %d",PC, $time,clock,IR,WD);  clock = 1;  #16 $finish;  end  endmodule  /\* Compiling and simulation  >iverilog  >vvp  //This is with beq enabled and DMemory[0]=5 and DMemory[1]=7  PC time clock IR WD  0 0 1 0101000100000000 5  2 1 0 0101001000000100 7  2 2 1 0101001000000100 7  4 3 0 0111011011000000 1  4 4 1 0111011011000000 1  6 5 0 1000110000000010 1  6 6 1 1000110000000010 1  8 7 0 0110000100000100 4  8 8 1 0110000100000100 4  10 9 0 0110001000000000 0  10 10 1 0110001000000000 0  12 11 0 0101000100000000 7  12 12 1 0101000100000000 7  14 13 0 0101001000000100 5  14 14 1 0101001000000100 5  16 15 0 0001011001000000 2  16 16 1 0001011001000000 2  //This is with beq enabled and DMemory[0]=7 and DMemory[1]=5  PC time clock IR WD  0 0 1 0101000100000000 5  2 1 0 0101001000000100 7  2 2 1 0101001000000100 7  4 3 0 0111011011000000 1  4 4 1 0111011011000000 1  6 5 0 1000110000000010 1  6 6 1 1000110000000010 1  8 7 0 0110000100000100 4  8 8 1 0110000100000100 4  10 9 0 0110001000000000 0  10 10 1 0110001000000000 0  12 11 0 0101000100000000 7  12 12 1 0101000100000000 7  14 13 0 0101001000000100 5  14 14 1 0101001000000100 5  16 15 0 0001011001000000 2  16 16 1 0001011001000000 2  //This is with bne enabled and DMemory[0]=5 and DMemory[1]=7  PC time clock IR WD  0 0 1 0101000100000000 5  2 1 0 0101001000000100 7  2 2 1 0101001000000100 7  4 3 0 0111011011000000 1  4 4 1 0111011011000000 1  6 5 0 1000111000000010 1  6 6 1 1000111000000010 1  8 7 0 0110000100000100 5  8 8 1 0110000100000100 5  10 9 0 0110001000000000 7  10 10 1 0110001000000000 7  12 11 0 0101000100000000 -2  12 12 1 0101000100000000 -2  14 13 0 xxxxxxxxxxxxxxxx x  14 14 1 xxxxxxxxxxxxxxxx x  16 15 0 xxxxxxxxxxxxxxxx x  16 16 1 xxxxxxxxxxxxxxxx x  //This is with bne enabled and DMemory[0]=7 and DMemory[1]=5  PC time clock IR WD  0 0 1 0101000100000000 7  2 1 0 0101001000000100 5  2 2 1 0101001000000100 5  4 3 0 0111011011000000 0  4 4 1 0111011011000000 0  6 5 0 1000111000000010 0  6 6 1 1000111000000010 0  8 7 0 0110000100000100 4  8 8 1 0110000100000100 4  10 9 0 0110001000000000 0  10 10 1 0110001000000000 0  12 11 0 0101000100000000 5  12 12 1 0101000100000000 5  14 13 0 0101001000000100 7  14 14 1 0101001000000100 7  16 15 0 0001011001000000 -2  16 16 1 0001011001000000 -2  \*/    // Components  module D\_flip\_flop(D,CLK,Q);  input D,CLK;  output Q;  wire CLK1,Y;  not not1 (CLK1,CLK);    D\_latch D1(D,CLK,Y),  D2(Y,CLK1,Q);    endmodule  module D\_latch(D,C,Q);    input D,C;  output Q;  wire x,y,D1,Q1;    nand nand1(x,D,C),  nand2(y,D1,C),  nand3(Q,x,Q1),  nand4(Q1,y,Q);    not not1(D1,D);    endmodule  module mux4x1(i0,i1,i2,i3,select,y);  input i0,i1,i2,i3;  input [1:0] select;  output y;  wire S0,S1,w1,w2,w3,w4;    not not1(S0,select[0]),  not2(S1,select[1]);    and and1(w1,i0,S1,S0),  and2(w2,i1,S1,select[0]),  and3(w3,i2,select[1],S0),  and4(w4,i3,select[1],select[0]);    or or1(y,w1,w2,w3,w4);    endmodule  module mux4x1\_16bit(i0,i1,i2,i3,select,y);  input [15:0] i0,i1,i2,i3;  input [1:0] select;  output [15:0] y;    mux4x1 mux1 (1'b0,i1[0], i2[0], i3[0], select[1:0],y[0]),  mux2 (1'b0,i1[1], i2[1], i3[1], select[1:0],y[1]),  mux3 (1'b0,i1[2], i2[2], i3[2], select[1:0],y[2]),  mux4 (1'b0,i1[3], i2[3], i3[3], select[1:0],y[3]),    mux5 (1'b0,i1[4], i2[4], i3[4], select[1:0],y[4]),  mux6 (1'b0,i1[5], i2[5], i3[5], select[1:0],y[5]),  mux7 (1'b0,i1[6], i2[6], i3[6], select[1:0],y[6]),  mux8 (1'b0,i1[7], i2[7], i3[7], select[1:0],y[7]),    mux9 (1'b0,i1[8], i2[8], i3[8], select[1:0],y[8]),  mux10(1'b0,i1[9], i2[9], i3[9], select[1:0],y[9]),  mux11(1'b0,i1[10],i2[10],i3[10],select[1:0],y[10]),  mux12(1'b0,i1[11],i2[11],i3[11],select[1:0],y[11]),    mux13(1'b0,i1[12],i2[12],i3[12],select[1:0],y[12]),  mux14(1'b0,i1[13],i2[13],i3[13],select[1:0],y[13]),  mux15(1'b0,i1[14],i2[14],i3[14],select[1:0],y[14]),  mux16(1'b0,i1[15],i2[15],i3[15],select[1:0],y[15]);    endmodule  module decoder(S1,S0,D3,D2,D1,D0);  input S0,S1;  output D0,D1,D2,D3;    not not1(notS0,S0),  not2(notS1,S1);    and and0(D0,notS1,notS0),  and1(D1,notS1,S0),  and2(D2,S1,notS0),  and3(D3,S1,S0);  endmodule  module halfadder(S,C,x,y);  input x,y;  output S,C;  xor (S,x,y);  and (C,x,y);  endmodule  module fulladder(S,C,x,y,z);  input x,y,z;  output S,C;  wire S1,D1,D2;  halfadder HA1 (S1,D1,x,y),  HA2 (S,D2,S1,z);  or g1(C,D2,D1);  endmodule  module mux2x1(A,B,select,OUT);  input A,B,select;  output OUT;  wire S\_inv,wire1,wire2;    not not1(S\_inv, select);  and and1(wire1,A,S\_inv),  and2(wire2,B,select);  or or1(OUT,wire1,wire2);  endmodule  module mux2x1\_2bit(A,B,select,OUT);    input [1:0] A,B;  input select;  output [1:0] OUT;    mux2x1 mux1(A[0],B[0],select,OUT[0]),  mux2(A[1],B[1],select,OUT[1]);    endmodule  module mux2x1\_16bit(A,B,select,OUT);  input [15:0] A,B;  input select;  output [15:0] OUT;    mux2x1 mux1 (A[0], B[0], select,OUT[0]),  mux2 (A[1], B[1], select,OUT[1]),  mux3 (A[2], B[2], select,OUT[2]),  mux4 (A[3], B[3], select,OUT[3]),  mux5 (A[4], B[4], select,OUT[4]),  mux6 (A[5], B[5], select,OUT[5]),  mux7 (A[6], B[6], select,OUT[6]),  mux8 (A[7], B[7], select,OUT[7]),  mux9 (A[8], B[8], select,OUT[8]),  mux10(A[9], B[9], select,OUT[9]),  mux11(A[10],B[10],select,OUT[10]),  mux12(A[11],B[11],select,OUT[11]),  mux13(A[12],B[12],select,OUT[12]),  mux14(A[13],B[13],select,OUT[13]),  mux15(A[14],B[14],select,OUT[14]),  mux16(A[15],B[15],select,OUT[15]);  endmodule |

**Test Results:**

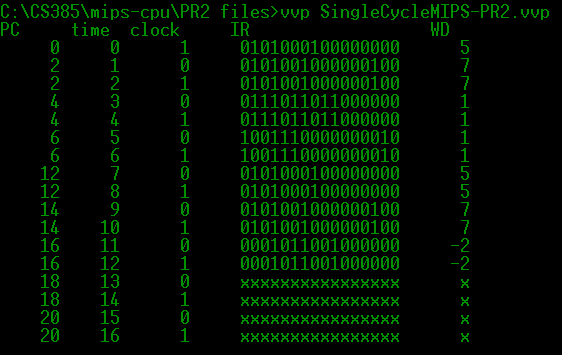
*Test with BEQ enabled and DMemory[0]=5 and DMemory[1]=7*

**

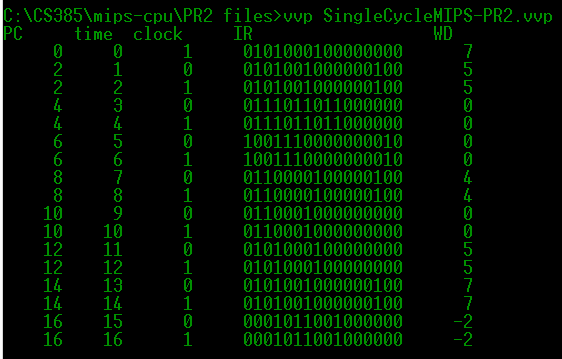
*Test with BEQ enabled and DMemory[0]=7 and DMemory[1]=5*

**

*Test with BNE enabled and DMemory[0]=5 and DMemory[1]=7*

**

*Test with BNE enabled and DMemory[0]=7 and DMemory[1]=5*

**