EE 471 Final Project
Designing a Single Cycle and Pipelined CPU

Cameron Forbis
Kendrick Tang
Alex Ching

ABSTRACT

This project involved building a single cycle MIPS CPU and a pipelined MIPS CPU with the traditional 5 stage execution pipeline. The system was designed in Verilog and test assembly programs were developed to verify the behavior and execution of the systems. For each CPU there were two variations of a sample C program that needed to be executed; the programs were assembled, linked, and run on the CPU. The signal tap logic analyzer allowed us to verify correct execution of the sample programs on both CPUs.

INTRODUCTION

The purpose of this project was to develop a single cycle MIPS CPU and then improve the implementation to a pipeline MIPS CPU. The Quartus development environment along with the DE-1 development board with Cyclone II FPGA were used in this design. Both projects were run and tested on the DE-1. Data memory for the systems was the SRAM chip on the DE-1 board, the instruction memory was implemented ad 32 by 128 RAM (to support the direct mapped cache with 4 word blocks).

DISCUSSION

Design Specification

Single Cycle CPU

The single cycle portion of the project had the following requirements:

- Support loading hand compiled C code into instruction memory
- Support register, immediate, and jump format instructions
- Support NOP, ADD, SUB, OR, XOR, SLT, SLL, LW, SW, J, JR, BGT instructions
- Appropriate data and control paths to allow instructions to complete in a single cycle
- Extra credit: Implement a 16 byte direct mapped instruction cache consisting of four four-byte blocks.

Pipelined CPU

The pipeline CPU project had the following requirements:

- Continue to support the same instruction set and addressing modes as the previous single cycle project.
- Modify the implementation to support a pipelined datapath
- Handle data hazards using forwarding and stalls

- Handle control hazards by computing branches in the ID stage and modifying the next address (delayed branches)
- Implement the same cache as the one in the single cycle CPU.

Design Procedure

Single Cycle CPU

The design started with an ALU, register file and SRAM driver. To allow for the five stages (fetch, decode, execute, memory access, write back) several control blocks were implemented to enable communication between the ALU, register file and SRAM driver. To ensure single cycle instructions, the controls used only combinational logic. The only new component that was synchronous to the clock was the program counter.

Each instruction needs to be decoded according to MIPS formatting and provide a subset of the inputs for the control blocks which then decide the datapath. Most of the control logic utilized XOR logic for equality, basic logical ANDs and ORs, and several MUXs to change the datapath depending on the instruction. In addition to the instruction, the flags on the ALU were utilized as control inputs. Specifically, the zero flag would be used with a SLT instruction to determine branching.

Pipelined CPU

For the pipeline CPU we started with a working single cycle MIPS CPU implementation. To begin we developed the pipelined datapath by separating the modules and control logic into the 5 MIPS pipeline phases: fetch, decode, execute, memory access, write back.

Next the group worked to handle the many data hazards. The hazards occurred when an instruction needed data from memory or a register file which a previous unfinished instruction had not yet written. To solve this we forwarded values from previous instruction to the stages where they were needed. Special control signals were created and driven by the forwarding unit module to manage when to forward these values.

In certain cases it was not possible to solve the data hazards by forwarding. When an instruction needs to read the value of a prior load word instruction, the value is not available until the memory access phase. In this case it is necessary to stall the pipeline by inserting a nop into the next execute stage and re-fetching the same instruction.

The last part of the project was handling the control hazards. These hazards occur because the pipeline must be constantly fed with instructions but the instructions after a branch are not known immediately. The branch logic was duplicated and moved to the decode phase where the branch is computed and executed. Because it still takes one cycle to compute the branch, the instruction after the branch is always executed. This is called a branch delay slot. In the test program we inserted a nop instruction afterwards, but if a more useful instruction exists, it can be used instead.

System Description

The public interface to the single cycle CPU and pipelined CPU are essentially identical. Both CPUs implement 32-bit MIPS formatted instructions and read and write to the SRAM 16-bit sign extended data and 16-bit data derived from 32-bit data with the 16 most significant bits chopped off. The CPUs also support the same instruction set, so they have the same capabilities. From this viewpoint, both CPUs have the same input and outputs.

The differences between the two CPU designs are more noticeable when looking at the timing and through-put of the CPU. The single cycle CPU has a very long datapath for each clock cycle since it has to go through all five stages in each clock cycle. The pipelined CPU has a much shorter datapath because the data only has to propagate through one stage at each clock cycle. As a result, the system clock can run much faster on a pipelined CPU. Furthermore, the single cycle CPU performs only one instruction at a time, while the pipelined CPU, as the name suggests, is working on multiple instructions at any given time. Typically a different instruction in each stage. As a result, the pipelined CPU will need monitor and adjust the pipeline for different data and control hazards - something which the single cycle CPU did not have to do.

Hardware Implementation

Single Cycle CPU

The top level design for the single cycle CPU is presented in Figure 1. The system datapath consists of the register file ALU, SRAM driver (data memory), and PC Next-er. The PC Next-er module consists of the logic required to compute the next PC value. It determines if branching, or jumping occurs and selects the appropriate value with multiplexors.

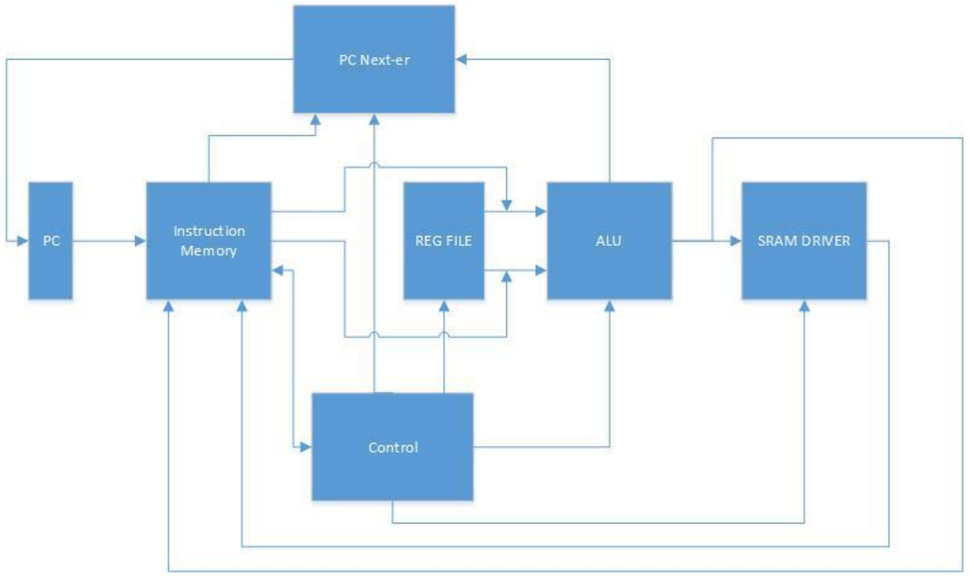

Figure 1: Top level diagram of single cycle CPU implementation.

Pipeline CPU

The top level design for the pipeline CPU is presented in Figure 2. The system is similar to the top level module for the single cycle CPU. The new components are the four pipeline registers that now separate the stages, the forwarding unit, and the hazard detection unit. The decoding and control has been moved to separate modules (this requires that the instruction be passed along unaltered between stages).

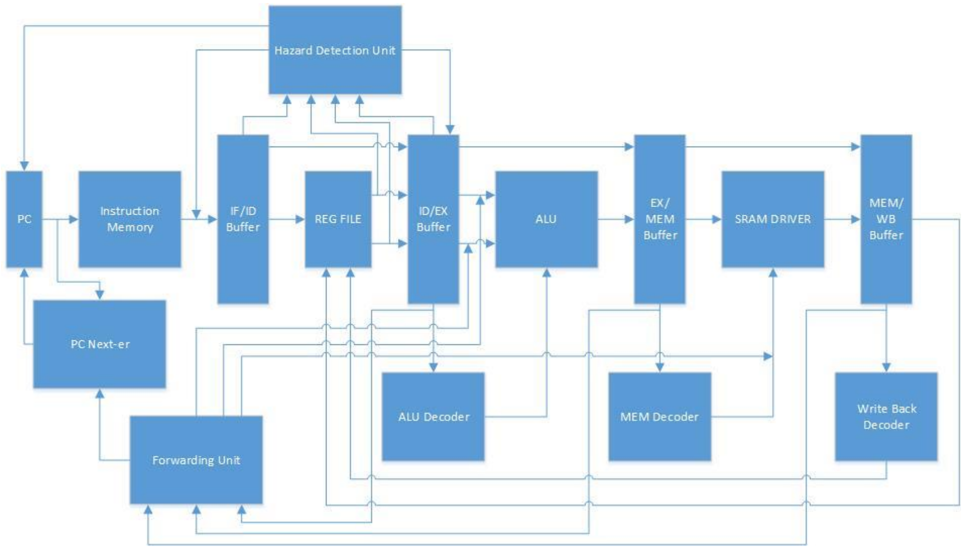

Figure 2: Top level diagram for pipelined CPU

TEST PLAN

Single Cycle CPU

For the single cycle CPU we had a test C program. Our expected behavior would be the correct execution of this program on the CPU by updating the correct variables in memory.

To test, signal tap was used to tap into the control signals and various points in the data path such as the inputs to the ALU and the bus to the SRAM chip. When a certain value or execution was incorrect we worked backwards to track the errant signal down. Testing continued in this fashion until the expected results were seen being written to SRAM.

Pipeline CPU

For the pipeline CPU our test consisted of three stages. First we tested the pipeline data path with no hazards. Next the support for data hazards by forwarding and stalling. Lastly we tested the control hazard handling.

At each stage we wrote test assembly programs and tested the execution results by writing values to memory. If the values were incorrect, we worked backwards to identify the cause. If the functionality is correct it is expected that pipeline CPU would behave like the single cycle CPU. One exception in the behavior would be the branch instruction has a delay slot of one instruction which is always executed.

TEST SPECIFICATION

Single Cycle CPU

The tests were assembly files that were assembled into binary and run by the CPU. The c program was compiled into assembly, assembled into a hex memory initialization file, and linked with the correct addresses. The test program was run on the CPU, when incorrect values were encountered, we back-tracked until the source of the error was found and corrected.

Pipeline CPU

Testing for the pipeline CPU was a multi step process. The PartD assembly file included simple R type instructions separated by many nop instructions to ensure no hazards occurred. This ensured correct execution on instruction along a pipeline datapath (even though this datapath did not yet handle hazards).

Next the group assembled a simple test assembly file called PartE that assessed the forwarding behavior for simple instruction cases. Testing continued as normal by backtracking to identify the source of errant signals. This simple test program handles many of the situations where forwarding is required.

Lastly the group tested the final behavior by assembling the test C program from the lab document. The final tests continued as before, but data forwarding and stalling had been mostly solved so the effort went into correcting the control hazard caused by the branch and jump instructions.

All the test C and assembly code are in the appendix, labelled appropriately.

TEST CASES

For both the single cycle and pipeline CPU, simple test programs were assembled and used as input to the system. The execution of these programs were verified in signal tap. If an incorrect value appeared, the execution was backtracked until the errant signal was discovered. Testing continues in this manner until the execution of the program is correct and the expected outputs are shown as written to SRAM.

RESULTS

Single Cycle CPU Signal Tap Results

The following Figures depict the writing of sample data to SRAM and the execution of the program for part A (the sample program with A = 7 and B = 5). Part B is the same sample C program but with A = 8 and B = 4. Figures 3 and 4 depict writing data and execution respectively for part A. Figures 5 and 6 depict writing data and execution for part B.

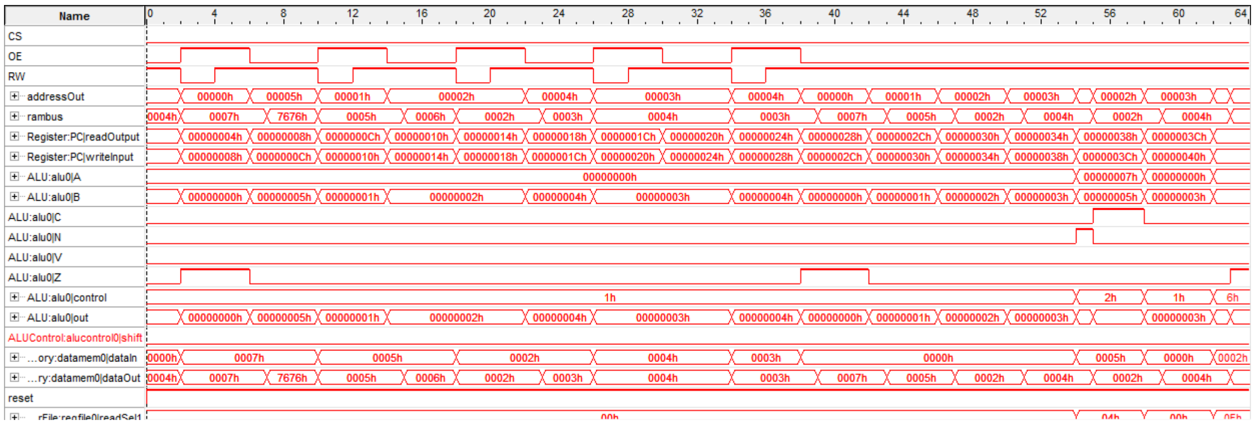

Figure 3: Single cycle CPU part A writing test data to SRAM

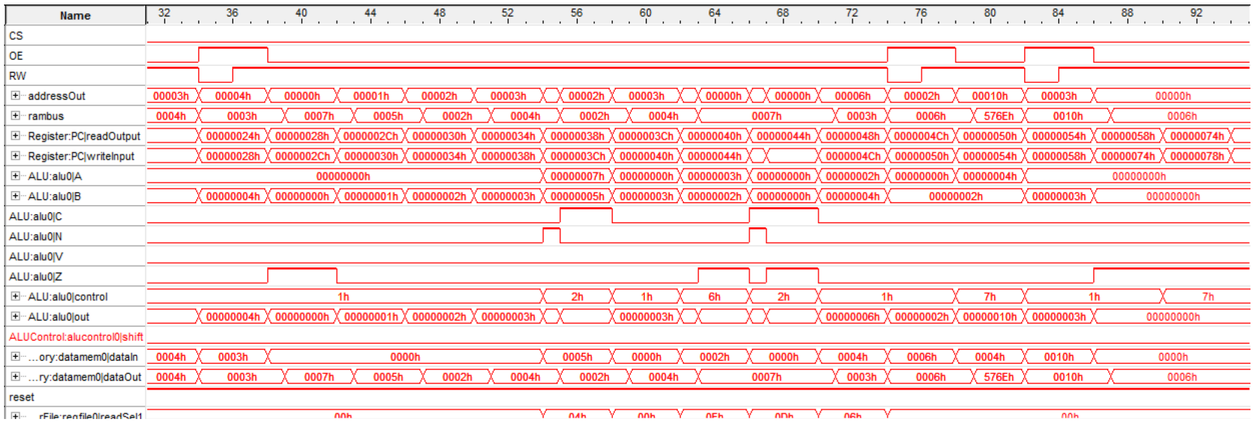

Figure 4: Part A computation and results

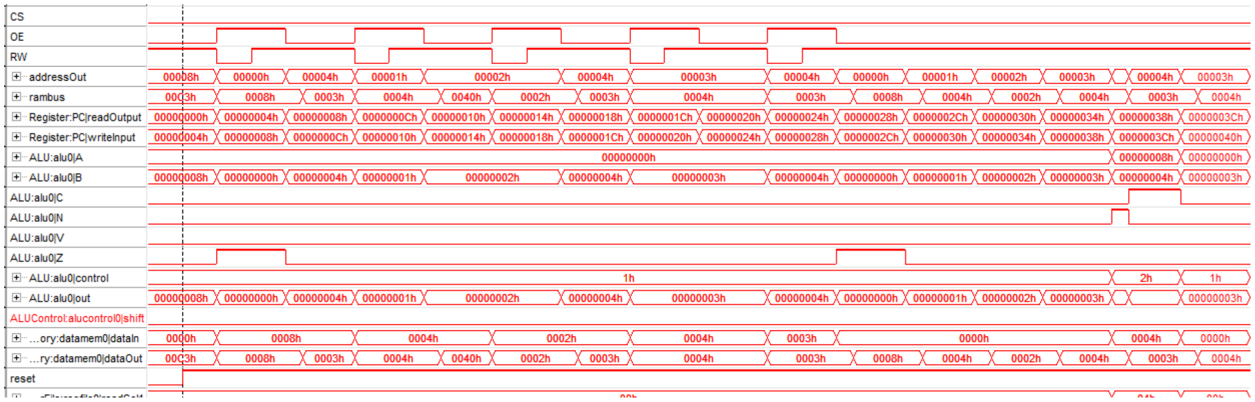

Figure 5: Single cycle CPU part B writing test data to SRAM

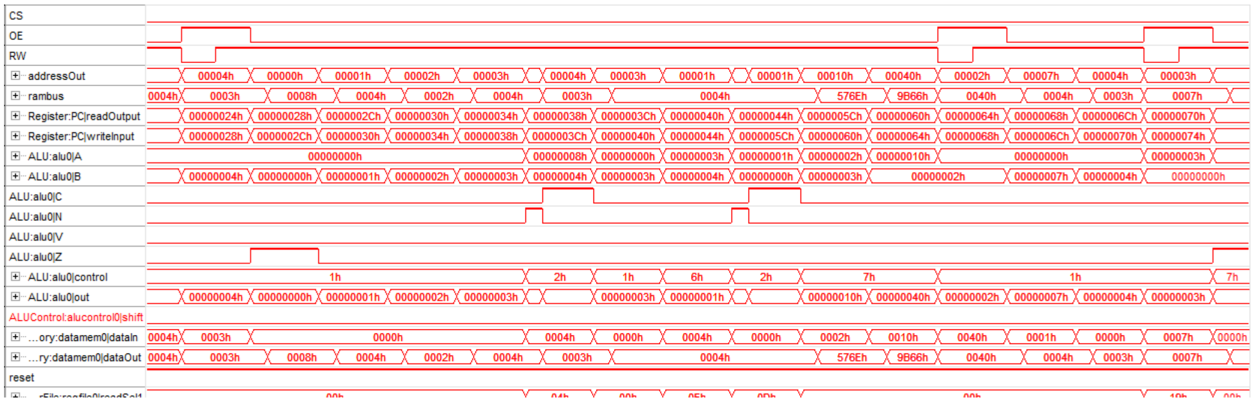

Figure 6: Part B computation results

Pipeline CPU Signal Tap Results

The following figures depict the execution of a different sample C program for the pipeline CPU. The new sample program has part A where B variable is initially assigned 4 and part B where B variable is assigned 2. Figures 7 to 9 show writing the test data, branching and execution, and writing the updated values to SRAM for Part A. Figures 10 to 12 show writing the test data, not branching, jumping and executing, and writing the updated values to SRAM for part B.

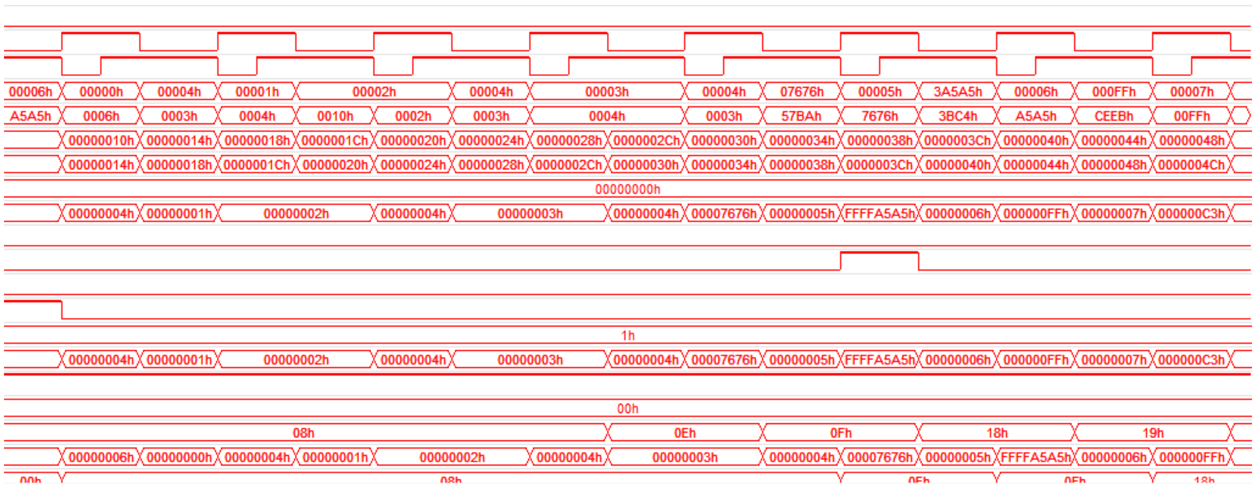

Figure 7: Writing test data into SRAM for part A (B = 4)

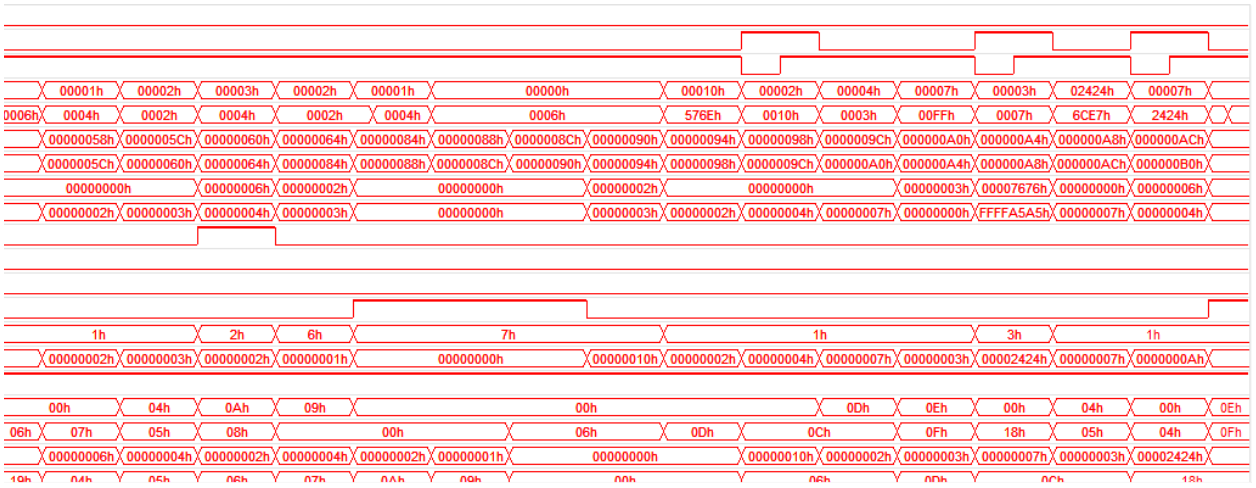

Figure 8: Computing and taking branch to else block (part A)

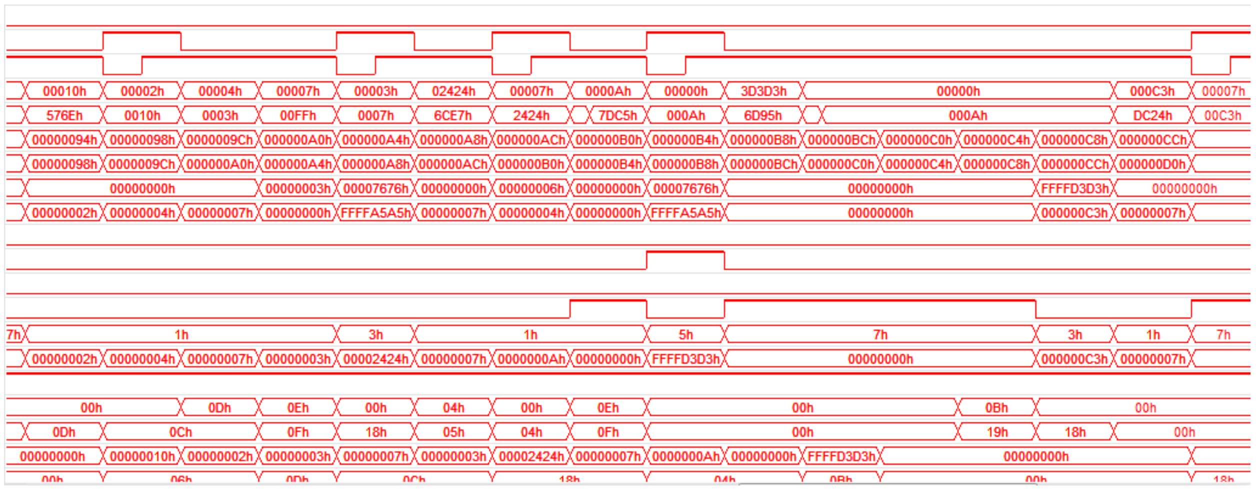

Figure 9: Writing new values to SRAM for part A

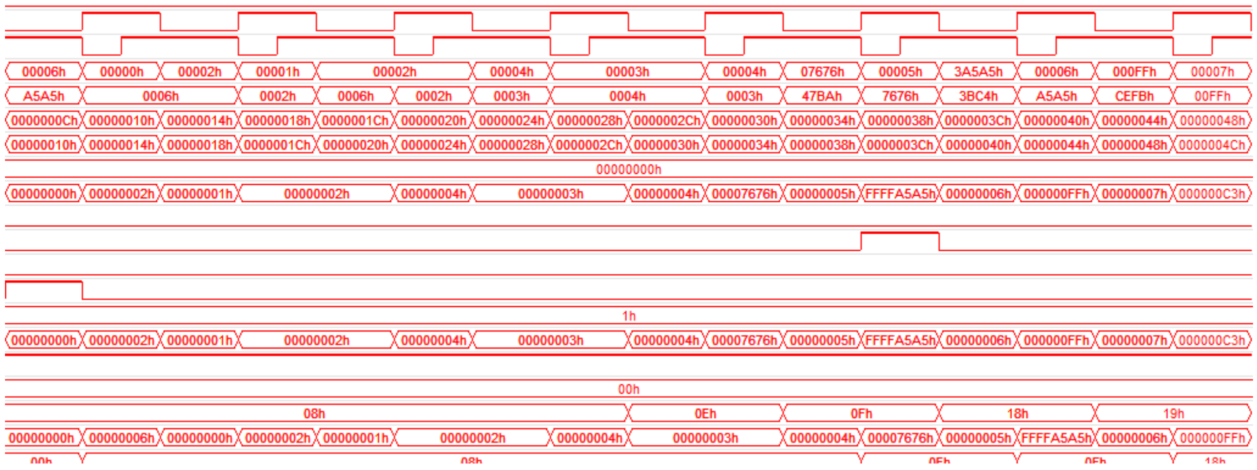

Figure 10: Writing test data to SRAM for part B (B = 4)

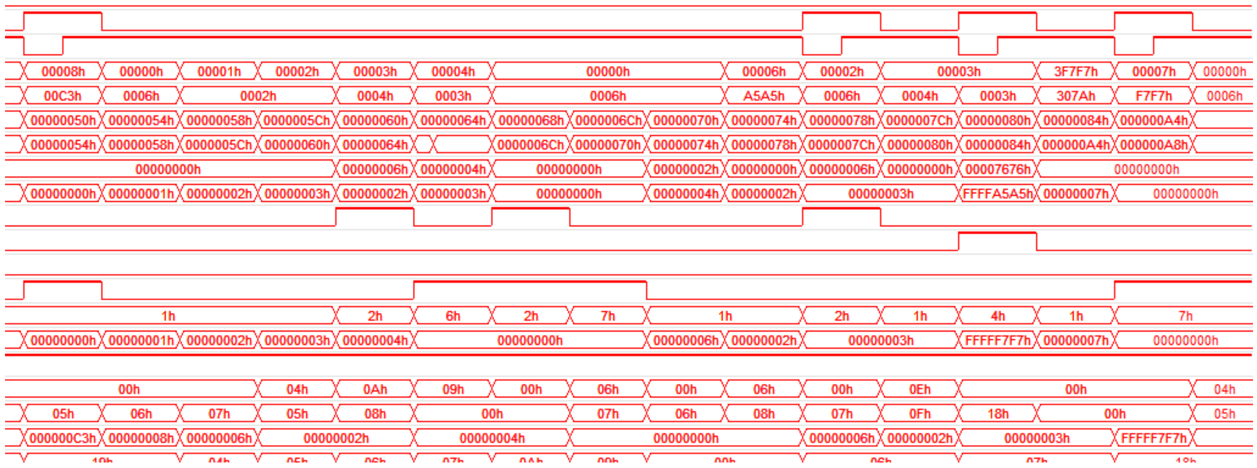

Figure 11: Computing and not taking branch, then jump over else block

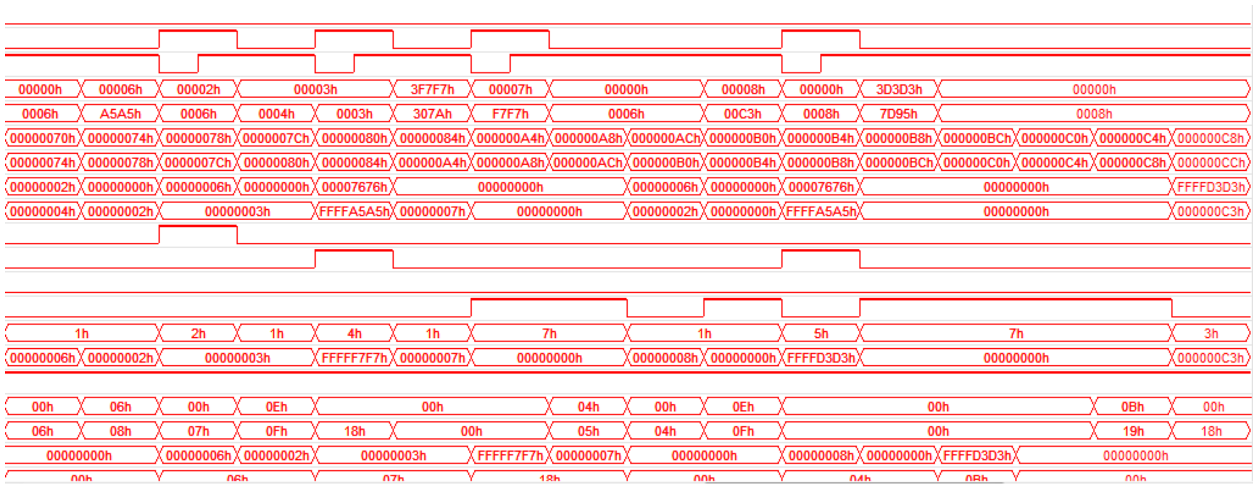

Figure 12: Writing new values to SRAM (part B)

ANALYSIS OF ERRORS

There were no errors which affected the final outcome of the project. The clock was slowed down slightly to accommodate the large signal tap test system that had to be synthesized into the design.

TROUBLESHOOTING ANALYSIS

Single Cycle CPU - For the single cycle CPU there were difficulties getting the datapath and control correct for certain instructions. For example, the shift instruction requires that the second operand to the ALU be from the shamt field in the R type instruction. This required additional muxes and control signals. There were also the immediate type instructions like load word and store word. Load word has the Rt field specify its destination. This required the use of another control signal. There were many cases where we forgot that the destination register is not always the Rd field and actually depends on which MIPS instruction is selected.

Pipeline CPU - For the pipeline CPU the most challenging part was getting the data forwarding working right. Difficulties in managing the control signals began to arise. To solve these kinds of errors, we carefully wrote out the contents of the pipeline for 4-5 instructions and ensured that the results in signal tap for the forwarding control signals were correct.

SUMMARY

The project successfully built a single cycle and pipeline CPU. The results of execution are as expected. The pipeline functioned as expected, data hazards were handled with forwarding and stalls. The control hazards were handled with a delay slot by computing the branch in the decode stage.

CONCLUSION

The project was completed and the test program executed without any problems. Our simple direct mapped cache design worked. The single cycle computer supported all instructions in the specification in a single clock cycle. Its control and data path worked very well. The pipeline CPU functioned in a similar way, but the different stages were executed in parallel allowing for a faster clock to potentially be used. Both projects fulfilled all the requirements. One place where improvement could be made would be the pipeline CPU. As the design continued, the group began to have difficulty managing the complexity of the design in Verilog. The pipeline CPU required far more wires and control signals than the single cycle. The impact of our design decisions such as the separate decoding was felt strongly in the later part of the project as it ultimately made the design more complicated. Future work should manage complexity early on, to ensure a clean and concise final HDL design.

APPENDIX A – CODE COMMON TO BOTH CPUS

Verilog for ALU

```
// 32 Bit ALU
module ALU (out, Z, V, C, N, A, B, control);
	input [31:0] A, B;
	input [2:0] control;
	output [31:0] out;
	output Z, V, C, N;

	wire [7:0] enab;
	wire [7:0] zero;
	wire [7:0] overflow;
	wire [7:0] cout;
	wire [7:0] neg;

	wire [255:0] result;
	wire sub;

	// assign subtraction for sub, slt and sll (sll doesn't use add or sub so it doesn't matter)
	assign sub = control[1];

	// 000 NOP
	assign result[31:0] = 32'd0;
	assign zero[0] = 1'b0;
	assign overflow[0] = 1'b0;
	assign cout[0] = 1'b0;
	assign neg[0] = 1'b0;

	// 001 ADD
	addsub32bit add0(sub, result[63:32], zero[1], overflow[1], cout[1], neg[1], A, B);

	// 010 SUB
	assign result[95:64] = result[63:32]; // result is output of adder if sub is true
	assign zero[2] = zero[1];
	assign overflow[2] = overflow[1];
	assign cout[2] = cout[1];
	assign neg[2] = neg[1];

	// 011 AND
	BitwiseAnd and0(result[127:96], zero[3], overflow[3], cout[3], neg[3], A, B);

	// 100 OR
	BitwiseOr or0(result[159:128], zero[4], overflow[4], cout[4], neg[4], A, B);

	// 101 XOR
	BitwiseXor xor0(result[191:160], zero[5], overflow[5], cout[5], neg[5], A, B);

	// 110 SLT
	SLT slt0(result[223:192], zero[6], overflow[6], cout[6], neg[6], A, B, result[95:64]);

	// 111 SLL
	SLL sll0(result[255:224], zero[7], overflow[7], cout[7], neg[7], A, B);

	Mux8_32 rmux(out, result, control);

	Mux8 zmux(Z, zero, control);
	Mux8 vmux(V, overflow, control);
	Mux8 cmux(C, cout, control);
	Mux8 nmux(N, neg, control);

endmodule

module ALUControl (aluControl, aluOp, funct);
	input [5:0] funct;
	input [1:0] aluOp;
	output [2:0] aluControl;

	wire [2:0] funcCon;

	wire ADD, SUB, AND, OR, XOR, SLT, SLL;
	wire opAdd, opSub, opFunc;

	assign ADD = ~|(funct ^ 6'b100000);
	assign SUB = ~|(funct ^ 6'b100010);
	assign AND = ~|(funct ^ 6'b100100);
	assign OR = ~|(funct ^ 6'b100101);
	assign XOR = ~|(funct ^ 6'b100110);
	assign SLT = ~|(funct ^ 6'b101010);
	assign SLL = ~|(funct ^ 6'b000000);

	assign funcCon[2] = OR | XOR | SLT | SLL;
	assign funcCon[1] = SUB | AND | SLT | SLL;
	assign funcCon[0] = ADD | AND | XOR | SLL;

	assign opAdd = ~|(aluOp ^ 2'b00);
	assign opSub = ~|(aluOp ^ 2'b01);
	assign opFunc = ~|(aluOp ^ 2'b10);

	// if opfunc and none of these is asserted, it defaults to NOP
	assign aluControl[2] = opFunc & funcCon[2];
	assign aluControl[1] = (opFunc & funcCon[1]) | opSub;
	assign aluControl[0] = (opFunc & funcCon[0]) | opAdd;

endmodule

module oneBitAdder(s, p, g, c, a, b);
	input c,a,b;
	output s,p,g;

	assign p = a ^ b;
	assign g = a & b;

	assign s = a ^ b ^ c;
endmodule


module fourBitLookAheadAdder(s, pOut, gOut, cIn, a, b);
	input cIn;
	input [3:0] a, b;

	output pOut, gOut;
	output [3:0] s;
	// output cOut;

	wire [3:1] c;
	wire [3:0] p, g;

	// create four one-bit-adders
	oneBitAdder adder0(s[0],p[0],g[0],cIn,a[0],b[0]),
				adder1(s[1],p[1],g[1],c[1],a[1],b[1]),
				adder2(s[2],p[2],g[2],c[2],a[2],b[2]),
				adder3(s[3],p[3],g[3],c[3],a[3],b[3]);

	// determine carries into one-bit-adders 3, 2, and 1. (adder 0 gets cIn carry)
	assign c[1] = g[0] | (p[0] & cIn);
	assign c[2] = g[1] | (g[0] & p[1]) | (p[0] & p[1] & cIn);
	assign c[3] = g[2] | (g[1] & p[2]) | (g[0] & p[2] & p[1]) | (p[0] & p[1] & p[2] & cIn);

	// determine carry out of the system, propogate out and generate out
	// assign cOut = g[3] | (g[2] & p[3]) | (g[1] & p[3] & p[2]) | (g[0] & p[3] & p[2] & p[1]) | (p[3]
& p[2] & p[1] & p[0] & cIn);
	assign pOut = p[3] & p[2] & p[1] & p[0];
	assign gOut = g[3] | (g[2] & p[3]) | (g[1] & p[3] & p[2]) | (g[0] & p[3] & p[2] & p[1]);
endmodule

module sixteenBitLookAheadAdder(s, pOut, gOut, cIn, a, b);
	input cIn;
	input [15:0] a, b;

	output [15:0] s;
	output pOut, gOut;
	//output cOut;

	wire [3:1] c;
	wire [3:0] p, g;

	// create four four-bit-adders
	fourBitLookAheadAdder adder0(s[3:0], p[0], g[0], cIn, a[3:0], b[3:0]),
						adder1(s[7:4], p[1], g[1], c[1], a[7:4], b[7:4]),
						adder2(s[11:8], p[2], g[2], c[2], a[11:8], b[11:8]),
						adder3(s[15:12], p[3], g[3], c[3], a[15:12],
b[15:12]);

	// determine carries into four-bit-adders 3, 2, and 1 (four-bit-adder 0 gets cIn carry)
	assign c[1] = g[0] | (p[0] & cIn);
	assign c[2] = g[1] | (g[0] & p[1]) | (p[0] & p[1] & cIn);
	assign c[3] = g[2] | (g[1] & p[2]) | (g[0] & p[2] & p[1]) | (p[0] & p[1] & p[2] & cIn);

	// determine carry out of the system, propogate out and generate out
	//assign cOut = g[3] | (g[2] & p[3]) | (g[1] & p[3] & p[2]) | (g[0] & p[3] & p[2] & p[1]) | (p[3] &
p[2] & p[1] & p[0] & cIn);
	assign pOut = p[3] & p[2] & p[1] & p[0];
	assign gOut = g[3] | (g[2] & p[3]) | (g[1] & p[3] & p[2]) | (g[0] & p[3] & p[2] & p[1]);
endmodule


module thirtyTwoBitLookAheadAdder(s, cOut, cIn, a, b);
	input cIn;
	input [31:0] a, b;

	output cOut;
	output [31:0] s;
	wire [1:0] p, g;
	wire c;

	sixteenBitLookAheadAdder adder0(s[15:0], p[0], g[0], cIn, a[15:0], b[15:0]),
							 adder1(s[31:16], p[1], g[1], c, a[31:16],
b[31:16]);

	assign c = g[0] | (p[0] & cIn);
	assign cOut = g[1] | (g[0] & p[1]) | (p[0] & p[1] & cIn);
endmodule


module addsub32bit(sub, result, zero, overflow, cout, neg, a,b);
	input [31:0] a,b;
	input sub;

	output [31:0] result;
	output zero, overflow, cout, neg;

	wire [31:0] bCorrected;

	assign bCorrected = ({32{sub}} & ~b) | ({32{~sub}} & b);

	thirtyTwoBitLookAheadAdder adder(result, cout, sub, a, bCorrected);

	assign zero = ~ (| result);
	assign overflow = ~sub & (~(a[31] ^ b[31]) & (a[31] ^ result[31])) | sub & (~(a[31] ^
~b[31]) & (a[31] ^ result[31]));
	assign neg = result[31];
endmodule

// SLL function A << B
module SLL(result, Z, V, C, N, A, B);
	input [31:0] A, B;
	output Z, V, C, N;
	output [31:0] result;

	wire [31:0] shiftBy2;

	ShiftLeftBy2If sl2_0(A, shiftBy2, B[1]);
	ShiftLeftBy1If sl1_0(shiftBy2, result, B[0]);

	assign Z = ~(| result);
	assign V = 0; //
	assign C = (B[0] & A[31]) | (B[1] & (A[31] | A[30])) | (B[0] & B[1] & (A[29] | A[30] | A[29]));
	assign N = result[31];

endmodule

module ShiftLeftBy2If(in, out, sel);
	input [31:0] in;
	input sel;
	output [31:0] out;
	assign out = sel ? {in[29:0], 2'b00} : in;
endmodule

module ShiftLeftBy1If(in, out, sel);
	input [31:0] in;
	input sel;
	output [31:0] out;
	assign out = sel ? {in[30:0], 1'b0} : in;
endmodule

// SLT function return (A less than B)

module SLT(result, Z, V, C, N, A, B, diff);
	input [31:0] A, B, diff;
	output Z,V,C,N;
	output[31:0] result;

	wire Apos;
	wire Bpos;

	assign V = 0;
	assign C = 0;
	assign N = 0;
	assign Z = ~result[0];

	assign Apos = ~A[31];
	assign Bpos = ~B[31];

	assign result[31:1] = 0;

	assign result[0] = (~(Apos ^ Bpos) & diff[31]) | (~Apos & Bpos);
	assign result[31:1] = 31'd0;
	// if A is positive and B is positive, return (A - B) < 0
	// if A is negative and B is negative, return (A - B) < 0
	// if A is negative and B is positive, return 1
```

```
        // if A is positive and B is negative, return 0

endmodule

module BitwiseAnd(result, zero, overflow, cout, neg, a, b);
        input [31:0] a;
        input [31:0] b;
        output [31:0] result;
        output zero, overflow, cout, neg;

        assign result = a & b;

        assign neg = result[31];
        assign zero = ~(| result);
        assign overflow = 0;
        assign cout = 0;
endmodule

module BitwiseOr(result, zero, overflow, cout, neg, a, b);
        input [31:0] a;
        input [31:0] b;
        output [31:0] result;
        output zero, overflow, cout, neg;

        assign result = a | b;

        assign neg = result[31];
        assign zero = ~(| result);
        assign overflow = 0;
        assign cout = 0;
endmodule

module BitwiseXor(result, zero, overflow, cout, neg, a, b);
        input [31:0] a;
        input [31:0] b;
        output [31:0] result;
        output zero, overflow, cout, neg;

        assign result = a ^ b;

        assign neg = result[31];
        assign zero = ~(| result);
        assign overflow = 0;
        assign cout = 0;
```

```
endmodule
```

Verilog for SRAM Driver

```
module DataMemory (dataOut, dataIn, address, read, write, clock, reset, CS, OE, RW,
addressOut, rambus);
	output [15:0] dataOut;
	input [15:0] dataIn;
	input [31:0] address;
	input read, write, clock, reset;

	// external chip pins
	output CS, OE;
	output RW;
	output [17:0] addressOut;
	inout [15:0] rambus;

	assign CS = 0;
	assign OE = write;
	assign rambus = write ? dataIn : 16'hzzzz;
	assign addressOut = address[17:0];

	assign RW = ~(clock&reset) | clock&reset&~write;

	assign dataOut = rambus;

endmodule
```

Verilog for Register File

```
module RegisterFile(readOut1, readOut2, readSel1, readSel2, writeSel, writeData, writeEnable,
clock, reset);

	output [31:0] readOut1, readOut2;

	input [31:0] writeData;
	input [4:0] readSel1, readSel2, writeSel;
	input writeEnable, clock, reset;

	wire [31:0]
readOutput1,readOutput2,readOutput3,readOutput4,readOutput5,readOutput6,readOutput7,rea
dOutput8,
	readOutput9,readOutput10,readOutput11,readOutput12,readOutput13,readOutput14,rea
dOutput15,readOutput16,readOutput17,readOutput18,readOutput19,readOutput20,
	readOutput21,readOutput22,readOutput23,readOutput24,readOutput25,readOutput26,re
adOutput27,readOutput28,readOutput29,readOutput30,readOutput31;

	wire [31:0] decodedWriteSel, w;

	Dec32 dec32_0(decodedWriteSel, writeSel);
	and and_0[31:0](w, decodedWriteSel, writeEnable);

	Register
			reg1(readOutput1, writeData,  w[1], clock, reset),
			reg2(readOutput2, writeData,  w[2], clock, reset),
			reg3(readOutput3, writeData,  w[3], clock, reset),
			reg4(readOutput4, writeData,  w[4], clock, reset),
			reg5(readOutput5, writeData,  w[5], clock, reset),
			reg6(readOutput6, writeData,  w[6], clock, reset),
			reg7(readOutput7, writeData,  w[7], clock, reset),
			reg8(readOutput8, writeData,  w[8], clock, reset),
			reg9(readOutput9, writeData,  w[9], clock, reset),
			reg10(readOutput10, writeData, w[10], clock, reset),
			reg11(readOutput11, writeData, w[11], clock, reset),
			reg12(readOutput12, writeData, w[12], clock, reset),
			reg13(readOutput13, writeData, w[13], clock, reset),
			reg14(readOutput14, writeData, w[14], clock, reset),
			reg15(readOutput15, writeData, w[15], clock, reset),
			reg16(readOutput16, writeData, w[16], clock, reset),
			reg17(readOutput17, writeData, w[17], clock, reset),
			reg18(readOutput18, writeData, w[18], clock, reset),
			reg19(readOutput19, writeData, w[19], clock, reset),
			reg20(readOutput20, writeData, w[20], clock, reset),
			reg21(readOutput21, writeData, w[21], clock, reset),
			reg22(readOutput22, writeData, w[22], clock, reset),
			reg23(readOutput23, writeData, w[23], clock, reset),
			reg24(readOutput24, writeData, w[24], clock, reset),
			reg25(readOutput25, writeData, w[25], clock, reset),
			reg26(readOutput26, writeData, w[26], clock, reset),
			reg27(readOutput27, writeData, w[27], clock, reset),
			reg28(readOutput28, writeData, w[28], clock, reset),
			reg29(readOutput29, writeData, w[29], clock, reset),
			reg30(readOutput30, writeData, w[30], clock, reset),
			reg31(readOutput31, writeData, w[31], clock, reset);

	Mux32_32
mux0(readOut1,{readOutput31,readOutput30,readOutput29,readOutput28,readOutput27,readO
utput26,readOutput25,readOutput24,readOutput23,
	readOutput22,readOutput21,readOutput20,readOutput19,readOutput18,readOutput17,readOutput16,readOutput15,readOutput14,readOutput13,readOutput12,
	readOutput11,readOutput10,readOutput9,readOutput8,readOutput7,readOutput6,readOutput5,readOutput4,readOutput3,readOutput2,readOutput1, 32'h00000000}, readSel1),

	mux1(readOut2,{readOutput31,readOutput30,readOutput29,readOutput28,readOutput27,readOutput26,readOutput25,readOutput24,readOutput23,
	readOutput22,readOutput21,readOutput20,readOutput19,readOutput18,readOutput17,readOutput16,readOutput15,readOutput14,readOutput13,readOutput12,
	readOutput11,readOutput10,readOutput9,readOutput8,readOutput7,readOutput6,readOutput5,readOutput4,readOutput3,readOutput2,readOutput1,32'h00000000}, readSel2);

endmodule
```

Verilog for Instruction Cache

```
module InstructionCache (dataOut, dataIn, address, hit, clock, reset);
	input [31:0] address;
	input [127:0] dataIn;
	input clock, reset;
	output [31:0] dataOut;
	output hit;

	wire [3:0] write;
	wire [31:0] dataOut3, dataOut2, dataOut1, dataOut0;

	CacheBlock block3 (dataOut3, dataIn, address[3:2], write[3], clock, reset);
	CacheBlock block2 (dataOut2, dataIn, address[3:2], write[2], clock, reset);
	CacheBlock block1 (dataOut1, dataIn, address[3:2], write[1], clock, reset);
	CacheBlock block0 (dataOut0, dataIn, address[3:2], write[0], clock, reset);

	tagTable tag0 (hit, address, clock, reset);

	Dec4 dec0 (write, address[5:4], ~hit);

	Mux4_32 mux0 (dataOut, {dataOut3, dataOut2, dataOut1, dataOut0}, address[5:4]);

endmodule

module CacheBlock (dataOut, dataIn, offset, write, clock, reset);
	input [127:0] dataIn;
	input [1:0] offset;
	input write, clock, reset;
	output [31:0] dataOut;
```

```
        wire [31:0] readOutput3, readOutput2, readOutput1, readOutput0;

        Register reg3 (readOutput3, dataIn[127:96], write, clock, reset),
                        reg2 (readOutput2, dataIn[95:64], write, clock, reset),
                        reg1 (readOutput1, dataIn[63:32], write, clock, reset),
                        reg0 (readOutput0, dataIn[31:0], write, clock, reset);

        Mux4_32 mux0 (dataOut, {readOutput3, readOutput2, readOutput1, readOutput0},
offset);

endmodule

module tagTable (hit, address, clock, reset);
        input [31:0] address;
        input clock, reset;
        output hit;

        wire [25:0] tag;
        wire valid;
        wire [31:0] data;
        wire [3:0] write;
        wire [31:0] out3, out2, out1, out0;

        Register r3 (out3, {6'b1, address[31:6]}, write[3], clock, reset);
        Register r2 (out2, {6'b1, address[31:6]}, write[2], clock, reset);
        Register r1 (out1, {6'b1, address[31:6]}, write[1], clock, reset);
        Register r0 (out0, {6'b1, address[31:6]}, write[0], clock, reset);

        Mux4_32 mux0 (data, {out3, out2, out1, out0}, address[5:4]);

        // if hit, write to the reg speced by index, setting valid = 1 and updating tag
        Dec4 dec0 (write, address[5:4], ~hit);

        assign tag = data[25:0];
        assign valid = data[26];

        assign hit = ~|(tag ^ address[31:6]) & valid;

endmodule
```

Verilog for Instruction Memory

```
module InstructionMemory(instruction, address, clock, reset);
```

```
	input [31:0] address;
	input clock, reset;
	output [31:0] instruction;

	wire [127:0] instructionBlock;
	wire hit;
	wire [31:0] cachedVal;
	wire [31:0] instr;

	InstructionMem mem0 (instructionBlock, address);

	InstructionCache cache0 (cachedVal, instructionBlock, address, hit, clock, reset);

	Mux4_32 (instr, instructionBlock, address[3:2]);

	assign instruction = hit ? cachedVal : instr;

endmodule

module InstructionMem(instruction, address);
// uses code from Hauck's instruction memory loader: $readmemh("…..

input [31:0] address;
output [127:0] instruction;
reg [127:0]instrmem[31:0];
reg [127:0] temp;

// reverse word order
assign instruction = {temp[31:0], temp[63:32], temp[95:64], temp[127:96]};

always @(address)
begin
	temp=instrmem[address/16];
end

initial
begin
$readmemh("partC.dat", instrmem);
end

endmodule
```

Verilog for Misc. Modules

```
module Register (readOutput, writeInput, write, clock, reset);
```

```
	input write, clock, reset;
	input [31:0] writeInput;
	output [31:0] readOutput;

	wire [31:0] newData;
	wire [31:0] oldData;
	wire [31:0] d;
	wire notwrite;

	and and0 [31:0] (newData, write, writeInput);
	not not0(notwrite, write);
	and and1 [31:0] (oldData, notwrite, readOutput);
	or or0 [31:0] (d, oldData, newData);
	Dflipflop Dflipflop0 [31:0] (readOutput, , d, clock, reset);
endmodule

module Mux32_32 (out, in, select);
	input [1023:0] in;
	input [4:0] select;
	output [31:0] out;
	wire [31:0] d1, d0;

	Mux16_32 m1 (d1, in[1023:512], select[3:0]);
	Mux16_32 m0 (d0, in[511:0], select[3:0]);

	Mux2_32 mf (out, {d1, d0}, select[4]);
endmodule

module Mux2_32 (out, in, select);
	input [63:0] in;
	input select;
	output [31:0] out;
	wire [31:0] d0;
	wire [31:0] d1;
	wire notSelect;

	not not0 (notSelect, select);
	and and0 [31:0] (d0, notSelect, in[31:0]);
	and and1 [31:0] (d1, select, in[63:32]);

	or or1 [31:0] (out, d0, d1);
endmodule

module Mux4_32 (out, in, select);
```

```
        input [127:0] in;
        input [1:0] select;
        output [31:0] out;
        wire [31:0] d0;
        wire [31:0] d1;

        Mux2_32 m1(d1, in [127:64], select[0]);
        Mux2_32 m0(d0, in [63:0], select[0]);

        Mux2_32 mf(out, {d1, d0}, select[1]);
endmodule

module Mux8_32 (out, in, select);
        input [255:0] in;
        input [2:0] select;
        output [31:0] out;
        wire [31:0] d1, d0;

        Mux4_32 m1 (d1, in[255:128], select[1:0]);
        Mux4_32 m0 (d0, in[127:0], select[1:0]);

        Mux2_32 mf (out, {d1, d0}, select[2]);
endmodule

module Mux16_32 (out, in, select);
        input [511:0] in;
        input [3:0] select;
        output [31:0] out;
        wire [31:0] d0, d1, d2, d3;

        Mux4_32 m3(d3, in[511:384], select[1:0]);
        Mux4_32 m2(d2, in[383:256], select[1:0]);
        Mux4_32 m1(d1, in[255:128], select[1:0]);
        Mux4_32 m0(d0, in[127:0], select[1:0]);

        Mux4_32 mf(out, {d3, d2, d1, d0}, select[3:2]);
endmodule

module Mux8 (out, in, select);
        input [7:0] in;
        input [2:0] select;
        output out;

        wire d1, d0;
        Mux4 m1 (d1, in[7:4], select[1:0]);
        Mux4 m0 (d0, in[3:0], select[1:0]);

        Mux2 mf (out, {d1, d0}, select[2]);
endmodule

module Mux4 (out, in, select);
        input [3:0] in;
        input [1:0] select;
        output out;

        wire d1, d0;
        Mux2 m1(d1, in[3:2], select[0]);
        Mux2 m0(d0, in[1:0], select[0]);

        Mux2 mf(out, {d1, d0}, select[1]);

endmodule

module Mux2 (out, in, select);
        input [1:0] in;
        input select;
        output out;

        assign out = select ? in[1] : in[0];
endmodule

module Dec32 (data, select);
        input [4:0] select;
        output [31:0] data;
        wire notSelect4;

        not not0 (notSelect4, select[4]);
        Dec16 Dec1 (data[31:16], select[3:0], select[4]);
        Dec16 Dec0 (data[15:0], select[3:0], notSelect4);

endmodule

module Dec16 (data, select, enab);
        input [3:0] select;
        output [15:0] data;
        input enab;
        wire [3:0] enabs;
```

```
	Dec4 Dec4(enabs, select[3:2], enab);

	Dec4 Dec3(data[15:12], select[1:0], enabs[3]);
	Dec4 Dec2(data[11:8], select[1:0], enabs[2]);
	Dec4 Dec1(data[7:4], select[1:0], enabs[1]);
	Dec4 Dec0(data[3:0], select[1:0], enabs[0]);

endmodule

module Dec4 (data, select, enab);
	input [1:0] select;
	input enab;
	output [3:0] data;
	wire notSelect1, notSelect0;

	not not1(notSelect1, select[1]);
	not not0(notSelect0, select[0]);

	and and0(data[3], select[1], select[0], enab);
	and and1(data[2], select[1], notSelect0, enab);
	and and2(data[1], notSelect1, select[0], enab);
	and and3(data[0], notSelect1, notSelect0, enab);

endmodule

module Dflipflop(q, qBar, D, clk, rst);
	input D, clk, rst;
	output q, qBar;
	reg q;
	not n1 (qBar, q);
	always@ (negedge rst or posedge clk)
	begin
	if(!rst)
	q = 0;
	else
	q = D;
	end
endmodule

module ClockDivider(divided_clocks, clock, reset);
	input clock, reset;
	output reg [31:0] divided_clocks;

	always @ (posedge clock or negedge reset) begin
```

```
                if (~reset)
                        divided_clocks = 2;
                else
                        divided_clocks = divided_clocks + 1;
        end
endmodule
```

Appendix B – SINGLE CYCLE CPU

Verilog for Top Level

```
module SingleCycleCPU(CS, OE, RW, addressOut, rambus, clock, reset);

        parameter sysClock = 1;

        input clock, reset;
        output CS, OE, RW;
        output [17:0] addressOut;
        inout [15:0] rambus;

        // register wires
        wire [31:0] readOut1, readOut2, writeData;
        wire [4:0] readSel1, readSel2, writeSel;
        wire writeEnable;

        // instruction decoder
        wire memRead, memWrite, branch, jump, jumpR, aluSel, memToReg, regDest, shift;
        wire [1:0] aluOp;

        // alu
        wire [31:0] result, A, B;
        wire Z, V, C, N;
        wire [2:0] aluControl;

        // control
        wire [31:0] instruction, currentAddress, nextAddress;

        // data memory
        wire [15:0] dataOut;

        // misc wiring
        wire [31:0] tempBsel;
        wire [31:0] divided_clocks;
```

```
	RegisterFile regfile0 (readOut1, readOut2, instruction[25:21], instruction[20:16], writeSel,
writeData, writeEnable, divided_clocks[sysClock], reset);
	DataMemory datamem0 (dataOut, readOut2, result, memRead, memWrite,
divided_clocks[sysClock], reset, CS, OE, RW, addressOut, rambus);
	ALU alu0 (result, Z, V, C, N, A, B, aluControl);

	Register PC (currentAddress, nextAddress, 1'b1, divided_clocks[sysClock], reset);

	Mux2_32 bsel (tempBsel, {{{16{instruction[15]}}, instruction[15:0]}, readOut2}, aluSel);
	Mux2_32 datasel (writeData, {{16{dataOut[15]}}, dataOut[15:0], result}, memToReg);
	Mux2_32 regsel (writeSel, {{{27{1'b0}}, instruction[15:11]}, {{27{1'b0}},
instruction[20:16]}}, regDest);
	Mux2_32 shamtsel (B, {{27'b0, instruction[10:6]}, tempBsel}, shift);
	Mux2_32 firstopsel (A, {readOut2, readOut1}, shift);

	InstructionMemory instrmem0 (instruction, currentAddress, divided_clocks[sysClock],
reset);

	InstructionDecoder decoder0 (writeEnable, memRead, memWrite, branch, jump, jumpR,
aluSel, memToReg, regDest, shift, aluOp, instruction[31:26], instruction[5:0]);

	ALUControl alucontrol0 (aluControl, aluOp, instruction[5:0]);

	Control control0 (nextAddress, currentAddress, branch, Z, jump, jumpR,
instruction[15:0], instruction[25:0], readOut1);

	ClockDivider divider0 (divided_clocks, clock, reset);

endmodule
```

Verilog for Control Path

```
module Control(nextAddress, currentAddress, branch, Z, jump, jumpR, immediate,
jumpAddress, readOut1);
	output [31:0] nextAddress;

	input [31:0] currentAddress;
	input [25:0] jumpAddress;
	input [15:0] immediate;
	input [31:0] readOut1;
	input branch, jump, jumpR, Z;
	wire [31:0] pcPlus4, branchAddress, address0, address1,
immediateSignExtendedAndShifted;

	assign immediateSignExtendedAndShifted[31:18] = {14{immediate[15]}};
	assign immediateSignExtendedAndShifted[17:0] = {immediate, 2'b00};

	addsub32bit addFour(0, pcPlus4,,,,,currentAddress, 32'h00000004),
				branchAdd(0, branchAddress,,,,,pcPlus4,
immediateSignExtendedAndShifted);

	Mux2_32 branchMux(address0, {branchAddress, pcPlus4}, branch & ~Z),
			jumpregMux(address1, {readOut1, address0}, jumpR),
			jumpMux(nextAddress,{{pcPlus4[31:28], jumpAddress, 2'b00}, address1},
jump);

endmodule

module InstructionDecoder (writeEnable, memRead, memWrite, branch, jump, jumpR, aluSel,
memToReg, regDest, shift, aluOp, opcode, funct);
	input [5:0] opcode, funct;
	output writeEnable, memRead, memWrite, branch, jump, jumpR, aluSel, memToReg,
regDest, shift;
	output [1:0] aluOp;


	wire Rformat;

	wire LW, SW, BNE, SLL, J, JR, ADDI;

	assign Rformat = ~|opcode;
	assign LW = ~|(opcode ^ 6'b100011);
	assign SW = ~|(opcode ^ 6'b101011);
	assign BNE = ~|(opcode ^ 6'b000101);
	assign ADDI = ~|(opcode ^ 6'b001000);
	assign SLL = ~|funct;

	assign J = ~|(opcode ^ 6'b000010);
	assign JR = ~|(funct ^ 6'b001000);

	// enables writing to the register file
	assign writeEnable = Rformat | LW | ADDI;

	// selects read operation for data memory
	assign memRead = LW;
	// selects write operation for data memory
	assign memWrite = SW;

	// selects branch operation for control block
	assign branch = BNE;

	// selects the jump operation for the control block (note to self, likely that we will actually
need two control sigs when JR is implemented).
	assign jump = J;

	assign jumpR = Rformat & JR;

	// selects the 16 bit immediate value as the input to the alu
	assign aluSel = LW | SW | ADDI;

	// selects the write data as the  16 bit value read from data memory rather than alu result
	assign memToReg = LW;

	// selects the destination register address as the rd field [15:11] otherwise rt field [20:16]
	assign regDest = Rformat;

	// assert shift control output to change the second input to the ALU to come from the shift
amount.
	// also changes the first amount to be from the rt field (not rs field as is usual)
	assign shift = Rformat & SLL;

	assign aluOp[1] = Rformat;
	assign aluOp[0] = BNE;

endmodule
```

Appendix C – CODE FOR PIPELINED CPU

Verilog for Top Level Pipeline CPU

```
module PipelineCPU(CS, OE, RW, addressOut, rambus, clock, reset);

	parameter sysClock = 1;

	input clock, reset;
	output CS, OE, RW;
	output [17:0] addressOut;
	inout [15:0] rambus;

	// register wires
	wire [31:0] readOut1, readOut2, writeData, writeIntoSRAM;
	wire [4:0] readSel1, readSel2, writeSel;
	wire writeEnable;

	// instruction decoder
	wire memRead, memWrite, branch, jump, jumpR, aluSel, memToReg, regDest, shift;
	wire [1:0] aluOp;

	// alu
	wire [31:0] result, A, B;
	wire Z, V, C, N;
	wire [2:0] aluControl;

	// control
	wire [31:0] instruction, currentAddress, tempNextAddress, nextAddress;
	wire [31:0] pcPlus4, branchAddress, address0, address1,
immediateSignExtendedAndShifted;

	// data memory
	wire [15:0] dataOut;

	// misc wiring
	wire [31:0] tempBsel, tempBsel2, tempAsel;
	wire [31:0] divided_clocks;
	wire [1:0] forwardA, forwardB;
	wire PCSrc;

	wire [31:0] pcPlus4ID, pcPlus4EX,
	tempInstruction, instructionID, instructionEX, instructionMEM, instructionWB,
	readOut1EX, readOut2EX, readOut2MEM,
	resultMEM, resultWB,
	dataOutMEM, dataOutWB, read1ForBNE, read2ForBNE, readOut1Temp,
readOut2Temp, tempAsel2,
	readOut1FromWB, readOut2FromWB, readOut1FromMEM, readOut2FromMEM,
writeIntoSRAMEX, writeIntoSRAMMEM;
	wire ZMEM, regDestMEM,regDestEX, writeEnableMEM, forwardC, stall, memReadEX,
forwardD,
	forwardE, forwardF, forwardG, forwardH, forwardI, memWriteEX;
	wire [4:0] RdMEM, RdWB;

	wire [31:0] instructionIDtemp, tempTempInstruction;
	// IF
	addsub32bit addFour(0, pcPlus4,,,,,currentAddress, 32'h00000004);

	InstructionMemory instrmem0 (tempInstruction, currentAddress,
divided_clocks[sysClock], reset);

	Mux2_32 branchMux(address0, {branchAddress, pcPlus4}, PCSrc),
		jumpregMux(address1, {readOut1, address0}, jumpR),
		jumpMux(tempNextAddress,{{pcPlus4ID[31:28], instructionID[25:0], 2'b00},
address1}, jump);

	Mux2_32 stallMux(nextAddress, {currentAddress, tempNextAddress}, stall);

	Register PC (currentAddress, nextAddress, 1'b1, divided_clocks[sysClock], reset);


	Mux2_32 stallMux2(tempTempInstruction, {instructionID, tempInstruction}, stall);

	IFID ifid0 (pcPlus4ID, instructionID, pcPlus4, tempTempInstruction,
divided_clocks[sysClock], reset);



	// ID

	Mux2_32 fwordf (readOut1FromWB, {writeData, readOut1Temp}, forwardF);
	Mux2_32 fwordG (readOut2FromWB, {writeData, readOut2Temp}, forwardG);

	Mux2_32 read3mux (readOut1FromMEM, {dataOut ,readOut1FromWB}, forwardH);
	Mux2_32 read4mux (readOut2FromMEM, {dataOut ,readOut2FromWB}, forwardI);

	Mux2_32 read1mux (readOut1, {result ,readOut1FromMEM}, forwardD);
	Mux2_32 read2mux (readOut2, {result ,readOut2FromMEM}, forwardE);

	Mux2_32 stallMux33(instructionIDtemp, {32'b0, instructionID}, stall | jump | jumpR |
PCSrc);

	HazardDetectionUnit HazardDetectionUnit0(stall, PCSrc, branchAddress, jump, jumpR,
memReadEX, instructionID, instructionEX, readOut1, readOut2, pcPlus4ID);

	RegisterFile regfile0 (readOut1Temp, readOut2Temp, instructionID[25:21],
instructionID[20:16], writeSel, writeData, writeEnable, divided_clocks[sysClock], reset);
```

```
	IDEX idex0 (pcPlus4EX, instructionEX, readOut1EX, readOut2EX, pcPlus4ID,
instructionIDtemp, readOut1, readOut2, divided_clocks[sysClock], reset);

	// EX
	EX_decode decEx (aluSel, shift, aluOp, regDestEX, memReadEX, memWriteEX,
instructionEX);

	ALU alu0 (result, Z, V, C, N, A, B, aluControl);
	ALUControl alucontrol0 (aluControl, aluOp, instructionEX[5:0]);
	Mux2_32 bsel (tempBsel, {{{16{instructionEX[15]}}, instructionEX[15:0]}, readOut2EX},
aluSel);
	Mux4_32 bsel2 (tempBsel2, {32'b0, resultMEM, writeData, tempBsel}, forwardB);
	Mux2_32 shamtsel (B, {{27'b0, instructionEX[10:6]}, tempBsel2}, shift);

	Mux2_32 firstopsel (tempAsel, {readOut2EX, readOut1EX}, shift);
	Mux4_32 asel (tempAsel2, {32'b0, resultMEM, writeData, tempAsel}, forwardA);
	Mux2_32 finalopsel (A, {tempBsel2, tempAsel2}, shift);

	//addsub32bit branchAdd(0, branchAddress,,,,,pcPlus4EX,
immediateSignExtendedAndShifted);

	ForwardingUnit funit0 (forwardA, forwardB, forwardC, forwardD, forwardE, forwardF,
forwardG, forwardH, forwardI,
	instructionEX[25:21], instructionEX[20:16], regDestMEM, regDest, instructionID,
instructionEX, instructionMEM,
	instructionWB, writeEnableMEM, writeEnable, regDestEX, memWrite, memRead,
instructionMEM[20:16]);

	EXMEM exmem0 (resultMEM, writeIntoSRAMMEM, instructionMEM, ZMEM, result,
readOut2EX, instructionEX, Z, divided_clocks[sysClock], reset);

	// MEM

	Mux2_32 forwardForStore(writeIntoSRAM ,{writeData,writeIntoSRAMMEM}, forwardC);

	MEM_decode decMem (memRead, memWrite, , , , regDestMEM, writeEnableMEM,
instructionMEM);
	DataMemory datamem0 (dataOut, writeIntoSRAM, resultMEM, memRead, memWrite,
divided_clocks[sysClock], reset, CS, OE, RW, addressOut, rambus);

	MEMWB memwb0 (dataOutWB, resultWB, instructionWB, dataOut, resultMEM,
instructionMEM, divided_clocks[sysClock], reset);
```

```
	// WB
	WB_decode decWb (writeEnable, memToReg, regDest, instructionWB);
	Mux2_32 datasel (writeData, {{16{dataOutWB[15]}}, dataOutWB[15:0], resultWB}, 
memToReg);
	Mux2_32 regsel (writeSel, {{{27{1'b0}}, instructionWB[15:11]}, {{27{1'b0}}, 
instructionWB[20:16]}}, regDest);

	ClockDivider divider0 (divided_clocks, clock, reset);

endmodule
```

Verilog for Hazard Detection

```
module HazardDetectionUnit(stall, PCSrc, branchAddress, jump, jumpR, memReadEX, 
instructionID, instructionEX, readOut1, readOut2, pcPlus4);
	output stall, PCSrc, jump, jumpR;
	output [31:0] branchAddress;

	input [31:0] instructionID, instructionEX, readOut1, readOut2, pcPlus4;
	input memReadEX;

	wire Rformat, BNE, J, JR, Z, branch;
	wire [31:0] branchAddress, address0, address1, tempNextAddress, 
immediateSignExtendedAndShifted;

	assign stall = memReadEX & (~|(instructionEX[20:16] ^ instructionID[25:21]) | 
~|(instructionEX[20:16] ^ instructionID[20:16]));

	assign immediateSignExtendedAndShifted[31:18] = {14{instructionID[15]}};
	assign immediateSignExtendedAndShifted[17:0] = {instructionID[15:0], 2'b00};

	ALU alu1 ( branchAddress, , , , , pcPlus4, immediateSignExtendedAndShifted, 6'b1);
	ALU alu2 ( , Z, , , , readOut2, readOut1, 6'b10);

	MEM_decode decode0( , , branch, jump, jumpR, , , instructionID);

	assign PCSrc = ~Z & branch;

endmodule
```

Verilog for Data Forwarding

```
module ForwardingUnit(forwardA, forwardB, forwardC, forwardD, forwardE, forwardF, forwardG,
forwardH, forwardI,
Rs, Rt, regDestMEM, regDestWB, instructionID, instructionEX, instructionMEM, instructionWB,
writeEnableMEM,
writeEnableWB, regDestEX, memWrite, memRead, RtMEM);
	input [4:0] Rs, Rt, RtMEM;
	input [31:0] instructionMEM, instructionWB, instructionEX, instructionID;
	input regDestMEM, writeEnableMEM, regDestWB, writeEnableWB, regDestEX,
memWrite, memRead;
	output [1:0] forwardA, forwardB;
	output forwardC, forwardD, forwardE, forwardF, forwardG, forwardH, forwardI;

	wire [4:0] RdMEM, RdWB, RdEX;
	wire writeEnableEX;

	assign RdMEM = regDestMEM ? instructionMEM[15:11] : instructionMEM[20:16];
	assign RdWB = regDestWB ? instructionWB[15:11] : instructionWB[20:16];
	assign RdEX = regDestEX ? instructionEX[15:11] : instructionEX[20:16];
	MEM_decode decode1(, , , , , , writeEnableEX, instructionEX);

	// if shift left logical, the first operand to the alu is actually Rt, in which case we want to

	assign forwardA[1] = writeEnableMEM & |(RdMEM ^ 5'b0) & ~|(Rs ^ RdMEM);
	assign forwardB[1] = writeEnableMEM & |(RdMEM ^ 5'b0) & ~|(Rt ^ RdMEM) &
regDestEX;

	assign forwardA[0] = ~forwardA[1] & writeEnableWB & |(RdWB ^ 5'b0) & ~|(Rs ^ RdWB);
	assign forwardB[0] = ~forwardB[1]& writeEnableWB & |(RdWB ^ 5'b0) & ~|(Rt ^ RdWB)
& regDestEX;

	assign forwardC = writeEnableWB & memWrite & ~|(RdWB ^ RtMEM) ;

	assign forwardD = writeEnableEX & ~|(RdEX ^ instructionID[25:21]) & |(RdEX ^ 5'b0);
	assign forwardE = writeEnableEX & ~|(RdEX ^ instructionID[20:16]) & |(RdEX ^ 5'b0);

	assign forwardF = writeEnableWB & ~|(RdWB ^ instructionID[25:21]) & |(RdWB ^ 5'b0);
	assign forwardG = writeEnableWB & ~|(RdWB ^ instructionID[20:16]) & |(RdWB ^ 5'b0);

	assign forwardH = writeEnableMEM & memRead & ~|(RdMEM ^ instructionID[25:21]) &
|RdMEM;
	assign forwardI = writeEnableMEM & memRead & ~|(RdMEM ^ instructionID[20:16]) &
|RdMEM;
```

```
endmodule
```

Verilog for Pipeline Registers

```
module IFID(pcPlus4Out, instructionOut, pcPlus4In, instructionIn, clock, reset);
	input [31:0] pcPlus4In, instructionIn;
	input clock, reset;
	output reg [31:0] pcPlus4Out, instructionOut;

	always @ (posedge clock or negedge reset) begin
		if (~reset) begin
			pcPlus4Out = 0;
			instructionOut = 0;
		end else begin
			pcPlus4Out = pcPlus4In;
			instructionOut = instructionIn;
		end
	end
endmodule

module IDEX(pcPlus4Out, instructionOut, readOut1Out, readOut2Out, pcPlus4In, instructionIn,
readOut1In, readOut2In, clock, reset);
	input [31:0] pcPlus4In, readOut1In, readOut2In, instructionIn;
	input clock, reset;
	output reg [31:0] pcPlus4Out, readOut1Out, readOut2Out, instructionOut;

	always @ (posedge clock or negedge reset) begin
		if (~reset) begin
			pcPlus4Out = 0;
			readOut1Out = 0;
			readOut2Out = 0;
			instructionOut = 0;
		end else begin
			pcPlus4Out = pcPlus4In;
			readOut1Out = readOut1In;
			readOut2Out = readOut2In;
			instructionOut = instructionIn;
		end
	end

endmodule
```

```
module EXMEM(resultOut, readOut2Out, instructionOut, ZOut, resultIn, readOut2In,
instructionIn, ZIn, clock, reset);
	input [31:0] resultIn, readOut2In, instructionIn;
	input ZIn, clock, reset;
	output reg [31:0] resultOut, readOut2Out, instructionOut;
	output reg ZOut;

	always @ (posedge clock or negedge reset) begin
		if (~reset) begin
			resultOut = 0;
			readOut2Out = 0;
			ZOut = 0;
			instructionOut = 0;
		end else begin
			resultOut = resultIn;
			readOut2Out = readOut2In;
			ZOut = ZIn;
			instructionOut = instructionIn;
		end
	end
endmodule

module MEMWB(dataOutOut, resultOut, instructionOut, dataOutIn, resultIn, instructionIn, clock,
reset);
	input [31:0] dataOutIn, resultIn, instructionIn;
	input clock, reset;
	output reg [31:0] dataOutOut, resultOut, instructionOut;

	always @(posedge clock or negedge reset) begin
		if (~reset) begin
			dataOutOut = 0;
			resultOut = 0;
			instructionOut = 0;
		end else begin
			dataOutOut = dataOutIn;
			resultOut = resultIn;
			instructionOut = instructionIn;
		end
	end
endmodule
```

Verilog for Pipeline Decoders

```
// Instruction Decoders
```

```
module EX_decode(aluSel, shift, aluOp, regDestEX, memReadEX, memWriteEX, instruction);
        input [31:0] instruction;
        output aluSel, shift, regDestEX, memReadEX, memWriteEX;
        output [1:0] aluOp;

        wire Rformat, LW, SW, ADDI, SLL, BNE;
        wire [5:0] opcode, funct;

        assign opcode = instruction[31:26];
        assign funct = instruction[5:0];

        assign Rformat = ~|opcode;
        assign LW = ~|(opcode ^ 6'b100011);
        assign SW = ~|(opcode ^ 6'b101011);
        assign ADDI = ~|(opcode ^ 6'b001000);
        assign SLL = ~|funct;
        assign BNE = ~|(opcode ^ 6'b000101);

        // selects the 16 bit immediate value as the input to the alu
        assign aluSel = LW | SW | ADDI;


        // selects read operation for data memory
        assign memReadEX = LW;

        assign memWriteEX = SW;

        // assert shift control output to change the second input to the ALU to come from the shift
amount.
        // also changes the first amount to be from the rt field (not rs field as is usual)
        assign shift = Rformat & SLL;

        assign regDestEX = Rformat;

        assign aluOp[1] = Rformat;
        assign aluOp[0] = BNE;

endmodule

module MEM_decode(memRead, memWrite, branch, jump, jumpR, regDestMEM,
writeEnableMEM, instruction);
        input [31:0] instruction;
        output memRead, memWrite, branch, jump, jumpR, regDestMEM, writeEnableMEM;
```

```
	wire J, JR, Rformat, LW, SW, BNE, ADDI;
	wire [5:0] opcode, funct;

	assign opcode = instruction[31:26];
	assign funct = instruction[5:0];

	assign J = ~|(opcode ^ 6'b000010);
	assign JR = ~|(funct ^ 6'b001000);
	assign Rformat = ~|opcode;
	assign LW = ~|(opcode ^ 6'b100011);
	assign SW = ~|(opcode ^ 6'b101011);
	assign BNE = ~|(opcode ^ 6'b000101);
	assign ADDI = ~|(opcode ^ 6'b001000);

	// selects read operation for data memory
	assign memRead = LW;

	// selects write operation for data memory
	assign memWrite = SW;

	// selects branch operation for control block
	assign branch = BNE;

	// selects the jump operation for the control block (note to self, likely that we will actually
need two control sigs when JR is implemented).
	assign jump = J;

	assign jumpR = Rformat & JR;

	assign regDestMEM = Rformat;

	assign writeEnableMEM = Rformat | LW | ADDI;

endmodule

module WB_decode(writeEnable, memToReg, regDest, instruction);
	input [31:0] instruction;
	output memToReg, regDest, writeEnable;

	wire [5:0] opcode;
	wire Rformat, LW, ADDI;

	assign opcode = instruction[31:26];
	assign Rformat = ~|opcode;
	assign LW = ~|(opcode ^ 6'b100011);
	assign ADDI = ~|(opcode ^ 6'b001000);

	// selects the write data as the  16 bit value read from data memory rather than alu result
	assign memToReg = LW;

	// selects the destination register address as the rd field [15:11] otherwise rt field [20:16]
	assign regDest = Rformat;

	// enables writing to the register file
	assign writeEnable = Rformat | LW | ADDI;

endmodule
```

Appendix D – TESTING FILES

First test for Single Cycle CPU

```
# a, at address 0 gets 7
addi $t0, $zero, 7
sw $t0, 0($zero)

# b at address 1 gets 5
addi $t0, $zero, 5
sw $t0, 1($zero)

# c at address 2 gets 2
addi $t0, $zero, 2
sw $t0, 2($zero)

# d at address 3 gets 4
addi $t0, $zero, 4
sw $t0, 3($zero)

# dPtr at address 4 gets address of d (3)
addi $t0, $zero, 3
sw $t0, 4($zero)

# load a to $s0
lw $a0, 0($zero)

# load b to $s1
```

```
lw $a1, 1($zero)

# load c to $s2
lw $a2, 2($zero)

# load d to $s3
lw $a3, 3($zero)

# ------------IF---------------

# $t4 gets A - B
sub $t4, $a0, $a1

#$t6 gets three
addi $t6, $zero, 3

# $t5 gets slt of A-B and $t6 (3)
slt $t5, $t6, $t4

bne $t5, $zero, Else

add $a2, $a2, $a3 # c = c + d
sw $a2, 2($zero)

sll $a3, $a3, 2 # d = d << 2
sw $a3, 3($zero)

j EndIf

# --------ELSE----------------

Else:
sll $a2, $a2, 3 # c = c << 5
sll $a2, $a2, 2
sw $a2, 2($zero)

addi $t5, $zero, 7 # t5 get 7
lw $t9, 4($zero) # dptr into t9
sw $t5, 0($t9) # store t5 at address of t9

EndIf:
```

First test Machine Code for Single Cycle CPU

```
@0
```

```
20080007ac08000020080005ac080001
@1
20080002ac08000220080004ac080003
@2
20080003ac0800048c0400008c050001
@3
8c0600028c07000300856022200e0003
@4
01cc682a15a0000500c73020ac060002
@5
00073880ac0700030800001d000630c0
@6
00063080ac060002200d00078c190004
@7
af2d0000000000000000000000000000
```

Second test for Single Cycle CPU

```
# a, at address 0 gets 7
addi $t0, $zero, 8
sw $t0, 0($zero)

# b at address 1 gets 5
addi $t0, $zero, 4
sw $t0, 1($zero)

# c at address 2 gets 2
addi $t0, $zero, 2
sw $t0, 2($zero)

# d at address 3 gets 4
addi $t0, $zero, 4
sw $t0, 3($zero)

# dPtr at address 4 gets address of d (3)
addi $t0, $zero, 3
sw $t0, 4($zero)

# load a to $s0
lw $a0, 0($zero)

# load b to $s1
lw $a1, 1($zero)

# load c to $s2
```

```
lw $a2, 2($zero)

# load d to $s3
lw $a3, 3($zero)

# ------------IF---------------

# $t4 gets A - B
sub $t4, $a0, $a1

#$t6 gets three
addi $t6, $zero, 3

# $t5 gets slt of A-B and $t6 (3)
slt $t5, $t6, $t4

bne $t5, $zero, Else

add $a2, $a2, $a3 # c = c + d
sw $a2, 2($zero)

sll $a3, $a3, 2 # d = d << 2
sw $a3, 3($zero)

j EndIf

# --------ELSE----------------

Else:
sll $a2, $a2, 3 # c = c << 5
sll $a2, $a2, 2
sw $a2, 2($zero)

addi $t5, $zero, 7 # t5 get 7
lw $t9, 4($zero) # dptr into t9
sw $t5, 0($t9) # store t5 at address of t9

EndIf:
```

Second test Machine Code for Single Cycle CPU

```
@0
20080008ac08000020080004ac080001
@1
```

```
20080002ac08000220080004ac080003
@2
20080003ac0800048c0400008c050001
@3
8c0600028c07000300856022200e0003
@4
01cc682a15a0000500c73020ac060002
@5
00073880ac0700030800001d000630c0
@6
00063080ac060002200d00078c190004
@7
af2d0000000000000000000000000000
```

Third test for Single Cycle CPU

```
#x, at address 0 gets 0x005F
addi $t0, $zero, 95
sw $t0, 0($zero)

#y, at address 2 gets 0x00AF
addi $t0, $zero, 175
sw $t0, 2($zero)

# load x to a0
lw $a0, 0($zero)

# load y to a1
lw $a1, 2($zero)

# x and y store to t0
and $t0, $a0, $a1

# addi
addi $t9, $zero, 52

# jump to
jr $t9

# noop (4)

# x or y to t1
or $t1, $a0, $a1
```

```
# x xor y to t2
xor $t2, $a0, $a1
```

Third Test Machine Code for Single Cycle CPU

```
@0
2008005fac080000200800afac080002
@1
8c0400008c0500020085402420190036
@2
03200008000000000000000000000000
@3
00000000008548250085502600000000
```

First test for Pipelined CPU – Final test with branch

```
# A at address 0 gets 6
addi $t0, $zero, 6
sw $t0, 0($zero)

# B at address 1 gets 4
addi $t0, $zero, 4
sw $t0, 1($zero)

# C at address 2 gets 2
addi $t0, $zero, 2
sw $t0, 2($zero)

# D at address 3 gets 4
addi $t0, $zero, 4
sw $t0, 3($zero)

# dptr at address 4 gets address of D (which is 3)
addi $t0, $zero, 3 # 3 is address of d (dPtr)
sw $t0, 4($zero)

# E at address 5 gets 0x7676
addi $t6, $zero, 30326
sw $t6, 5($zero)

# F at address 6 gets 0xA5A5
addi $t7, $zero, 42405
```

```
sw $t7, 6($zero)

# G at address 7 gets 0xFF
addi $t8, $zero, 255
sw $t8, 7($zero)

# H at address 8 gets 0xC3
addi $t9, $zero, 195
sw $t9, 8($zero)

# store A-D into registers
lw $a0, 0($zero)
lw $a1, 1($zero)
lw $a2, 2($zero)
lw $a3, 3($zero)

sub $t2, $a0, $a1

# t0 still has 3 in it
slt $t1, $t2, $t0

bne $t1, $zero, Else
sll $zero, $zero, $zero
# c = c + 4 or equivalently c = c + d
add $a2, $a2, $a3
sw $a2, 2($zero)

# d = c - 3
sub $a3, $a2, $t0
sw $a3, 3($zero)

# g = E | F
or $t8, $t6, $t7
sw $t8, 7($zero)

j End
sll $zero, $zero, $zero
Else:

# c = c << 3
sll $a2, $a2, 3
sw $a2, 2($zero)

# get dptr into t5
```

```
lw $t5, 4($zero)
addi $t4, $zero, 7

# store dptr to address of t5
sw $t4, 0($t5)

# g = e and f
and $t8, $t6, $t7
sw $t8, 7($zero)

End:

add $a0, $a0, $a1
sw $a0, 0($zero)

# g = (e ^ f) & H
xor $t3, $t6, $t7
and $t8, $t3, $t9
sw $t8, 7($zero)
```

First test Machine code for Pipelined CPU – Final test without branch

```
@0
20080006ac08000020080004ac080001
@1
20080002ac08000220080004ac080003
@2
20080003ac080004200e7676ac0e0005
@3
200fa5a5ac0f0006201800ffac180007
@4
201900c3ac1900088c0400008c050001
@5
8c0600028c0700030085502201448482a
@6
15200008000000000c73020ac060002
@7
00c83822ac07000301cfc025ac180007
@8
0800002900000000000630c0ac060002
@9
8c0d0004200c0007adac000001cfc024
@A
ac18000700852020ac04000001CF5826
```

```
@C
0179c024ac18000700000000000000000
```

Second test for Pipelined CPU

```
addi $t0, $zero, 6
sw $t0, 0($zero)

addi $t0, $zero, 2
sw $t0, 1($zero)

addi $t0, $zero, 2
sw $t0, 2($zero)

addi $t0, $zero, 4
sw $t0, 3($zero)

addi $t0, $zero, 3 # 3 is address of d (dPtr)
sw $t0, 4($zero)

addi $t6, $zero, 30326
sw $t6, 5($zero)

addi $t7, $zero, 42405
sw $t7, 6($zero)

addi $t8, $zero, 255
sw $t8, 7($zero)

addi $t9, $zero, 195
sw $t9, 8($zero)

lw $a0, 0($zero)
lw $a1, 1($zero)
lw $a2, 2($zero)
lw $a3, 3($zero)

sub $t2, $a0, $a1

# t0 still has 3 in it
slt $t1, $t2, $t0

bne $t1, $zero, Else
sll $zero, $zero, $zero
```

```
# c = c + 4 or equivalently c = c + d
add $a2, $a2, $a3
sw $a2, 2($zero)

# d = c - 3
sub $a3, $a2, $t0
sw $a3, 3($zero)

# g = E | F
or $t8, $t6, $t7
sw $t8, 7($zero)

j End
sll $zero, $zero, $zero
Else:

# c = c << 3
sll $a2, $a2, 3
sw $a2, 2($zero)

# get dptr into t5
lw $t5, 4($zero)
addi $t4, $zero, 7

# store dptr to address of t5
sw $t4, 0($t5)

# g = e and f
and $t8, $t6, $t7
sw $t8, 7($zero)

End:

# a = a + b
add $a0, $a0, $a1
sw $a0, 0($zero)

# g = (e ^ f) & H
xor $t3, $t6, $t7
and $t8, $t3, $t9
sw $t8, 7($zero)
```

Second test Machine code for Pipelined CPU

```
@0
20080006ac08000020080002ac080001
@1
20080002ac08000220080004ac080003
@2
20080003ac080004200e7676ac0e0005
@3
200fa5a5ac0f0006201800ffac180007
@4
201900c3ac1900088c0400008c050001
@5
8c0600028c0700030085502220148482a
@6
15200008000000000000c73020ac060002
@7
00c83822ac07000301cfc025ac180007
@8
0800002900000000000630c0ac060002
@9
8c0d0004200c0007adac000001cfc024
@A
ac18000700852020ac04000001CF5826
@C
0179c024ac1800070000000000000000
```

Third test Machine code for Pipelined CPU – No data or control hazards

```
@0
20080009000000000000000000000000
@1
00000000000000002009000800000000
@2
00000000000000000000000000000000
@3
00000000000000000109502200000000
```

Fourth test Machine code for Pipelined CPU – No control hazards

```
@0
20080007ac08000020080005ac080001

@1
20080002ac08000220080004ac080003
```

```
@2
20080003ac0800048c04000000844020

@3
8c05000100a540208c06000200000000
```