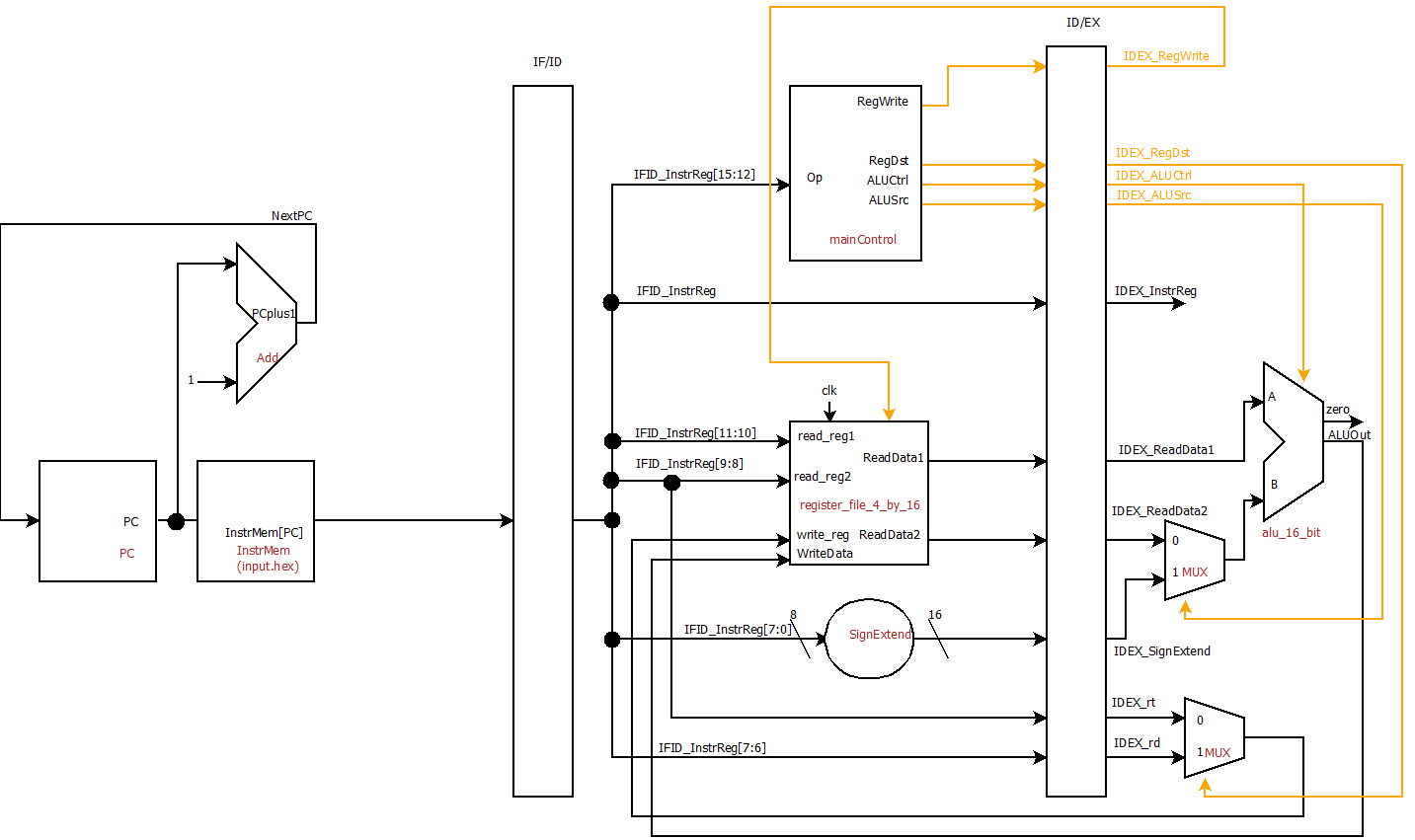
CS 385 – Progress Report II

Rachael Grabowski: Diagrams, progress report  
Michael Lang: Code

# I. Diagrams

## Simplified Three-Stage Pipelined Datapath for addi and R-type Instructions

Diagram is viewed best when document size is 200% or larger.



# II. Verilog Source Code

## module alu\_1\_bit(result, c\_out, a, b, less, c\_in, op);

input a, b;

input less;

input c\_in;

input [2:0] op;

output result, c\_out;

wire m, n, p, q, r;

not g1(m, b);

mux\_2\_to\_1 mux1(n, b, m, op[2]);

and g2(p, a, n);

or g3(q, a, n);

full\_adder fa(r, c\_out, a, n, c\_in);

mux\_4\_to\_1 mux2(result, p, q, r, less, op[1:0]);

endmodule

## module alu\_1\_bit\_msb(result, overflow, set, c\_out, a, b, less, c\_in, op);

input a, b;

input less;

input c\_in;

input [2:0] op;

output result, overflow, set, c\_out;

wire m, n, p, q;

not g1(m, b);

mux\_2\_to\_1 mux1(n, b, m, op[2]);

and g2(p, a, n);

or g3(q, a, n);

full\_adder fa(set, c\_out, a, n, c\_in);

mux\_4\_to\_1 mux2(result, p, q, set, less, op[1:0]);

xor g4(overflow, c\_out, c\_in);

endmodule

## module alu\_4\_bit(result, zero, c\_out, a, b, less, c\_in, op);

input [3:0] a;

input [3:0] b;

input less;

input c\_in;

input [2:0] op;

output [3:0] result;

output zero, c\_out;

wire p, q, r, s, t;

alu\_1\_bit alu\_0(result[0], p, a[0], b[0], less, c\_in, op),

alu\_1(result[1], q, a[1], b[1], 0, p, op),

alu\_2(result[2], r, a[2], b[2], 0, q, op),

alu\_3(result[3], c\_out, a[3], b[3], 0, r, op);

or g1(s, result[0], result[1]),

g2(t, result[2], result[3]);

nor g3(zero, s, t);

endmodule

## module alu\_4\_bit\_last(result, overflow, set, zero, c\_out, a, b, c\_in, op);

input [3:0] a;

input [3:0] b;

input c\_in;

input [2:0] op;

output [3:0] result;

output overflow, set, zero, c\_out;

wire p, q, r, s, t;

alu\_1\_bit alu\_0(result[0], p, a[0], b[0], 0, c\_in, op),

alu\_1(result[1], q, a[1], b[1], 0, p, op),

alu\_2(result[2], r, a[2], b[2], 0, q, op);

alu\_1\_bit\_msb alu\_3(result[3], overflow, set, c\_out, a[3], b[3], 0, r, op);

or g1(s, result[0], result[1]),

g2(t, result[2], result[3]);

nor g3(zero, s, t);

endmodule

## module alu\_16\_bit(result, overflow, zero, c\_out, a, b, op);

input [15:0] a;

input [15:0] b;

input [2:0] op;

output [15:0] result;

output overflow, zero, c\_out;

wire p, q, r, s, t, u, v, w, x;

wire less;

alu\_4\_bit alu\_0(result[3:0], s, p, a[3:0], b[3:0], less, op[2], op),

alu\_1(result[7:4], t, q, a[7:4], b[7:4], 0, p, op),

alu\_2(result[11:8], u, r, a[11:8], b[11:8], 0, q, op);

alu\_4\_bit\_last alu\_3(result[15:12], overflow, less, v, c\_out, a[15:12], b[15:12], r, op);

and g1(w, s, t),

g2(x, u, v),

g3(zero, w, x);

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

## module decoder\_2\_to\_4(out, sel);

input [1:0] sel;

output [3:0] out;

wire not\_sel0, not\_sel1;

not n0(not\_sel0, sel[0]),

n1(not\_sel1, sel[1]);

and a0(out[0], not\_sel0, not\_sel1),

a1(out[1], sel[0], not\_sel1),

a2(out[2], not\_sel0, sel[1]),

a3(out[3], sel[0], sel[1]);

endmodule

## module full\_adder(sum, c\_out, a, b, c\_in);

input a, b, c\_in;

output sum, c\_out;

wire p, q, r;

half\_adder ha1(p, q, a, b),

ha2(sum, r, p, c\_in);

or g1(c\_out, r, q);

endmodule

## module half\_adder(sum, c\_out, a, b);

input a, b;

output sum, c\_out;

xor g1 (sum, a, b);

and g2 (c\_out, a, b);

endmodule

## module mux\_2\_to\_1(x, a, b, sel);

input a, b, sel;

output x;

wire p, q, r;

not g1 (p, sel);

and g2 (q, a, p),

g3 (r, b, sel);

or g4 (x, q, r);

endmodule

## module mux\_4\_to\_1(x, a, b, c, d, sel);

input a, b, c, d;

input [1:0] sel;

output x;

wire p, q;

mux\_2\_to\_1 mux1(p, a, b, sel[0]),

mux2(q, c, d, sel[0]),

mux3(x, p, q, sel[1]);

endmodule

## module mux\_4\_to\_2(x, a, b, sel);

input [1:0] a, b;

input sel;

output [1:0] x;

mux\_2\_to\_1 mux1(x[0], a[0], b[0], sel),

mux2(x[1], a[1], b[1], sel);

endmodule

## module mux\_32\_to\_16(x, a, b, sel);

input [15:0] a, b;

input sel;

output [15:0] x;

mux\_2\_to\_1 m0(x[0], a[0], b[0], sel),

m1(x[1], a[1], b[1], sel),

m2(x[2], a[2], b[2], sel),

m3(x[3], a[3], b[3], sel),

m4(x[4], a[4], b[4], sel),

m5(x[5], a[5], b[5], sel),

m6(x[6], a[6], b[6], sel),

m7(x[7], a[7], b[7], sel),

m8(x[8], a[8], b[8], sel),

m9(x[9], a[9], b[9], sel),

m10(x[10], a[10], b[10], sel),

m11(x[11], a[11], b[11], sel),

m12(x[12], a[12], b[12], sel),

m13(x[13], a[13], b[13], sel),

m14(x[14], a[14], b[14], sel),

m15(x[15], a[15], b[15], sel);

endmodule

## module mux\_64\_to\_16(x, a, b, c, d, sel);

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

input [1:0] sel;

output [15:0] x;

mux\_4\_to\_1 m0(x[0], a[0], b[0], c[0], d[0], sel),

m1(x[1], a[1], b[1], c[1], d[1], sel),

m2(x[2], a[2], b[2], c[2], d[2], sel),

m3(x[3], a[3], b[3], c[3], d[3], sel),

m4(x[4], a[4], b[4], c[4], d[4], sel),

m5(x[5], a[5], b[5], c[5], d[5], sel),

m6(x[6], a[6], b[6], c[6], d[6], sel),

m7(x[7], a[7], b[7], c[7], d[7], sel),

m8(x[8], a[8], b[8], c[8], d[8], sel),

m9(x[9], a[9], b[9], c[9], d[9], sel),

m10(x[10], a[10], b[10], c[10], d[10], sel),

m11(x[11], a[11], b[11], c[11], d[11], sel),

m12(x[12], a[12], b[12], c[12], d[12], sel),

m13(x[13], a[13], b[13], c[13], d[13], sel),

m14(x[14], a[14], b[14], c[14], d[14], sel),

m15(x[15], a[15], b[15], c[15], d[15], sel);

endmodule

## module register\_16\_bit(out, in, clk);

input [15:0] in;

input clk;

output [15:0] out;

d\_flip\_flop d0(in[0], clk, out[0]),

d1(in[1], clk, out[1]),

d2(in[2], clk, out[2]),

d3(in[3], clk, out[3]),

d4(in[4], clk, out[4]),

d5(in[5], clk, out[5]),

d6(in[6], clk, out[6]),

d7(in[7], clk, out[7]),

d8(in[8], clk, out[8]),

d9(in[9], clk, out[9]),

d10(in[10], clk, out[10]),

d11(in[11], clk, out[11]),

d12(in[12], clk, out[12]),

d13(in[13], clk, out[13]),

d14(in[14], clk, out[14]),

d15(in[15], clk, out[15]);

endmodule

## module register\_file\_4\_by\_16(read\_data1, read\_data2, read\_reg1, read\_reg2, write\_reg, write\_data, reg\_write, clk);

input [1:0] read\_reg1, read\_reg2, write\_reg;

input [15:0] write\_data;

input reg\_write, clk;

output [15:0] read\_data1, read\_data2;

wire [3:0] w, c;

wire p;

wire [15:0] q1, q2, q3;

decoder\_2\_to\_4 dec(w, write\_reg);

and g1(p, reg\_write, clk),

g2(c[1], p, w[1]),

g3(c[2], p, w[2]),

g4(c[3], p, w[3]);

register\_16\_bit r1(q1, write\_data, c[1]),

r2(q2, write\_data, c[2]),

r3(q3, write\_data, c[3]);

mux\_64\_to\_16 m1(read\_data1, 0, q1, q2, q3, read\_reg1),

m2(read\_data2, 0, q1, q2, q3, read\_reg2);

endmodule

## module MainControl(Op,Control);

input [3:0] Op;  
 output reg [9:0] Control;

always @(Op) case (Op)  
 // {RegDst, BEQ, BNE, MemtoReg, MemWrite, ALUSrc, RegWrite, ALUctl[2], ALUctl[1], ALUctl[0]}  
 4'b0000: Control <= 10'b1000001010; // add  
 4'b0001: Control <= 10'b1000001110; // sub   
 4'b0010: Control <= 10'b1000001000; // and   
 4'b0011: Control <= 10'b1000001001; // or   
 4'b0100: Control <= 10'b0000011010; // addi   
 4'b0101: Control <= 10'b0001111010; // lw   
 4'b0110: Control <= 10'b0000110010; // sw   
 4'b0111: Control <= 10'b1000001111; // slt   
 4'b1000: Control <= 10'b0100000110; // beq  
 4'b1001: Control <= 10'b0010000110; // bne   
 endcase  
endmodule

## module CPU (clock, PC, IFID\_InstrReg, IDEX\_InstrReg, WriteData);

module CPU (clock, PC, IFID\_InstrReg, IDEX\_InstrReg, WriteData);

input clock;

output [15:0] WriteData, IFID\_InstrReg, IDEX\_InstrReg;

output [15:0] PC;

// IF

wire [15:0] NextPC;

reg [15:0] PC, IFID\_InstrReg;

reg [15:0] InstrMem[0:511];

alu\_16\_bit fetch(NextPC, unused1, unused2, unused3, PC, 1, 3'b010);

// Read hex data from text file into InstrMem

initial

$readmemh("input3a.hex", InstrMem); //change to input3b.hex for run2

// ID

reg [15:0] IDEX\_InstrReg;

wire [5:0] control;

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

reg [2:0] IDEX\_ALUctl;

wire [15:0] ReadData1, ReadData2, SignExtend, WriteData;

reg [15:0] IDEX\_ReadData1, IDEX\_ReadData2, IDEX\_SignExtend;

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

register\_file\_4\_by\_16 regFile(ReadData1, ReadData2, IFID\_InstrReg[11:10], IFID\_InstrReg[9:8], WriteReg, WriteData, IDEX\_RegWrite, clock);

main\_control mainCtrl(IFID\_InstrReg[15:12], control);

assign SignExtend = {{8{IFID\_InstrReg[7]}},IFID\_InstrReg[7:0]}; // sign extension

// EXE

wire [15:0] B, ALUout;

wire [1:0] WriteReg;

alu\_16\_bit ex(ALUout, unused4, unused5, unused6, IDEX\_ReadData1, B, IDEX\_ALUctl);

mux\_4\_to\_2 mux1(WriteReg, IDEX\_rt, IDEX\_rd, IDEX\_RegDst);

mux\_32\_to\_16 mux2(B, IDEX\_ReadData2, IDEX\_SignExtend, IDEX\_ALUSrc);

assign WriteData = ALUout;

initial PC = 0;

always @(negedge clock) begin

// Stage 1 -- IF

PC <= NextPC;

IFID\_InstrReg <= InstrMem[PC];

// Stage 2 -- ID

IDEX\_InstrReg <= IFID\_InstrReg;

{IDEX\_RegDst, IDEX\_ALUSrc, IDEX\_RegWrite, IDEX\_ALUctl} <= control;

IDEX\_ReadData1 <= ReadData1;

IDEX\_ReadData2 <= ReadData2;

IDEX\_SignExtend <= SignExtend;

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

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

// Stage 3 -- EX

// do nothing.

end

endmodule

// Test module

module test ();

reg clock;

wire [15:0] PC, IFID\_InstrReg, IDEX\_InstrReg, WriteData;

CPU test\_cpu(clock, PC, IFID\_InstrReg, IDEX\_InstrReg, WriteData);

always #1 clock = ~clock;

initial begin

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

$monitor ("%2d %3d %h %h %h", $time, PC, IFID\_InstrReg, IDEX\_InstrReg, WriteData);

clock = 1;

#29 $finish;

end

endmodule

# III. Testing & Results

## Testing Code

### input3a.hex

410f // addi $1, $0, 15 ==> 0100000100001111

4207 // addi $2, $0, 7 ==> 0100001000000111

0000 // nop ==> 0000000000000000

26c0 // and $3, $1, $2 ==> 0010011011000000

0000 // nop ==> 0000000000000000

1780 // sub $2, $1, $3 ==> 0001011110000000

0000 // nop ==> 0000000000000000

3b80 // or $2, $2, $3 ==> 0011101110000000

0000 // nop ==> 0000000000000000

0bc0 // add $3, $2, $3 ==> 0000101111000000

0000 // nop ==> 0000000000000000

7e40 // slt $1, $3, $2 ==> 0111111001000000

7b40 // slt $1, $2, $3 ==> 0111101101000000

0000 // nop ==> 0000000000000000

### input3b.hex

410f // addi $1, $0, 15 ==> 0100000100001111

4207 // addi $2, $0, 7 ==> 0100001000000111

26c0 // and $3, $1, $2 ==> 0010011011000000

1780 // sub $2, $1, $3 ==> 0001011110000000

3b80 // or $2, $2, $3 ==> 0011101110000000

0bc0 // add $3, $2, $3 ==> 0000101111000000

7e40 // slt $1, $3, $2 ==> 0111111001000000

7b40 // slt $1, $2, $3 ==> 0111101101000000

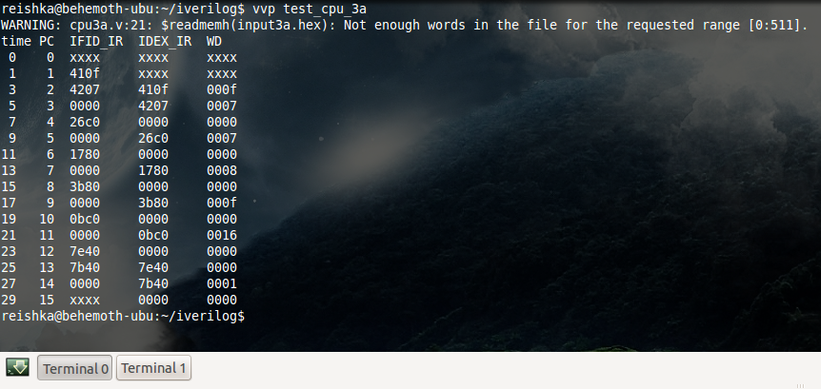
0000 // nop ==> 0000000000000000

## Testing Results

The test program performs a series of R-type instructions, and two addi instructions. The simulation is executed twice: The first run accounts for hazards, as nops have been inserted into input3a.hex by hand. The second run does not account for hazards.   
Please note that the warning is intentional. Since the code is reading the instructions from a file directly into a reg vector, it's leaving most of the vector uninitialized, thus causing the warning.

### Execution1

#### Screenshot



#### Plaintext

reishka@behemoth-ubu:~/iverilog$ vvp test\_cpu\_3a

WARNING: cpu3a.v:21: $readmemh(input3a.hex): Not enough words in the file for the requested range [0:511].

time PC IFID\_IR IDEX\_IR WD

0 0 xxxx xxxx xxxx

1 1 410f xxxx xxxx

3 2 4207 410f 000f

5 3 0000 4207 0007

7 4 26c0 0000 0000

9 5 0000 26c0 0007

11 6 1780 0000 0000

13 7 0000 1780 0008

15 8 3b80 0000 0000

17 9 0000 3b80 000f

19 10 0bc0 0000 0000

21 11 0000 0bc0 0016

23 12 7e40 0000 0000

25 13 7b40 7e40 0000

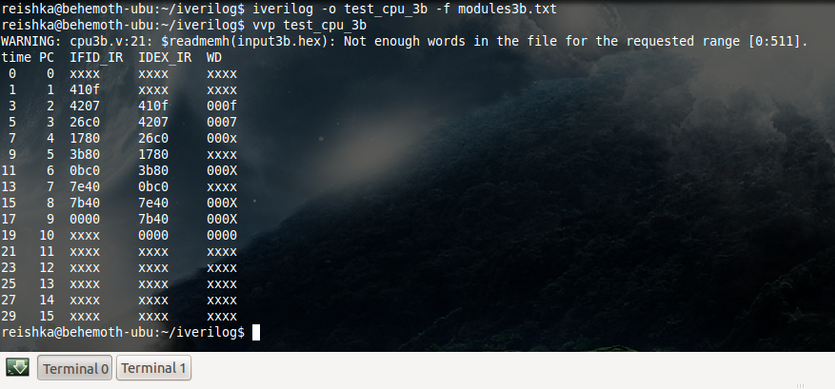
27 14 0000 7b40 0001

29 15 xxxx 0000 0000

reishka@behemoth-ubu:~/iverilog$

### Execution 2

#### Screenshot



#### Plaintext

reishka@behemoth-ubu:~/iverilog$ iverilog -o test\_cpu\_3b -f modules3b.txt

reishka@behemoth-ubu:~/iverilog$ vvp test\_cpu\_3b

WARNING: cpu3b.v:21: $readmemh(input3b.hex): Not enough words in the file for the requested range [0:511].

time PC IFID\_IR IDEX\_IR WD

0 0 xxxx xxxx xxxx

1 1 410f xxxx xxxx

3 2 4207 410f 000f

5 3 26c0 4207 0007

7 4 1780 26c0 000x

9 5 3b80 1780 xxxx

11 6 0bc0 3b80 000X

13 7 7e40 0bc0 xxxx

15 8 7b40 7e40 000X

17 9 0000 7b40 000X

19 10 xxxx 0000 0000

21 11 xxxx xxxx xxxx

23 12 xxxx xxxx xxxx

25 13 xxxx xxxx xxxx

27 14 xxxx xxxx xxxx

29 15 xxxx xxxx xxxx

reishka@behemoth-ubu:~/iverilog$