Michael Santoro

Bradley Morales

Julio Sarzosa

Progress Report 3

**Description:**

For this progress report, we were tasked to implement a 3-stage pipelined MIPS processor that is capable of completing all of the r-type instructions, in addition to addi. This was done by combining the EX, MEM, and WB stages of the standard 5-stage pipelined processor, in addition to using the IF and ID stages.

**Instruction Set:**

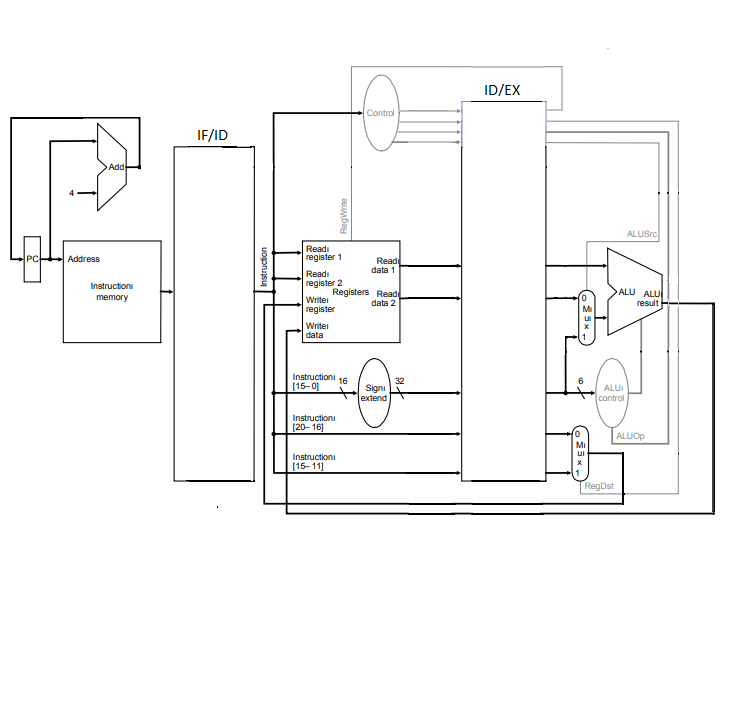
| Instruction | OpCode | RegDst | ALUsrc | RegWrite | ALUCtl |
| --- | --- | --- | --- | --- | --- |
| add | 0000 | 1 | 0 | 1 | 0010 |
| sub | 0001 | 1 | 0 | 1 | 0110 |
| and | 0010 | 1 | 0 | 1 | 0000 |
| or | 0011 | 1 | 0 | 1 | 0001 |
| nor | 0100 | 1 | 0 | 1 | 1100 |
| nand | 0101 | 1 | 0 | 1 | 1101 |
| slt | 0110 | 1 | 0 | 1 | 0111 |
| addi | 0111 | 0 | 1 | 1 | 0010 |



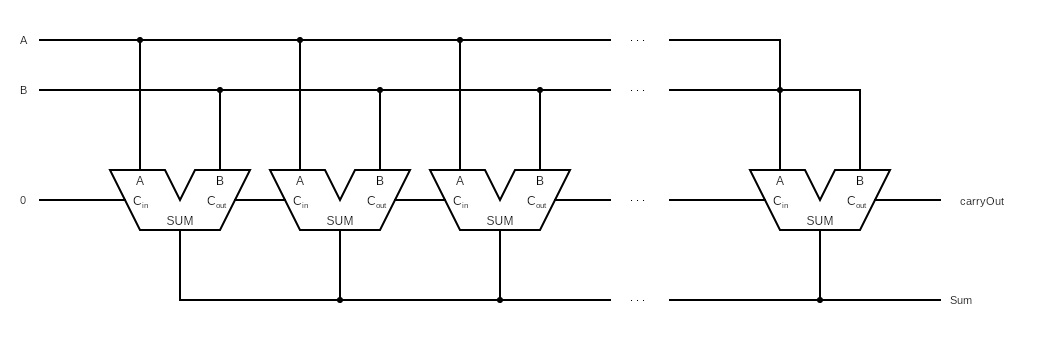
**Logic Diagrams:**

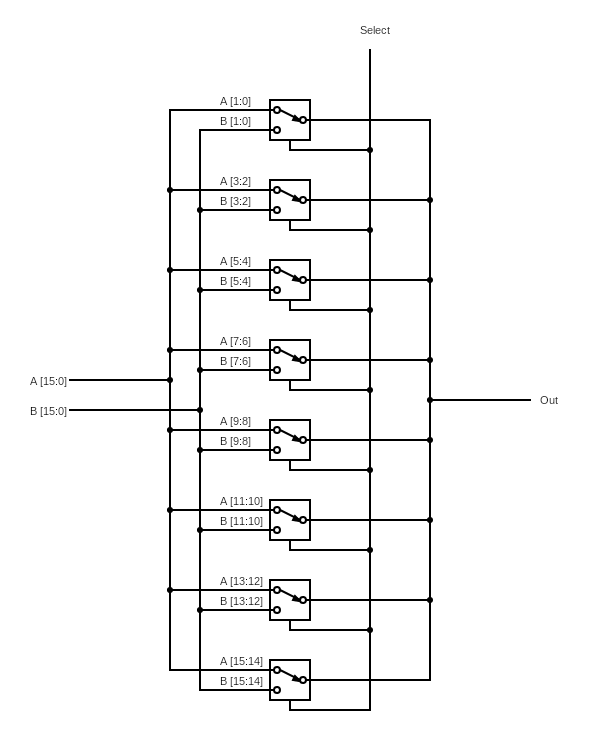
**ALU Diagram:**

**CPU Diagram:**



**16-Bit Full Adder Diagram:**

Note: The software used to create this diagram represents a full adder as the shape of an ALU. In this diagram these are supposed to be 16 Full-Adders strung together.

**16-Bit 2x1 Multiplexor:**

**Verilog Code:**

// Michael, Brad, Julio

// Progress Report 3

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;

reg [15:0] Regs[0:15];

assign RD1 = Regs[RR1];

assign RD2 = Regs[RR2];

initial Regs[0] = 0;

always @(negedge clock)

if (RegWrite==1 & WR!=0)

Regs[WR] <= WD;

endmodule

// 16-bit MIPS ALU in Verilog using modified template of 4-bit ALU

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

input [15:0] a;

input [15:0] b;

input [3:0] op;

output [15:0] result;

output zero;

wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;

//16 single bit alu's

ALU1 alu0 (a[0], b[0], op[3], op[2], op[1:0],set,op[2],c1,result[0]);

ALU1 alu1 (a[1], b[1], op[3], op[2], op[1:0],1'b0, c1, c2,result[1]);

ALU1 alu2 (a[2], b[2], op[3], op[2], op[1:0],1'b0, c2, c3,result[2]);

ALU1 alu3 (a[3], b[3], op[3], op[2], op[1:0],1'b0, c3, c4,result[3]);

ALU1 alu4 (a[4], b[4], op[3], op[2], op[1:0],1'b0, c4, c5,result[4]);

ALU1 alu5 (a[5], b[5], op[3], op[2], op[1:0],1'b0, c5, c6,result[5]);

ALU1 alu6 (a[6], b[6], op[3], op[2], op[1:0],1'b0, c6, c7,result[6]);

ALU1 alu7 (a[7], b[7], op[3], op[2], op[1:0],1'b0, c7, c8,result[7]);

ALU1 alu8 (a[8], b[8], op[3], op[2], op[1:0],1'b0, c8, c9,result[8]);

ALU1 alu9 (a[9], b[9], op[3], op[2], op[1:0],1'b0, c9, c10,result[9]);

ALU1 alu10 (a[10],b[10],op[3], op[2], op[1:0],1'b0, c10, c11,result[10]);

ALU1 alu11 (a[11],b[11],op[3], op[2], op[1:0],1'b0, c11, c12,result[11]);

ALU1 alu12 (a[12],b[12],op[3], op[2], op[1:0],1'b0, c12, c13,result[12]);

ALU1 alu13 (a[13],b[13],op[3], op[2], op[1:0],1'b0, c13, c14,result[13]);

ALU1 alu14 (a[14],b[14],op[3], op[2], op[1:0],1'b0, c14, c15,result[14]);

ALUmsb alu15 (a[15],b[15],op[3], op[2], op[1:0],1'b0, c15, c16,result[15],set);

nor nor1(zero,result [15:0]);

endmodule

// 1-bit ALU for bits 0-14

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

input a,b,less,carryin,ainvert, binvert;

input [1:0] op;

output carryout,result;

wire sum, a\_and\_b, a\_or\_b, b\_inv, a\_inv;

// Choose if we use a or a invert

not not1(a\_inv, a);

mux2x1 mux1(a,a\_inv,ainvert,a1);

// Chose if we use b or b invert

not not2(b\_inv, b);

mux2x1 mux2(b,b\_inv,binvert,b1);

and and1(a\_and\_b, a1, b1);

or or1(a\_or\_b, a1, b1);

fulladder adder1(sum,carryout,a1,b1,carryin);

mux4x1 mux3(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);

endmodule

// 1-bit ALU for smb

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

input a,b,less,carryin,ainvert,binvert;

input [1:0] op;

output carryout,result,sum;

wire sum, a\_and\_b, a\_or\_b, b\_inv, a\_inv;

// Choose if we use a or a invert

not not1(a\_inv, a);

mux2x1 mux1(a,a\_inv,ainvert,a1);

// Choose if we use b or b invert

not not2(b\_inv, b);

mux2x1 mux2(b,b\_inv,binvert,b1);

and and1(a\_and\_b, a1, b1);

or or1(a\_or\_b, a1, b1);

fulladder adder1(sum,carryout,a1,b1,carryin);

mux4x1 mux3(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);

endmodule

// Adders

module halfadder (S,carryOut,A,B);

input A,B;

output S,carryOut;

xor (S,A,B);

and (carryOut,A,B);

endmodule

module fulladder (S,carryOut,A,B,carryIn);

input A,B,carryIn;

output S,carryOut;

wire S1,D1,D2;

halfadder HA1 (S1,D1,A,B),

HA2 (S,D2,S1,carryIn);

or g1(carryOut,D2,D1);

endmodule

// Multiplexors

// 16-bit 4x1 Mux

module mux16bit(a, b, c, d, select, OUT);

input [15:0] a, b, c, d;

input [1:0] select;

output [15:0] OUT;

//16 4x1 mux

mux4x1 mux0(a[0], b[0], c[0], d[0], select, OUT[0]);

mux4x1 mux1(a[1], b[1], c[1], d[1], select, OUT[1]);

mux4x1 mux2(a[2], b[2], c[2], d[2], select, OUT[2]);

mux4x1 mux3(a[3], b[3], c[3], d[3], select, OUT[3]);

mux4x1 mux4(a[4], b[4], c[4], d[4], select, OUT[4]);

mux4x1 mux5(a[5], b[5], c[5], d[5], select, OUT[5]);

mux4x1 mux6(a[6], b[6], c[6], d[6], select, OUT[6]);

mux4x1 mux7(a[7], b[7], c[7], d[7], select, OUT[7]);

mux4x1 mux8(a[8], b[8], c[8], d[8], select, OUT[8]);

mux4x1 mux9(a[9], b[9], c[9], d[9], select, OUT[9]);

mux4x1 mux10(a[10], b[10], c[10], d[10], select, OUT[10]);

mux4x1 mux11(a[11], b[11], c[11], d[11], select, OUT[11]);

mux4x1 mux12(a[12], b[12], c[12], d[12], select, OUT[12]);

mux4x1 mux13(a[13], b[13], c[13], d[13], select, OUT[13]);

mux4x1 mux14(a[14], b[14], c[14], d[14], select, OUT[14]);

mux4x1 mux15(a[15], b[15], c[15], d[15], select, OUT[15]);

endmodule

// 1-bit 4x1 Mux

module mux4x1(a,b,c,d,select,OUT);

input a,b,c,d;

input [1:0] select;

output OUT;

mux2x1 mux1(a, b, select[0], m1);

mux2x1 mux2(c, d, select[0], m2);

mux2x1 mux3(m1, m2, select[1], OUT);

endmodule

// 16-bit 2x1 Mux

// Rewritten For Extra Work to use 8 2-bit 2x1 Mux's instead of 16 1-bit 2x1 Mux's

module mux2x1\_16bit(A, B, select, OUT);

input [15:0] A,B;

input select;

output [15:0] OUT;

//8 2-bit 2x1 mux's

mux2x1\_2bit mux1(A[1:0],B[1:0],select,OUT[1:0]);

mux2x1\_2bit mux2(A[3:2],B[3:2],select,OUT[3:2]);

mux2x1\_2bit mux3(A[5:4],B[5:4],select,OUT[5:4]);

mux2x1\_2bit mux4(A[7:6],B[7:6],select,OUT[7:6]);

mux2x1\_2bit mux5(A[9:8],B[9:8],select,OUT[9:8]);

mux2x1\_2bit mux6(A[11:10],B[11:10],select,OUT[11:10]);

mux2x1\_2bit mux7(A[13:12],B[13:12],select,OUT[13:12]);

mux2x1\_2bit mux8(A[15:14],B[15:14],select,OUT[15:14]);

endmodule

// 2-bit 2x1 Mux

module mux2x1\_2bit(A,B,select,OUT);

input [1:0] A,B;

input select;

output [1:0] OUT;

//2 2x1 muxs

mux2x1 mux1(A[0], B[0], select, OUT[0]),

mux2(A[1], B[1], select, OUT[1]);

endmodule

// 1-bit 2x1 Mux

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

input A,B,select;

output OUT;

not not1(i0, select);

and and1(i1, A, i0);

and and2(i2, B, select);

or or1(OUT, i1, i2);

endmodule

module MainControl (Op,Control);

input [3:0] Op;

output reg [6:0] Control;

// Control bits: RegDst,ALUSrc,RegWrite,ALUCtl

always @(Op) case (Op)

4'b0000: Control <= 7'b1010010; // add

4'b0001: Control <= 7'b1010110; // sub

4'b0010: Control <= 7'b1010000; // and

4'b0011: Control <= 7'b1010001; // or

4'b0100: Control <= 7'b1011100; // nor

4'b0101: Control <= 7'b1011101; // nand

4'b0110: Control <= 7'b1010111; // slt

4'b0111: Control <= 7'b0110010; // addi

endcase

endmodule

module CPU (clock,PC,IFID\_IR,IDEX\_IR,WD);

input clock;

output [15:0] PC,IFID\_IR,IDEX\_IR,WD;

initial begin

// Program with nop's - no hazards

IMemory[0] = 16'b0111\_00\_01\_00001111; // addi $t1, $0, 15 ($t1=15)

IMemory[1] = 16'b0111\_00\_10\_00000111; // addi $t2, $0, 7 ($t2=7)

IMemory[2] = 16'b0000000000000000; // nop

IMemory[3] = 16'b0010\_01\_10\_11\_000000; // and $t3, $t1, $t2 ($t3=7)

IMemory[4] = 16'b0000000000000000; // nop

IMemory[5] = 16'b0001\_01\_11\_10\_000000; // sub $t2, $t1, $t3 ($t2=8)

IMemory[6] = 16'b0000000000000000; // nop

IMemory[7] = 16'b0011\_10\_11\_10\_000000; // or $t2, $t2, $t3 ($t2=15)

IMemory[8] = 16'b0000000000000000; // nop

IMemory[9] = 16'b0000\_10\_11\_11\_000000; // add $t3, $t2, $t3 ($t3=22)

IMemory[10] = 16'b0000000000000000; // nop

IMemory[11] = 16'b0100\_10\_11\_01\_000000; // nor $t1, $t2, $t3 ($t1=-32)

IMemory[12] = 16'b0110\_11\_10\_01\_000000; // slt $t1, $t3, $t2 ($t1=0)

IMemory[13] = 16'b0110\_10\_11\_01\_000000; // slt $t1, $t2, $t3 ($t1=1

end

/\*

initial begin

// Program without nop's - wrong results due to data hazards

IMemory[0] = 16'b0111\_00\_01\_00001111; // addi $t1, $0, 15 ($t1=15)

IMemory[1] = 16'b0111\_00\_10\_00000111; // addi $t2, $0, 7 ($t2=7)

IMemory[2] = 16'b0010\_01\_10\_11\_000000; // and $t3, $t1, $t2 ($t3=7)

IMemory[3] = 16'b0001\_01\_11\_10\_000000; // sub $t2, $t1, $t3 ($t2=8)

IMemory[4] = 16'b0011\_10\_11\_10\_000000; // or $t2, $t2, $t3 ($t2=15)

IMemory[5] = 16'b0000\_10\_11\_11\_000000; // add $t3, $t2, $t3 ($t3=22)

IMemory[6] = 16'b0100\_10\_11\_01\_000000; // nor $t1, $t2, $t3 ($t1=-32)

IMemory[7] = 16'b0110\_11\_10\_01\_000000; // slt $t1, $t3, $t2 ($t1=0)

IMemory[8] = 16'b0110\_10\_11\_01\_000000; // slt $t1, $t2, $t3 ($t1=1

end

\*/

// Pipeline stages

//=== IF STAGE ===

wire [15:0] NextPC;

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

//--------------------------------

reg[15:0] IFID\_IR;

//--------------------------------

ALU fetch (4'b0010,PC,16'd2,NextPC,Unused);

//=== ID STAGE ===

wire [6:0] Control;

wire [15:0] RD1,RD2,SignExtend,WD;

wire [15:0] FWD\_RD1,FWD\_RD2; // Outputs of the forwarding muxes

reg [15:0] IDEX\_IR; // For monitoring the pipeline

reg IDEX\_RegWrite,IDEX\_ALUSrc,IDEX\_RegDst;

reg [3:0] IDEX\_ALUOp;

reg [15:0] IDEX\_RD1,IDEX\_RD2,IDEX\_SignExt;

reg [1:0] IDEX\_rt,IDEX\_rd;

wire [1:0] WR;

reg\_file rf (IFID\_IR[11:10],IFID\_IR[9:8],WR,WD,IDEX\_RegWrite,RD1,RD2,clock);

MainControl MainCtr (IFID\_IR[15:12],Control);

assign SignExtend = {{8{IFID\_IR[7]}},IFID\_IR[7:0]};

//=== EXE STAGE ===

wire [15:0] B,ALUOut;

ALU ex (IDEX\_ALUOp, IDEX\_RD1, B, ALUOut, Zero);

mux2x1\_16bit m1(IDEX\_RD2,IDEX\_SignExt,IDEX\_ALUSrc,B); // ALUSrc Mux

mux2x1\_2bit m2(IDEX\_rt,IDEX\_rd,IDEX\_RegDst,WR); // RegDst Mux

assign WD = ALUOut;

// Forwarding multiplexers

mux2x1\_16bit m3 (RD1,ALUOut,(IDEX\_RegWrite && WR==IFID\_IR[11:10]),FWD\_RD1);

mux2x1\_16bit m4 (RD2,ALUOut,(IDEX\_RegWrite && WR==IFID\_IR[9:8]),FWD\_RD2);

initial begin

PC = 0;

IFID\_IR = 0; // clear pipeline register to avoid forwarding from empty pipeline

IDEX\_RegWrite = 0;

end

// Running the pipeline

always @(negedge clock) begin

// Stage 1 - IF

PC <= NextPC;

IFID\_IR <= IMemory[PC>>1];

// Stage 2 - ID

IDEX\_IR <= IFID\_IR; // For monitoring the pipeline

{IDEX\_RegDst,IDEX\_ALUSrc,IDEX\_RegWrite,IDEX\_ALUOp} <= Control;

// No Forwarding

IDEX\_RD1 <= RD1;

IDEX\_RD2 <= RD2;

// Forwarding

//IDEX\_RD1 <= FWD\_RD1;

//IDEX\_RD2 <= FWD\_RD2;

IDEX\_SignExt <= SignExtend;

IDEX\_rt <= IFID\_IR[9:8];

IDEX\_rd <= IFID\_IR[7:6];

// Stage 3 - EX

// No transfers needed here - on negedge WD is written into register WR

end

endmodule

// Test module

module test ();

reg clock;

wire signed [15:0] PC,IFID\_IR,IDEX\_IR,WD;

CPU test\_cpu(clock,PC,IFID\_IR,IDEX\_IR,WD);

always #1 clock = ~clock;

initial begin

$display ("PC IFID\_IR IDEX\_IR WD");

$monitor ("%2d %h %h %2d",PC,IFID\_IR,IDEX\_IR,WD);

clock = 1;

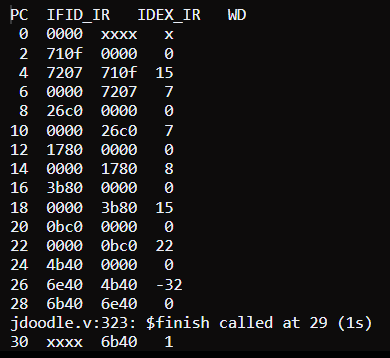
#29 $finish;

end

endmodule

**Output:**

Program with nop's - no hazards



Program without nop's - wrong results due to data hazards

