CS385 – CPU Project

Robert Rotaru, Bryan Bigelow, Anthony Cerritelli

# Progress Report 2

## Task Format

**Task**

* Subtask – [Contributor 1], [Contributor 2], etc.

## Tasks

**16-Bit Extended Test Program**

* Assembly – Bryan
* Binary – Bryan

**Branching Logic**

* Bne/beq Instructions – Bryan, Rob, Tony
* Branching Modules – Rob, Tony

**Data Memory**

* Memory modules – Rob, Tony
* Sw/lw instructions – Bryan, Rob, Tony

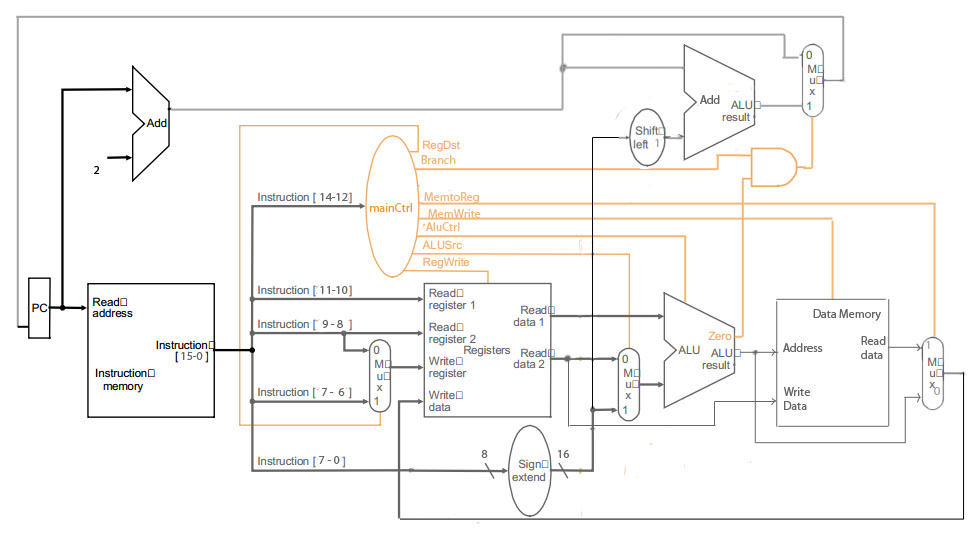
**Diagrams**

* Single-Cycle Path – Tony
* Internals - Bryan

**Report** - Robert Rotaru, Bryan Bigelow, Anthony Cerritelli

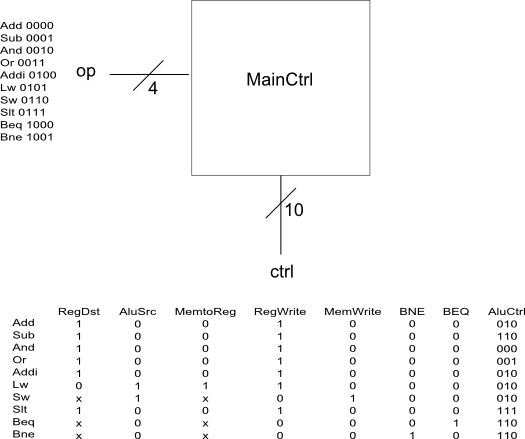
## Diagrams

**Single-Cycle Datapath**

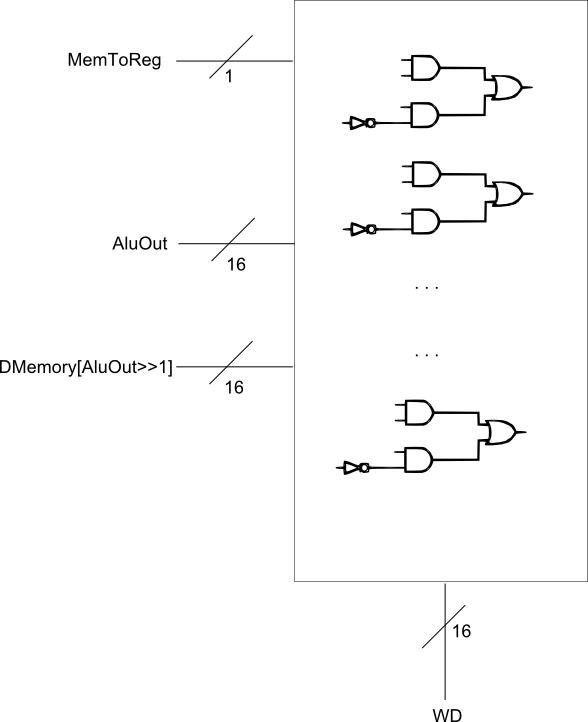


**Internals**

Transcoding ALU table into internal signals



Determining Write Data



## Source Code

/\* CS 385 - Semester Project - Progress Report 1

Authors:

Robert Rotaru

Bryan Bigelow

Anthony Cerritelli

Content:

16-bit MIPS CPU in Verilog

All source code and project work can be found on GitHub at:

https://github.com/rrotaru/CS385-CPU

\*/

/\*\*\* Multiplexers \*\*\*/

module mux2x1(A,B,select,OUT);

input A,B,select;

output OUT;

wire x,y;

and g1(x,A,~select),

g2(y,B,select);

or g3(OUT,x,y);

endmodule

module mux4x1(i0,i1,i2,i3,select,O);

input i0,i1,i2,i3;

input [1:0] select;

output O;

wire w,x,y,z;

and g1(w,i0,~select[1],~select[0]),

g2(x,i1,~select[1],select[0]),

g3(y,i2,select[1],~select[0]),

g4(z,i3,select[1],select[0]);

or g5(O,w,x,y,z);

endmodule

module mux16Bit4x1(i0, i1, i2, i3, select, O);

input [15:0] i0, i1, i2, i3;

input [1:0] select;

output [15:0] O;

mux4x1 mux0(i0[0], i1[0], i2[0], i3[0], select, O[0]),

mux1(i0[1], i1[1], i2[1], i3[1], select, O[1]),

mux2(i0[2], i1[2], i2[2], i3[2], select, O[2]),

mux3(i0[3], i1[3], i2[3], i3[3], select, O[3]),

mux4(i0[4], i1[4], i2[4], i3[4], select, O[4]),

mux5(i0[5], i1[5], i2[5], i3[5], select, O[5]),

mux6(i0[6], i1[6], i2[6], i3[6], select, O[6]),

mux7(i0[7], i1[7], i2[7], i3[7], select, O[7]),

mux8(i0[8], i1[8], i2[8], i3[8], select, O[8]),

mux9(i0[9], i1[9], i2[9], i3[9], select, O[9]),

mux10(i0[10], i1[10], i2[10], i3[10], select, O[10]),

mux11(i0[11], i1[11], i2[11], i3[11], select, O[11]),

mux12(i0[12], i1[12], i2[12], i3[12], select, O[12]),

mux13(i0[13], i1[13], i2[13], i3[13], select, O[13]),

mux14(i0[14], i1[14], i2[14], i3[14], select, O[14]),

mux15(i0[15], i1[15], i2[15], i3[15], select, O[15]);

endmodule

module mux2bit2x1(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 mux16bit2x1(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

/\*\*\* 16-bit D flip flop \*\*\*/

module D\_16\_Flip\_flop(D,CLK,Q);

input [15:0] D;

input CLK;

output [15:0] Q;

D\_flip\_flop f0(D[0], CLK, Q[0]),

f1(D[1], CLK, Q[1]),

f2(D[2], CLK, Q[2]),

f3(D[3], CLK, Q[3]),

f4(D[4], CLK, Q[4]),

f5(D[5], CLK, Q[5]),

f6(D[6], CLK, Q[6]),

f7(D[7], CLK, Q[7]),

f8(D[8], CLK, Q[8]),

f9(D[9], CLK, Q[9]),

f10(D[10], CLK, Q[10]),

f11(D[11], CLK, Q[11]),

f12(D[12], CLK, Q[12]),

f13(D[13], CLK, Q[13]),

f14(D[14], CLK, Q[14]),

f15(D[15], CLK, Q[15]);

endmodule

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

/\*\*\* 16-bit register source code \*\*\*/

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

D\_16\_Flip\_flop r1 (wd,c1,q1);

D\_16\_Flip\_flop r2 (wd,c2,q2);

D\_16\_Flip\_flop r3 (wd,c3,q3);

// output port

mux16Bit4x1 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 a (regwrite\_and\_clock,regwrite,clock);

and a1 (c1,regwrite\_and\_clock,w1),

a2 (c2,regwrite\_and\_clock,w2),

a3 (c3,regwrite\_and\_clock,w3);

endmodule

module decoder (S1,S0,D3,D2,D1,D0);

input S0,S1;

output D0,D1,D2,D3;

not n1 (notS0,S0),

n2 (notS1,S1);

and a0 (D0,notS1,notS0),

a1 (D1,notS1, S0),

a2 (D2, S1,notS0),

a3 (D3, S1, S0);

endmodule

/\*\*\* ALU and arithmetic source code \*\*\*/

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

// 1-bit ALU for bits 0-14

module ALU1 (a,b,binvert,op,less,carryin,carryout,result);

input a,b,less,carryin,binvert;

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

// 1-bit ALU for the most significant bit

module ALUmsb (a,b,binvert,op,less,carryin,carryout,result,sum);

input a,b,less,carryin,binvert;

input [1:0] op;

output carryout,result,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

module ALU (op,a,b,result,zero);

input [15:0] a;

input [15:0] 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,c16;

ALU1 alu0 (a[0], b[0], op[2], op[1:0],set,op[2],c1, result[0]);

ALU1 alu1 (a[1], b[1], op[2], op[1:0],0, c1, c2, result[1]);

ALU1 alu2 (a[2], b[2], op[2], op[1:0],0, c2, c3, result[2]);

ALU1 alu3 (a[3], b[3], op[2], op[1:0],0, c3, c4, result[3]);

ALU1 alu4 (a[4], b[4], op[2], op[1:0],0, c4, c5, result[4]);

ALU1 alu5 (a[5], b[5], op[2], op[1:0],0, c5, c6, result[5]);

ALU1 alu6 (a[6], b[6], op[2], op[1:0],0, c6, c7, result[6]);

ALU1 alu7 (a[7], b[7], op[2], op[1:0],0, c7, c8, result[7]);

ALU1 alu8 (a[8], b[8], op[2], op[1:0],0, c8, c9, result[8]);

ALU1 alu9 (a[9], b[9], op[2], op[1:0],0, c9, c10,result[9]);

ALU1 alu10 (a[10],b[10],op[2], op[1:0],0, c10, c11,result[10]);

ALU1 alu11 (a[11],b[11],op[2], op[1:0],0, c11, c12,result[11]);

ALU1 alu12 (a[12],b[12],op[2], op[1:0],0, c12, c13,result[12]);

ALU1 alu13 (a[13],b[13],op[2], op[1:0],0, c13, c14,result[13]);

ALU1 alu14 (a[14],b[14],op[2], op[1:0],0, c14, c15,result[14]);

ALUmsb alu15 (a[15],b[15],op[2], op[1:0],0, 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 branchCtrl (BranchOp, Zero, BranchOut);

input [1:0] BranchOp;

input Zero;

output BranchOut;

wire not\_Zero, w1, w2;

not G1(not\_Zero, Zero);

and G2(w1, BranchOp[0], Zero),

G3(w2, BranchOp[1], not\_Zero);

or G4(BranchOut, w1, w2);

endmodule

/\*\*\* 16-bit CPU control source code \*\*\*/

module mainCtrl (op, ctrl);

//input [2:0] op;

input [3:0] op;

//output reg [5:0] ctrl;

output reg [9:0] ctrl;

always @(op) case (op)

// RegDst, AluSrc, MemtoReg, RegWrite, MemWrite, BNE, BEQ, AluCtrl

4'b0000: ctrl <= 10'b1001000010; // ADD 1 0 0 1 0 0 0 010

4'b0001: ctrl <= 10'b1001000110; // SUB 1 0 0 1 0 0 0 110

4'b0010: ctrl <= 10'b1001000000; // AND 1 0 0 1 0 0 0 000

4'b0011: ctrl <= 10'b1001000001; // OR 1 0 0 1 0 0 0 001

4'b0100: ctrl <= 10'b0101000010; // ADDI 1 0 0 1 0 0 0 010

4'b0101: ctrl <= 10'b0111000010; // LW 0 1 1 1 0 0 0 010

4'b0110: ctrl <= 10'b0100100010; // SW X 1 X 0 1 0 0 010

4'b0111: ctrl <= 10'b1001000111; // SLT 1 0 0 1 0 0 0 111

4'b1000: ctrl <= 10'b0000001110; // BEQ X 0 X 0 0 0 1 110

4'b1001: ctrl <= 10'b0000010110; // BNE X 0 X 0 0 1 0 110

endcase

endmodule

module CPU (clock, WD, IR);

input clock;

output [15:0] AluOut, IR, WD;

reg[15:0] PC, IMemory[0:1023], DMemory[0:1023];

wire [15:0] IR, NextPC, A, B, AluOut, RD2, SignExtend, PCplus2, Target;

wire [2:0] AluCtrl;

wire [1:0] WR, Branch;

/\* Test Program \*/

initial begin

// Simple program to load 2 from DMemory[0] into $1 and 4 from

// DMemory[1] into $2, then loops to decrement $2 by $1 until

// $2 reaches 0. Finishes by storing 0 into DMemory[1].

IMemory[0] = 16'b0101000100000000; // lw $1, 0($0) 2 Load DMemory[0] into $1

IMemory[1] = 16'b0101001000000010; // lw $2, 2($0) 4 Load DMemory[1] into $2

IMemory[2] = 16'b0111011011000000; // slt $3, $1, $2 1 Set $3 on $1 < $2

IMemory[3] = 16'b1000110000000100; // beq $3, $0, 4 X Branch to IMemory[8] if $3 == 0

IMemory[4] = 16'b0110000100000010; // sw $1, 2($0) X Store $1 into DMemory[1]

IMemory[5] = 16'b0110001000000000; // sw $2, 0($0) X Store $2 into DMemory[2]

IMemory[6] = 16'b0000000000000000; // add $0, $0, $0 0 No operation

IMemory[7] = 16'b0101000100000000; // lw $1, 0($0) 4 Load DMemory[0] into $1

IMemory[8] = 16'b0101001000000010; // lw $2, 2($0) 2 Load DMemory[1] into $2

IMemory[9] = 16'b0000000000000000; // add $0, $0, $0 0 No operation

IMemory[10] = 16'b0001011001000000; // sub $1, $1, $2 2 $1 <- $1 - $2

IMemory[11] = 16'b1001000111111100; // bne $1, $0, -4 X Branch to IMemory[8] if $1 != 0

IMemory[12] = 16'b0110000100000010; // sw $1, 2($0) 0 Store $1 into DMemory[1]

IMemory[13] = 16'b1000010000000100; // beq $1, $0, 4 X Branch to IMemory[18] if $1 == 0

IMemory[14] = 16'b0100000100000001; // addi $t1, $0, 1 X Branched over

IMemory[15] = 16'b0100000100000001; // addi $t1, $0, 1 X Branched over

IMemory[16] = 16'b0100000100000001; // addi $t1, $0, 1 X Branched over

IMemory[17] = 16'b0100000100000001; // addi $t1, $0, 1 X Branched over

IMemory[18] = 16'b0101001000000010; // lw $2, 2($0) 0 Load DMemory[1] into $2

// Data

DMemory [0] = 16'h2;

DMemory [1] = 16'h4;

end

initial PC = 0;

assign IR = IMemory[PC>>1];

assign SignExtend = {{8{IR[7]}},IR[7:0]};

reg\_file rf (IR[11:10], IR[9:8], WR, WD, RegWrite, A, RD2, clock);

ALU fetch (3'b010, PC, 16'b10, PCplus2, Unused1);

ALU exec (AluCtrl, A, B, AluOut, Zero);

ALU branch (3'b010,SignExtend<<1,PCplus2,Target,Unused2);

mainCtrl main (IR[15:12], {RegDst, AluSrc, MemtoReg, RegWrite, MemWrite, Branch, AluCtrl});

// assign WR

mux2bit2x1 muxWR (IR[9:8], IR[7:6], RegDst, WR);

// assign WD

mux16bit2x1 muxWD (AluOut, DMemory[AluOut>>1], MemtoReg, WD);

// assign B

mux16bit2x1 muxB (RD2, SignExtend, AluSrc, B);

// assign NextPC

branchCtrl bctrl (Branch, Zero, BranchOut);

mux16bit2x1 muxBranch (PCplus2, Target, BranchOut, NextPC);

always @(negedge clock) begin

PC <= NextPC;

if (MemWrite) DMemory[AluOut>>1] <= RD2;

end

endmodule

/\*\*\* CPU testing source code \*\*\*/

module test();

reg clock;

wire [15:0] WD,IR;

CPU test\_cpu(clock,WD,IR);

always #1 clock = ~clock;

initial begin

$display ("time clock\tIR\tIR\t\t\tWD\tWD");

$monitor ("%2d %b\t\t%h\t%b\t%h\t%b", $time,clock,IR,IR,WD,WD);

clock = 1;

#37 $finish;

end

endmodule

## Testing

Source Files\$ iverilog -o mips-cpu single-cycle-datapath.vl

Source FIles\$ vvp mips-cpu

time clock IR IR WD WD

0 1 5100 0101000100000000 0002 0000000000000010

1 0 5202 0101001000000010 0004 0000000000000100

2 1 5202 0101001000000010 0004 0000000000000100

3 0 76c0 0111011011000000 0001 0000000000000001

4 1 76c0 0111011011000000 0001 0000000000000001

5 0 8c04 1000110000000100 0001 0000000000000001

6 1 8c04 1000110000000100 0001 0000000000000001

7 0 6102 0110000100000010 0002 0000000000000010

8 1 6102 0110000100000010 0002 0000000000000010

9 0 6200 0110001000000000 0000 0000000000000000

10 1 6200 0110001000000000 0000 0000000000000000

11 0 0000 0000000000000000 0000 0000000000000000

12 1 0000 0000000000000000 0000 0000000000000000

13 0 5100 0101000100000000 0004 0000000000000100

14 1 5100 0101000100000000 0004 0000000000000100

15 0 5202 0101001000000010 0002 0000000000000010

16 1 5202 0101001000000010 0002 0000000000000010

17 0 0000 0000000000000000 0000 0000000000000000

18 1 0000 0000000000000000 0000 0000000000000000

19 0 1640 0001011001000000 0002 0000000000000010

20 1 1640 0001011001000000 0002 0000000000000010

21 0 91fc 1001000111111100 fffe 1111111111111110

22 1 91fc 1001000111111100 fffe 1111111111111110

23 0 5202 0101001000000010 0002 0000000000000010

24 1 5202 0101001000000010 0002 0000000000000010

25 0 0000 0000000000000000 0000 0000000000000000

26 1 0000 0000000000000000 0000 0000000000000000

27 0 1640 0001011001000000 0000 0000000000000000

28 1 1640 0001011001000000 0000 0000000000000000

29 0 91fc 1001000111111100 0000 0000000000000000

30 1 91fc 1001000111111100 0000 0000000000000000

31 0 6102 0110000100000010 0002 0000000000000010

32 1 6102 0110000100000010 0002 0000000000000010

33 0 8404 1000010000000100 0000 0000000000000000

34 1 8404 1000010000000100 0000 0000000000000000

35 0 5202 0101001000000010 0000 0000000000000000

36 1 5202 0101001000000010 0000 0000000000000000

37 0 xxxx xxxxxxxxxxxxxxxx xxxx xxxxxxxxxxxxxxxx