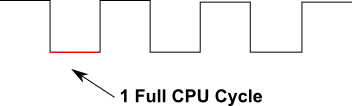
CS385 – Final Report

Robert Rotaru, Bryan Bigelow, Anthony Cerritelli

# Introduction

# Architecture

**Description –** The CPU is built using a 16-bit architecture that includes pipelining – meant to improve the cycle efficiency and protect against certain kinds of hazards. It follows the standard single cycle datapath. What this means is that areas of the CPU ready to execute run on a single clock cycle. This happens during a negative edge of the clock.

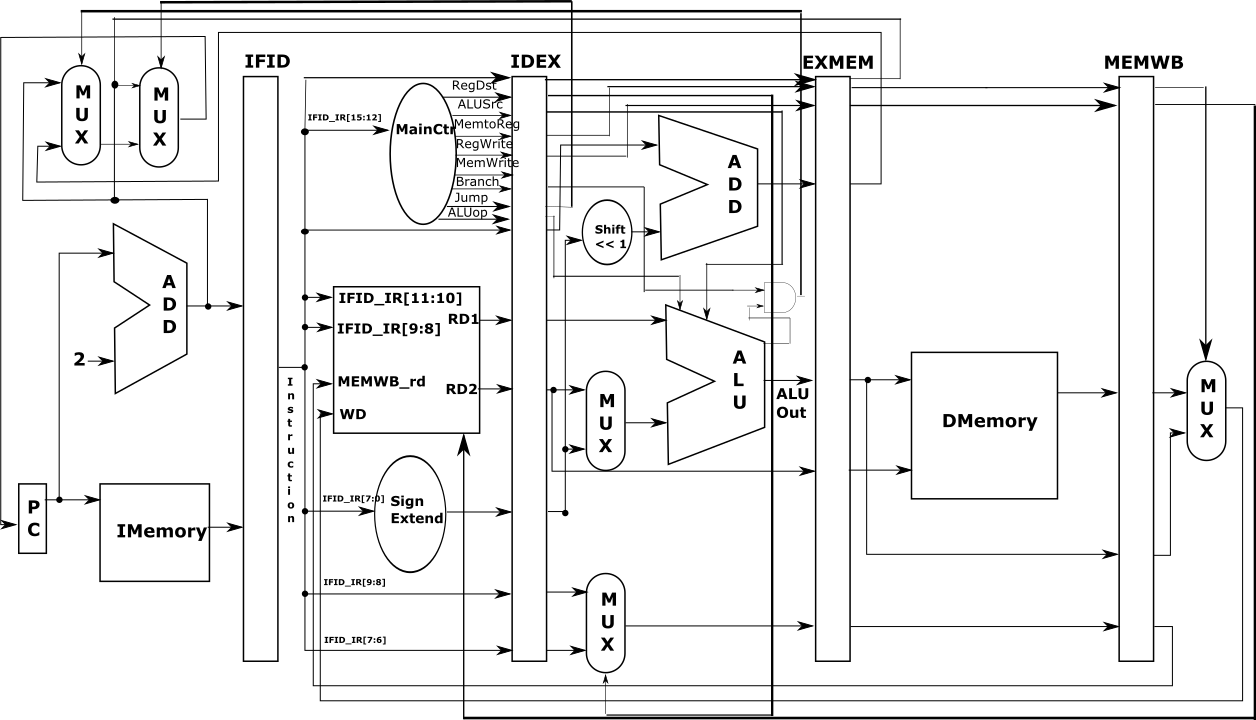


The execution will occur within five distinct areas:

* IF – Includes instruction memory and program counter
* ID – Includes registers, instruction controllers, and branching logic
* EX – Includes the ALU
* MEM – Includes the data memory of the program
* WB – Includes the write back logic

Registers are defined based on a 2-bit number with a max of 4 registers. Only 3 of those registers are available for writing. The last one, the 0 register, is read only for the constant 0.

**Final Datapath –**

****

# Instruction Set

|  |  |  |
| --- | --- | --- |
| **Instruction Set** | | **Instruction Format** |
| add | 0000 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | op | rs | rt | rd | Unused | | 4 | 2 | 2 | 2 | 6 | |
| sub | 0001 |
| and | 0010 |
| or | 0011 |
| slt | 0111 |
| addi | 0100 | |  |  |  |  | | --- | --- | --- | --- | | op | rs | rt | value / address | | 4 | 2 | 2 | 8 | |
| sw | 0110 |
| lw | 0101 |
| beq | 1000 | |  |  | | --- | --- | | op | address | | 4 | 12 | |
| bne | 1001 |
| j | 1010 |

# Components

## Program Counter

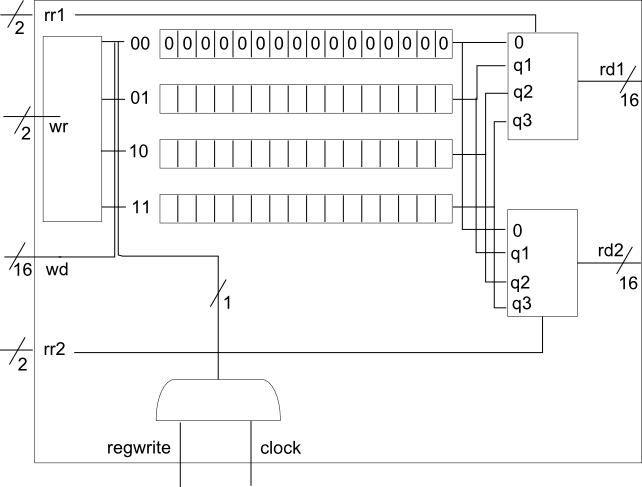
**Description** – The program counter is a requirement for running programs greater than a single instruction. It determines the current location of the program by adding the current size of an instruction to the current value of the program counter. Additionally, to support breaks in the cycle using loops or logic statements it must be capable of being set.

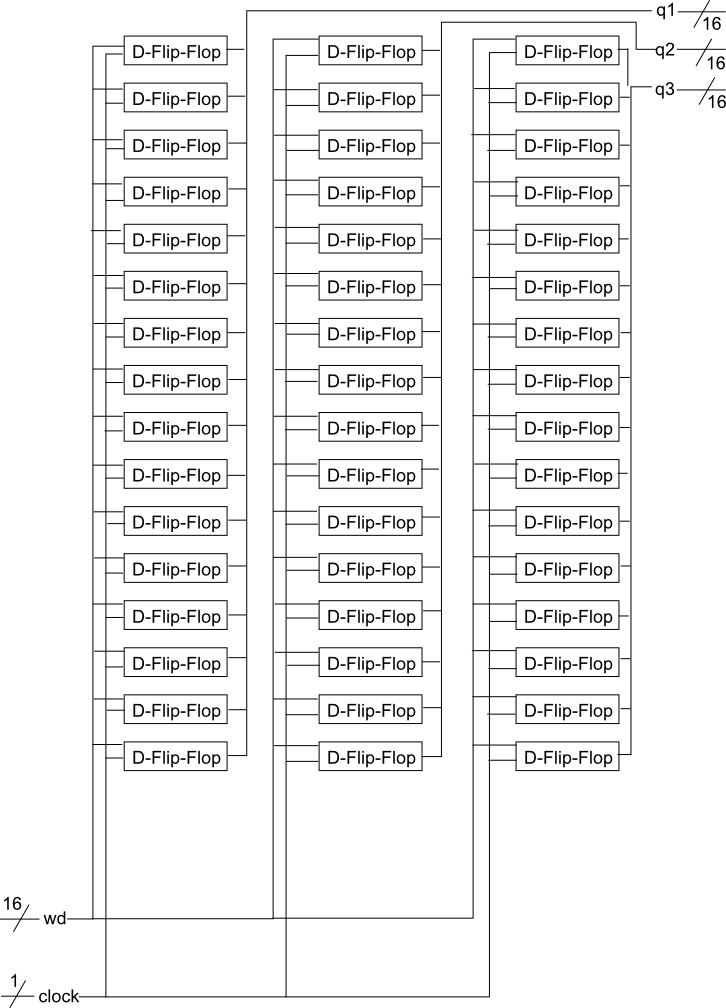
## Registers

**Description –** Registers store and maintain data for use between each clock pulse cycle. They are used both in reading data and for writing back into registers for future use. Because of this, they are one of the few pieces of the CPU utilized in multiple cycles and is more prone to hazards in a pipelined data path.

This CPU design utilizes a 16-bit D-flip-flop architecture for register storage and retrieval. This is achieved by chaining together 1-bit D-flip-flops on a negative edge change. Additionally, each D-flip-flop is actually pieced together from two D-latchs.

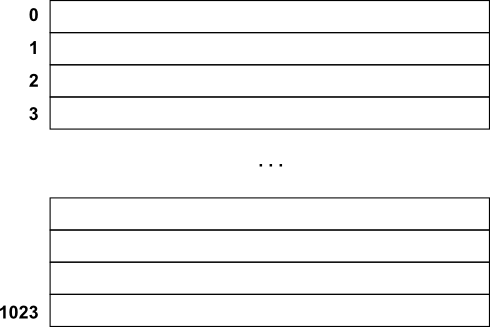
**Diagram –**



****

## Instruction / Data Memory

**Description –** Data memory is implemented as an array of 16-bit storage locations for data. The data is stored within these array indexes, referenced, and read or written to in the MEM section of the CPU cycle. Indices are calculated based on the program counter shifted by the size of the program counter increase.



Instruction memory is handled similarly – though it requires less functionality as it is meant only to be retrieved and not written to.

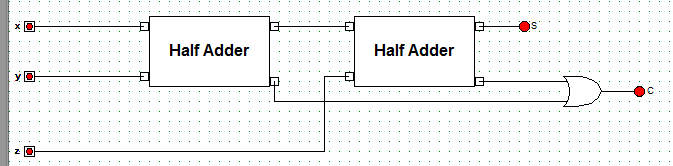
## ALU

**Description –** The Arithmetic Logic Unit, or ALU, is the workhorse of the CPU producing the results of different operations – with the exception of branch and memory instructions. Depending upon the Opcode it may produce several different instructions with the supplied register data.

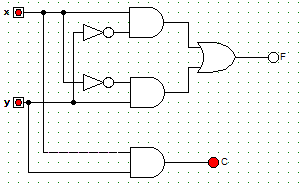
This CPU uses a 16-bit ALU built upon a hierarchical style. The 16-bit ALU is built upon 16, 1-bit ALU’s (including a special most significant bit ALU). Those 1 bit ALU’s are made using a full-adder that in turn is made up of half-adders.

**Diagram –**

Full Adder



Half Adder



**Truth Table(s) –**

Half Adder

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | C | S |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

## Pipelines

**Description –** The four pipelines of the single cycle data path CPU provide monitoring, hazard protection, and performance improvements to the running of programs:

* Monitoring – Storing certain values within the pipeline and transferring it forward to the next level pipelines, the CPU may monitor the instructions as they pass through.
* Hazard Protection – By storing certain pieces of information between cycles it can be used in future cycles before it completely makes its way through the CPU. By doing this there is a large decrease in hazards for the program.
* Performance Improvements – By utilizing the hazard protection explained above there is less of a need for artificial no operations (dummy instructions) to introduce delays for the CPU to catch up.

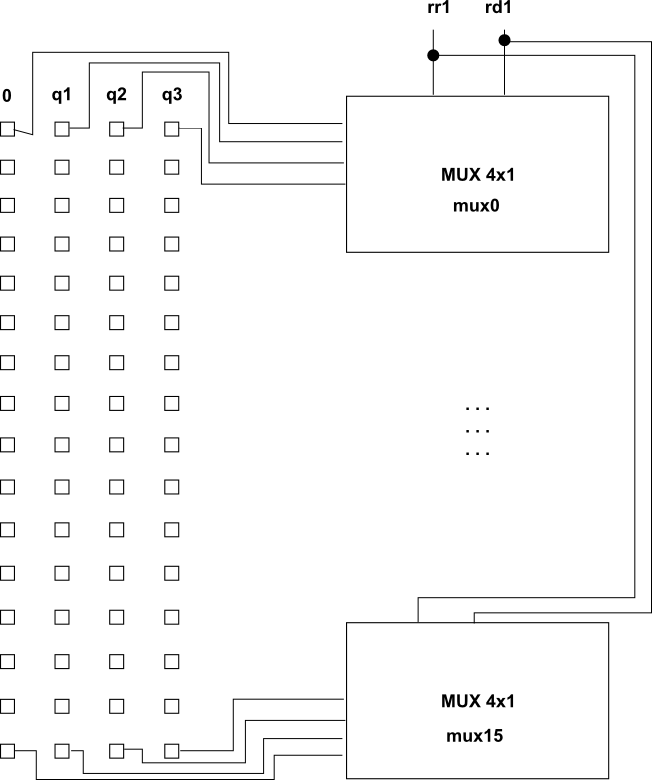
## Multiplexers

**Description –** Several different kinds of multiplexers are contained within the CPU architecture. They provide support for selecting appropriate values from several potential values based on things like instruction operation codes, write back enabled, etc.

A 16-bit 4x1 multiplexer, made from chained 1-bit 4x1 multiplexers, is utilized within the register file to determine register data. Additionally, a 16-bit 2x1 multiplexer is used for data in the WB phase of the CPU.

**Diagram –**

16 bit 4x1 Mux (Register File)



**Truth Table(s) –**

4x1 Multiplexer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s1 | s2 | x0 | x1 | x2 | x3 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

# Program

/\* 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 [3:0] op;

output reg [11:0] ctrl;

always @(op) case (op)

//IDEX\_RegDst,IDEX\_ALUSrc[1:0],IDEX\_MemtoReg,IDEX\_RegWrite,IDEX\_MemWrite,IDEX\_Branch[1:0],IDEX\_Jump,IDEX\_ALUop[2:0]

4'b0000: ctrl <= 12'b100010000010; // ADD

4'b0001: ctrl <= 12'b100010000110; // SUB

4'b0010: ctrl <= 12'b100010000000; // AND

4'b0011: ctrl <= 12'b100010000001; // OR

4'b0111: ctrl <= 12'b100010000111; // SLT

4'b0101: ctrl <= 12'b001110000010; // LW

4'b0110: ctrl <= 12'b001001000010; // SW

4'b1000: ctrl <= 12'b000000010110; // BEQ

4'b1001: ctrl <= 12'b000000100110; // BNE

4'b0100: ctrl <= 12'b001010000010; // ADDI

4'b1111: ctrl <= 12'b010010000001; // LUI

4'b1010: ctrl <= 12'b010010001010; // J

endcase

endmodule

module JumpAddress (PCplus2, IDEX\_IR, Jump\_Address);

input [15:0] PCplus2;

input [15:0] IDEX\_IR;

output [15:0] Jump\_Address;

assign Jump\_Address = {PCplus2[15:12], IDEX\_IR[11:0]};

endmodule

module ALUSrcControl (ALUSrc, RD2, SignExt, ShiftToUpper, ALUSrcOp);

input [15:0] RD2, SignExt, ShiftToUpper;

input [1:0] ALUSrcOp;

output [15:0] ALUSrc;

wire w0;

wire w1\_0,w1\_1,w1\_2,w1\_3,w1\_4,w1\_5,w1\_6,w1\_7,w1\_8,w1\_9,w1\_10,w1\_11,w1\_12,w1\_13,w1\_14,w1\_15;

wire w2\_0,w2\_1,w2\_2,w2\_3,w2\_4,w2\_5,w2\_6,w2\_7,w2\_8,w2\_9,w2\_10,w2\_11,w2\_12,w2\_13,w2\_14,w2\_15;

wire w3\_0,w3\_1,w3\_2,w3\_3,w3\_4,w3\_5,w3\_6,w3\_7,w3\_8,w3\_9,w3\_10,w3\_11,w3\_12,w3\_13,w3\_14,w3\_15;

nor nor1(w0, ALUSrcOp[0], ALUSrcOp[1]);

and and1\_0 (w1\_0, w0, RD2[0]); and and1\_1 (w1\_1, w0, RD2[1]);

and and1\_2 (w1\_2, w0, RD2[2]); and and1\_3 (w1\_3, w0, RD2[3]);

and and1\_4 (w1\_4, w0, RD2[4]); and and1\_5 (w1\_5, w0, RD2[5]);

and and1\_6 (w1\_6, w0, RD2[6]); and and1\_7 (w1\_7, w0, RD2[7]);

and and1\_8 (w1\_8, w0, RD2[8]); and and1\_9 (w1\_9, w0, RD2[9]);

and and1\_10(w1\_10, w0, RD2[10]); and and1\_11(w1\_11, w0, RD2[11]);

and and1\_12(w1\_12, w0, RD2[12]); and and1\_13(w1\_13, w0, RD2[13]);

and and1\_14(w1\_14, w0, RD2[14]); and and1\_15(w1\_15, w0, RD2[15]);

and and2\_0 (w2\_0, ALUSrcOp[0], SignExt[0]); and and2\_1 (w2\_1, ALUSrcOp[0], SignExt[1]);

and and2\_2 (w2\_2, ALUSrcOp[0], SignExt[2]); and and2\_3 (w2\_3, ALUSrcOp[0], SignExt[3]);

and and2\_4 (w2\_4, ALUSrcOp[0], SignExt[4]); and and2\_5 (w2\_5, ALUSrcOp[0], SignExt[5]);

and and2\_6 (w2\_6, ALUSrcOp[0], SignExt[6]); and and2\_7 (w2\_7, ALUSrcOp[0], SignExt[7]);

and and2\_8 (w2\_8, ALUSrcOp[0], SignExt[8]); and and2\_9 (w2\_9, ALUSrcOp[0], SignExt[9]);

and and2\_10(w2\_10, ALUSrcOp[0], SignExt[10]); and and2\_11(w2\_11, ALUSrcOp[0], SignExt[11]);

and and2\_12(w2\_12, ALUSrcOp[0], SignExt[12]); and and2\_13(w2\_13, ALUSrcOp[0], SignExt[13]);

and and2\_14(w2\_14, ALUSrcOp[0], SignExt[14]); and and2\_15(w2\_15, ALUSrcOp[0], SignExt[15]);

and and3\_0 (w3\_0, ALUSrcOp[1], ShiftToUpper[0]); and and3\_1 (w3\_1, ALUSrcOp[1], ShiftToUpper[1]);

and and3\_2 (w3\_2, ALUSrcOp[1], ShiftToUpper[2]); and and3\_3 (w3\_3, ALUSrcOp[1], ShiftToUpper[3]);

and and3\_4 (w3\_4, ALUSrcOp[1], ShiftToUpper[4]); and and3\_5 (w3\_5, ALUSrcOp[1], ShiftToUpper[5]);

and and3\_6 (w3\_6, ALUSrcOp[1], ShiftToUpper[6]); and and3\_7 (w3\_7, ALUSrcOp[1], ShiftToUpper[7]);

and and3\_8 (w3\_8, ALUSrcOp[1], ShiftToUpper[8]); and and3\_9 (w3\_9, ALUSrcOp[1], ShiftToUpper[9]);

and and3\_10(w3\_10, ALUSrcOp[1], ShiftToUpper[10]); and and3\_11(w3\_11, ALUSrcOp[1], ShiftToUpper[11]);

and and3\_12(w3\_12, ALUSrcOp[1], ShiftToUpper[12]); and and3\_13(w3\_13, ALUSrcOp[1], ShiftToUpper[13]);

and and3\_14(w3\_14, ALUSrcOp[1], ShiftToUpper[14]); and and3\_15(w3\_15, ALUSrcOp[1], ShiftToUpper[15]);

or(ALUSrc[0], w1\_0, w2\_0, w3\_0); or(ALUSrc[1], w1\_1, w2\_1, w3\_1);

or(ALUSrc[2], w1\_2, w2\_2, w3\_2); or(ALUSrc[3], w1\_3, w2\_3, w3\_3);

or(ALUSrc[4], w1\_4, w2\_4, w3\_4); or(ALUSrc[5], w1\_5, w2\_5, w3\_5);

or(ALUSrc[6], w1\_6, w2\_6, w3\_6); or(ALUSrc[7], w1\_7, w2\_7, w3\_7);

or(ALUSrc[8], w1\_8, w2\_8, w3\_8); or(ALUSrc[9], w1\_9, w2\_9, w3\_9);

or(ALUSrc[10], w1\_10, w2\_10, w3\_10); or(ALUSrc[11], w1\_11, w2\_11, w3\_11);

or(ALUSrc[12], w1\_12, w2\_12, w3\_12); or(ALUSrc[13], w1\_13, w2\_13, w3\_13);

or(ALUSrc[14], w1\_14, w2\_14, w3\_14); or(ALUSrc[15], w1\_15, w2\_15, w3\_15);

endmodule

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

input clock;

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

/\* Test Program \*/

initial begin

// Simple program to load 1 from DMemory[0] into $1 and 3 from

// DMemory[1] into $2, add $1 to $2, add 1 to $1, then swaps

// the values in $1 and $2 and loops to decrement $2 by $1 until

// $2 reaches 0. Then stores 3 in DMemory[0] and 1 in DMemory[1],

// jumps to the begining and loads $2 with 1 and $1 with 3 then

// branches to then end when $2 < $ 1.

IMemory[0] = 16'b0000000000000000; // nop

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

IMemory[2] = 16'b0101001000000010; // lw $2, 2($0) 3 Load DMemory[1] into $2

IMemory[3] = 16'b0000000000000000; // nop

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

IMemory[5] = 16'b0000000000000000; // nop

IMemory[6] = 16'b0000100110000000; // add $2, $2, $1 4

IMemory[7] = 16'b0100010100000001; // addi $1, $1, 1 2

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

IMemory[9] = 16'b0000000000000000; // nop

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

IMemory[11] = 16'b0111011011000000; // slt $3, $1, $2 1 Set $3 on $1 < $2

IMemory[12] = 16'b0000000000000000; // nop

IMemory[13] = 16'b0000000000000000; // nop

IMemory[14] = 16'b0000000000000000; // nop

IMemory[15] = 16'b1000110000101001; // beq $3, $0, 39 X Branch to IMemory[56] if $3 == 0

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

IMemory[17] = 16'b0000000000000000; // nop

IMemory[18] = 16'b0000000000000000; // nop

IMemory[19] = 16'b0110000100000010; // sw $1, 2($0) X Store $1 into DMemory[1]

IMemory[20] = 16'b0110001000000000; // sw $2, 0($0) X Store $2 into DMemory[0]

IMemory[21] = 16'b0000000000000000; // nop

IMemory[22] = 16'b0000000000000000; // nop

IMemory[23] = 16'b0101000100000000; // lw $1, 0($0) 4 Load DMemory[0] into $1

IMemory[24] = 16'b0101001000000010; // lw $2, 2($0) 2 Load DMemory[1] into $2

IMemory[25] = 16'b0000000000000000; // nop

IMemory[26] = 16'b0000000000000000; // nop

IMemory[27] = 16'b0000000000000000; // nop

IMemory[28] = 16'b0001011001000000; // sub $1, $1, $2 2 $1 <- $1 - $2

IMemory[29] = 16'b0000000000000000; // nop

IMemory[30] = 16'b0000000000000000; // nop

IMemory[31] = 16'b0000000000000000; // nop

IMemory[32] = 16'b1001000111111011; // bne $1, $0, -5 X Branch to IMemory[27] if $1 != 0

IMemory[33] = 16'b0000000000000000; // nop

IMemory[34] = 16'b0000000000000000; // nop

IMemory[35] = 16'b0000000000000000; // nop

IMemory[36] = 16'b0110000100000010; // sw $1, 2($0) 0 Store $1 into DMemory[1]

IMemory[37] = 16'b1000010000000111; // beq $1, $0, 4 X Branch to IMemory[45] if $1 == 0

IMemory[38] = 16'b0000000000000000; // nop

IMemory[39] = 16'b0000000000000000; // nop

IMemory[40] = 16'b0000000000000000; // nop

IMemory[41] = 16'b0100000100000001; // addi $1, $0, 1 X Branched over

IMemory[42] = 16'b0100000100000001; // addi $1, $0, 1 X Branched over

IMemory[43] = 16'b0100000100000001; // addi $1, $0, 1 X Branched over

IMemory[44] = 16'b0100000100000001; // addi $1, $0, 1 X Branched over

IMemory[45] = 16'b0101001000000010; // lw $2, 2($0) 0 Load DMemory[1] into $2

IMemory[46] = 16'b0100000100000011; // addi $1, $0, 3

IMemory[47] = 16'b0100001000000001; // addi $2, $0, 1

IMemory[48] = 16'b0000000000000000; // nop

IMemory[49] = 16'b0000000000000000; // nop

IMemory[50] = 16'b0000000000000000; // nop

IMemory[51] = 16'b0110000100000000; // sw $1, 0($0)

IMemory[52] = 16'b0110001000000010; // sw $2, 2($0)

IMemory[53] = 16'b1010000000000000; // j

IMemory[54] = 16'b0000000000000000; // nop

IMemory[55] = 16'b0000000000000000; // nop

IMemory[56] = 16'b0000000000000000; // nop

// Data

DMemory [0] = 16'h1;

DMemory [1] = 16'h3;

end

// Pipeline stages

// IF

wire [15:0] PCplus2, NextPC;

wire [15:0] Jump\_Address;

wire [15:0] branch\_mux\_result;

reg[15:0] PC, IMemory[0:1023], IFID\_IR, IFID\_PCplus2;

ALU fetch (3'b010,PC,2,PCplus2,Unused1);

reg [1:0] EXMEM\_Branch;

reg EXMEM\_Zero;

reg [15:0] EXMEM\_Target;

branchCtrl bCtrl (EXMEM\_Branch, EXMEM\_Zero, branchOut);

mux16bit2x1 branchMux (PCplus2, EXMEM\_Target, branchOut, branch\_mux\_result);

JumpAddress jump\_add (PCplus2, IDEX\_IR, Jump\_Address);

mux16bit2x1 jumpMux (branch\_mux\_result, Jump\_Address, IDEX\_Jump, NextPC);

// ID

wire [11:0] Control;

reg IDEX\_RegWrite,IDEX\_MemtoReg,IDEX\_MemWrite,IDEX\_RegDst,IDEX\_Jump;

reg [1:0] IDEX\_Branch, IDEX\_ALUSrc;

reg [2:0] IDEX\_ALUOp;

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

reg [15:0] IDEX\_PCplus2,IDEX\_RD1,IDEX\_RD2,IDEX\_SignExt,IDEXE\_IR;

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

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

reg MEMWB\_RegWrite;

reg [1:0] MEMWB\_rd;

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

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

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

// EXE

reg EXMEM\_RegWrite,EXMEM\_MemtoReg,EXMEM\_MemWrite;

wire [15:0] Target;

reg [15:0] EXMEM\_ALUOut,EXMEM\_RD2;

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

reg [1:0] EXMEM\_rd;

wire [15:0] B,ALUOut;

wire [1:0] WR;

ALU branch (3'b010,IDEX\_SignExt<<1,IDEX\_PCplus2,Target,Unused2);

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

mux2bit2x1 RegDstMux (IDEX\_rt, IDEX\_rd, IDEX\_RegDst, WR); // RegDst Mux

ALUSrcControl ALUSrcControl1(B, IDEX\_RD2, IDEX\_SignExt, IDEX\_SignExt<<8, IDEX\_ALUSrc);

// MEM

reg MEMWB\_MemtoReg;

reg [15:0] DMemory[0:1023],MEMWB\_MemOut,MEMWB\_ALUOut;

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

wire [15:0] MemOut;

assign MemOut = DMemory[EXMEM\_ALUOut>>1];

always @(negedge clock) if (EXMEM\_MemWrite) DMemory[EXMEM\_ALUOut>>1] <= EXMEM\_RD2;

// WB

mux16bit2x1 Mem2Reg (MEMWB\_ALUOut, MEMWB\_MemOut, MEMWB\_MemtoReg, WD); // MemtoReg Mux

initial begin

PC = 0;

// Init registers

IDEX\_RegWrite=0;IDEX\_MemtoReg=0;IDEX\_Branch=0;IDEX\_Jump=0;IDEX\_MemWrite=0;IDEX\_ALUSrc=0;IDEX\_RegDst=0;IDEX\_ALUOp=0;

IFID\_IR=0;

EXMEM\_RegWrite=0;EXMEM\_MemtoReg=0;EXMEM\_Branch=0;EXMEM\_MemWrite=0;

EXMEM\_Target=0;

MEMWB\_RegWrite=0;MEMWB\_MemtoReg=0;

end

// Running the pipeline

always @(negedge clock) begin

// IF

PC <= NextPC;

IFID\_PCplus2 <= PCplus2;

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

// ID

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

{IDEX\_RegDst,IDEX\_ALUSrc,IDEX\_MemtoReg,IDEX\_RegWrite,IDEX\_MemWrite,IDEX\_Branch,IDEX\_Jump,IDEX\_ALUOp} <= Control;

IDEX\_PCplus2 <= IFID\_PCplus2;

IDEX\_RD1 <= RD1;

IDEX\_RD2 <= RD2;

IDEX\_SignExt <= SignExtend;

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

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

// EXE

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

EXMEM\_RegWrite <= IDEX\_RegWrite;

EXMEM\_MemtoReg <= IDEX\_MemtoReg;

EXMEM\_Branch <= IDEX\_Branch;

EXMEM\_MemWrite <= IDEX\_MemWrite;

EXMEM\_Target <= Target;

EXMEM\_Zero <= Zero;

EXMEM\_ALUOut <= ALUOut;

EXMEM\_RD2 <= IDEX\_RD2;

EXMEM\_rd <= WR;

// MEM

MEMWB\_IR <= EXMEM\_IR; // For monitoring the pipeline

MEMWB\_RegWrite <= EXMEM\_RegWrite;

MEMWB\_MemtoReg <= EXMEM\_MemtoReg;

MEMWB\_MemOut <= MemOut;

MEMWB\_ALUOut <= EXMEM\_ALUOut;

MEMWB\_rd <= EXMEM\_rd;

// WB

// Register write happens on neg edge of the clock (if MEMWB\_RegWrite is asserted)

end

endmodule

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

module test();

reg clock;

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

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

always #1 clock = ~clock;

initial begin

$display ("time\tPC\tIFID\_IR\tIDEX\_IR\tEXMEM\_IR\tMEMWB\_IR\tWD");

$monitor ("%2d\t%3d\t%h\t%h\t%h\t\t%h\t\t%d", $time,PC,IFID\_IR,IDEX\_IR,EXMEM\_IR,MEMWB\_IR,WD);

clock = 1;

#163 $finish;

end

endmodule

# Simulation Results

/\* Compiling and simulation

C:\Users\User\git\forks\CS385-CPU\Source Files\Final>iverilog -o out pipelined-datapath.v

C:\Users\User\git\forks\CS385-CPU\Source Files\Final>vvp out

time PC IFID\_IR IDEX\_IR EXMEM\_IR MEMWB\_IR WD

0 0 0000 xxxx xxxx xxxx x

1 2 0000 0000 xxxx xxxx x

3 4 5100 0000 0000 xxxx x

5 6 5202 5100 0000 0000 0

7 8 0000 5202 5100 0000 0

9 10 0000 0000 5202 5100 1

11 12 0000 0000 0000 5202 3

13 14 0980 0000 0000 0000 0

15 16 4501 0980 0000 0000 0

17 18 0000 4501 0980 0000 0

19 20 0000 0000 4501 0980 4

21 22 0000 0000 0000 4501 2

23 24 76c0 0000 0000 0000 0

25 26 0000 76c0 0000 0000 0

27 28 0000 0000 76c0 0000 0

29 30 0000 0000 0000 76c0 1

31 32 8c29 0000 0000 0000 0

33 34 0000 8c29 0000 0000 0

35 36 0000 0000 8c29 0000 0

37 38 0000 0000 0000 8c29 1

39 40 6102 0000 0000 0000 0

41 42 6200 6102 0000 0000 0

43 44 0000 6200 6102 0000 0

45 46 0000 0000 6200 6102 2

47 48 5100 0000 0000 6200 0

49 50 5202 5100 0000 0000 0

51 52 0000 5202 5100 0000 0

53 54 0000 0000 5202 5100 4

55 56 0000 0000 0000 5202 2

57 58 1640 0000 0000 0000 0

59 60 0000 1640 0000 0000 0

61 62 0000 0000 1640 0000 0

63 64 0000 0000 0000 1640 2

65 66 91fb 0000 0000 0000 0

67 68 0000 91fb 0000 0000 0

69 70 0000 0000 91fb 0000 0

71 56 0000 0000 0000 91fb 65534

73 58 1640 0000 0000 0000 0

75 60 0000 1640 0000 0000 0

77 62 0000 0000 1640 0000 0

79 64 0000 0000 0000 1640 0

81 66 91fb 0000 0000 0000 0

83 68 0000 91fb 0000 0000 0

85 70 0000 0000 91fb 0000 0

87 72 0000 0000 0000 91fb 0

89 74 6102 0000 0000 0000 0

91 76 8407 6102 0000 0000 0

93 78 0000 8407 6102 0000 0

95 80 0000 0000 8407 6102 2

97 90 0000 0000 0000 8407 0

99 92 5202 0000 0000 0000 0

101 94 4103 5202 0000 0000 0

103 96 4201 4103 5202 0000 0

105 98 0000 4201 4103 5202 0

107 100 0000 0000 4201 4103 3

109 102 0000 0000 0000 4201 1

111 104 6100 0000 0000 0000 0

113 106 6202 6100 0000 0000 0

115 108 a000 6202 6100 0000 0

117 110 0000 a000 6202 6100 0

119 0 0000 0000 a000 6202 2

121 2 0000 0000 0000 a000 0

123 4 5100 0000 0000 0000 0

125 6 5202 5100 0000 0000 0

127 8 0000 5202 5100 0000 0

129 10 0000 0000 5202 5100 3

131 12 0000 0000 0000 5202 1

133 14 0980 0000 0000 0000 0

135 16 4501 0980 0000 0000 0

137 18 0000 4501 0980 0000 0

139 20 0000 0000 4501 0980 4

141 22 0000 0000 0000 4501 4

143 24 76c0 0000 0000 0000 0

145 26 0000 76c0 0000 0000 0

147 28 0000 0000 76c0 0000 0

149 30 0000 0000 0000 76c0 0

151 32 8c29 0000 0000 0000 0

153 34 0000 8c29 0000 0000 0

155 36 0000 0000 8c29 0000 0

157 114 0000 0000 0000 8c29 0

159 116 xxxx 0000 0000 0000 0

161 118 xxxx xxxx 0000 0000 0

163 120 xxxx xxxx xxxx 0000 0

\*/