E85: Digital Electronics and Computer Engineering
Lab 10: Multicycle Controller

Objective

In Labs 10 and 11, you will design a multicycle ARM processor in SystemVerilog and test it on a simple machine language program. This will tie together everything that you have learned in E85 about digital design, hardware description languages, assembly language, and microarchitecture, and give you the chance to design and debug a complex system. In Lab 10, you will build and test the controller. In Lab 11, you will build the datapath and test the whole system. You will need a working multicycle processor for your final exam.

1. Multicycle ARM Controller

Before you start developing the controller, make sure to take a look at the following diagrams. All figures and tables are provided at the end of this document.

- ➢ *Figure 7.41* (p. 423) shows the Main FSM.
- ➢ *Table 7.6* (p. 416) defines the Instruction Decoder.
- ➢ Page 400 and *HDL Example 7.3* (p. 446) show the PC Logic.
- ➢ *Table 7.3* (p. 400) and *HDL Example 7.3* (p. 446) define the ALU Decoder.
- ➢ *Table 6.3* (p. 307) and *HDL Example 7.4* (p. 447) have the Condition Check logic.
- ➢ *Figure 7.31* (p. 415) shows the controller for the multicycle processor implementing a subset of ARMv4.

Write a hierarchical Verilog description of the multicycle controller. When outputs are don't care, set them to 0 so they have a deterministic value to simplify testing.

The controller should have the following module declaration and should follow the hierarchy of Figure 7.31. Remember that Op, Funct, and Rd are bitfields of Instr. Remember that ALUFlags[3:0] correspond to N, Z, C, and V, respectively.

```
module controller(input  logic         clk,
                  input  logic         reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic         PCWrite,
                  output logic         MemWrite,
                  output logic         RegWrite,
                  output logic         IRWrite,
                  output logic         AdrSrc,
                  output logic [1:0]   RegSrc,
                  output logic         ALUSrcA,
                  output logic [1:0]   ALUSrcB,
                  output logic [1:0]   ResultSrc,
                  output logic [1:0]   ImmSrc,
                  output logic [1:0]   ALUControl);
```

2. Test Bench

Generating good test vectors is often harder than writing the code you are testing. This semester, the vectors are provided for you to increase the amount of sleep you'll get. Get the controller_testbench.sv and controller.tv from the class website. Read them and understand what they are doing.

Compile and test your controller with Modelsim. Make sure you run for long enough to get a message that all of the tests were completed with 0 errors.

3. Debugging Hints

Unless you are extraordinary unlucky, your controller won't work perfectly on the first try. If it did work, you would have missed out on the main learning objective of this lab and the next, which is how to systematically debug a complex system. You will need your controller in Lab 11, so take the time to fully debug.

Here are some tips to reduce the amount of time that debugging will take.

Minimize the number of bugs you have

Each bug takes a long time to locate, so a bit of extra time during the design phase can save you a lot of time during the debug phase.

- Remember that you are building hardware, so sketch the hardware you want and write the Verilog idioms that imply that hardware. Don't fall into the trap of writing Verilog code without thinking of the hardware it is implying.
- Proofread your code. Make sure your signal names are spelled consistently and that module inputs/outputs are listed in the correct order.
- Synthesize your design once in Quartus and look for warnings or errors. Make sure you understand which warnings are normal (e.g. no timing constraints set) and which need to be fixed. Take these warnings very seriously; they are the fastest way to detect subtle bugs in your design.
- Simulate your design with Modelsim and look for warnings when compiling. Modelsim has a different Verilog analyzer and will detect types of mistakes that don't produce warnings in Quartus. Take these warnings seriously too.

Minimize the time it takes to run a test

Once you are in the debugging phase, choose a workflow that is efficient so you can make a change to your code and rerun the test in a matter of seconds rather than minutes.

- All testing can be done in Modelsim. You do not need to use Quartus, and recompiling in Quartus is an unnecessary time-consuming step. However, if you have made major changes, you might wish to occasionally resynthesize the design in Quartus and look for warnings hinting that you've introduced new bugs.

- Add relevant waveforms in Modelsim. It's usually worthwhile to add all the signals in a module that you are debugging so that you don't have to go through the tedious process of adding more signals and resimulating. Change the radix to display 32-bit signals in hexadecimal.
- Remember that you don't need to restart Modelsim and re-add signals each time you change your code. Instead:
    - Compile -> Compile All
    - Make sure you have no warnings
    - At the command line, rerun the simulation by typing
        - `restart -f`
        - `run 1000` (or however long you wish to run)

Systematically find your bugs

Inexperienced designers can waste enormous amounts of time debugging without a clear plan in mind. The following techniques can save you many hours.

- Understand what the expected inputs and outputs should be. Write down your expectations. This takes time, but will usually save far more time than it takes.
- Find the first place where a signal doesn't match your expectations. One bad signal will usually trigger others downstream, so focus your debugging on the first known error and don't worry yet about subsequent errors. For example, if tests 1 and 5 fail, start debugging test 1, not test 5.
- Make sure the simulator displays all signals involved in computing the bad signal. If necessary, add them to the simulation and resimulate as given above. If one of these inputs is bad, repeat this process to continue tracing it back.
- Once all the inputs are good and the output is bad, you've localized your bug. Examine the relevant Verilog module and fix the mistake.
- Repeat this process until all bugs have been fixed.

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. Hierarchical SystemVerilog for your controller module matching the declaration given above.
3. Does your controller pass your test vectors?

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.

Figure 7.31 Multicycle control unit

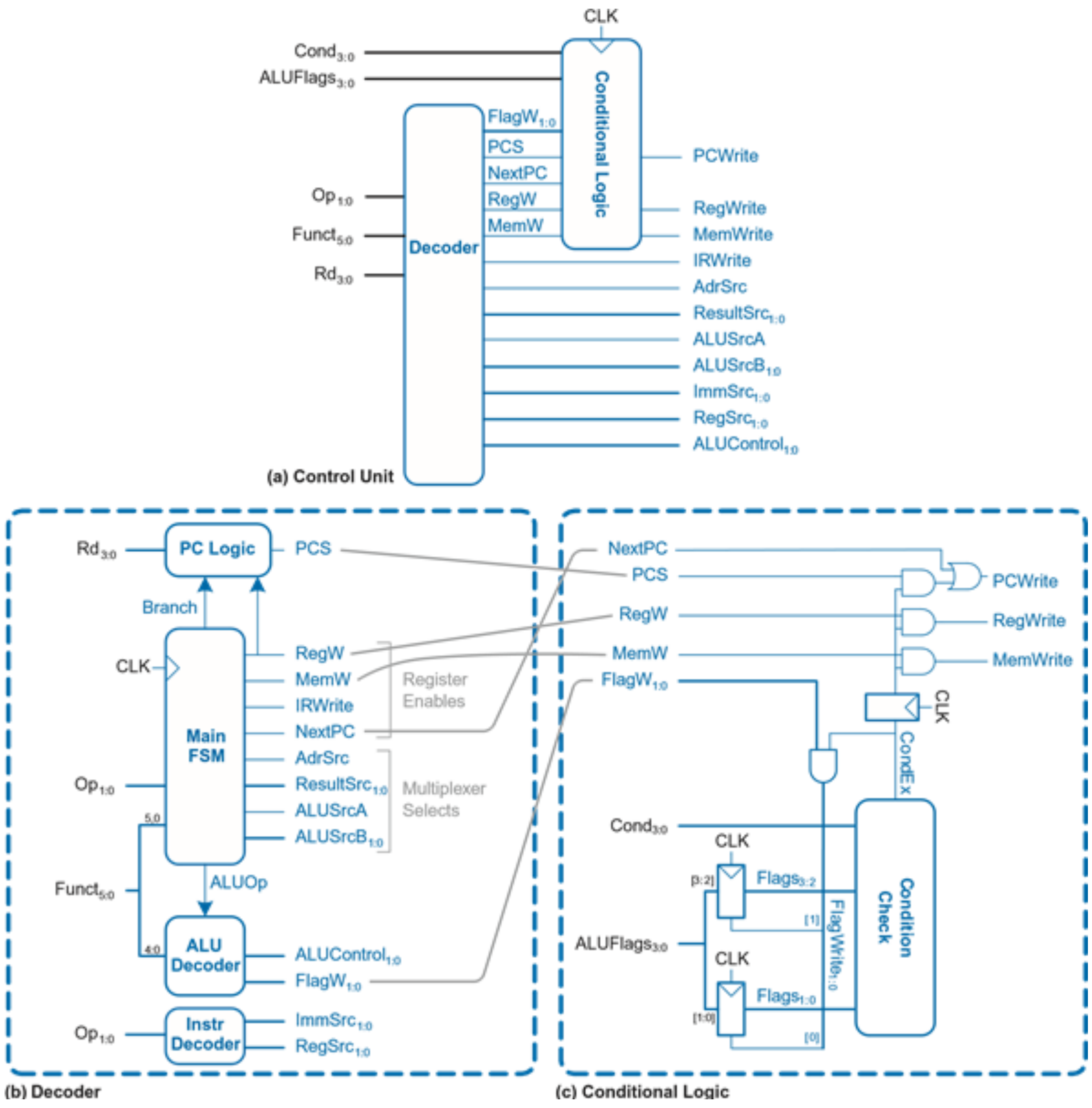
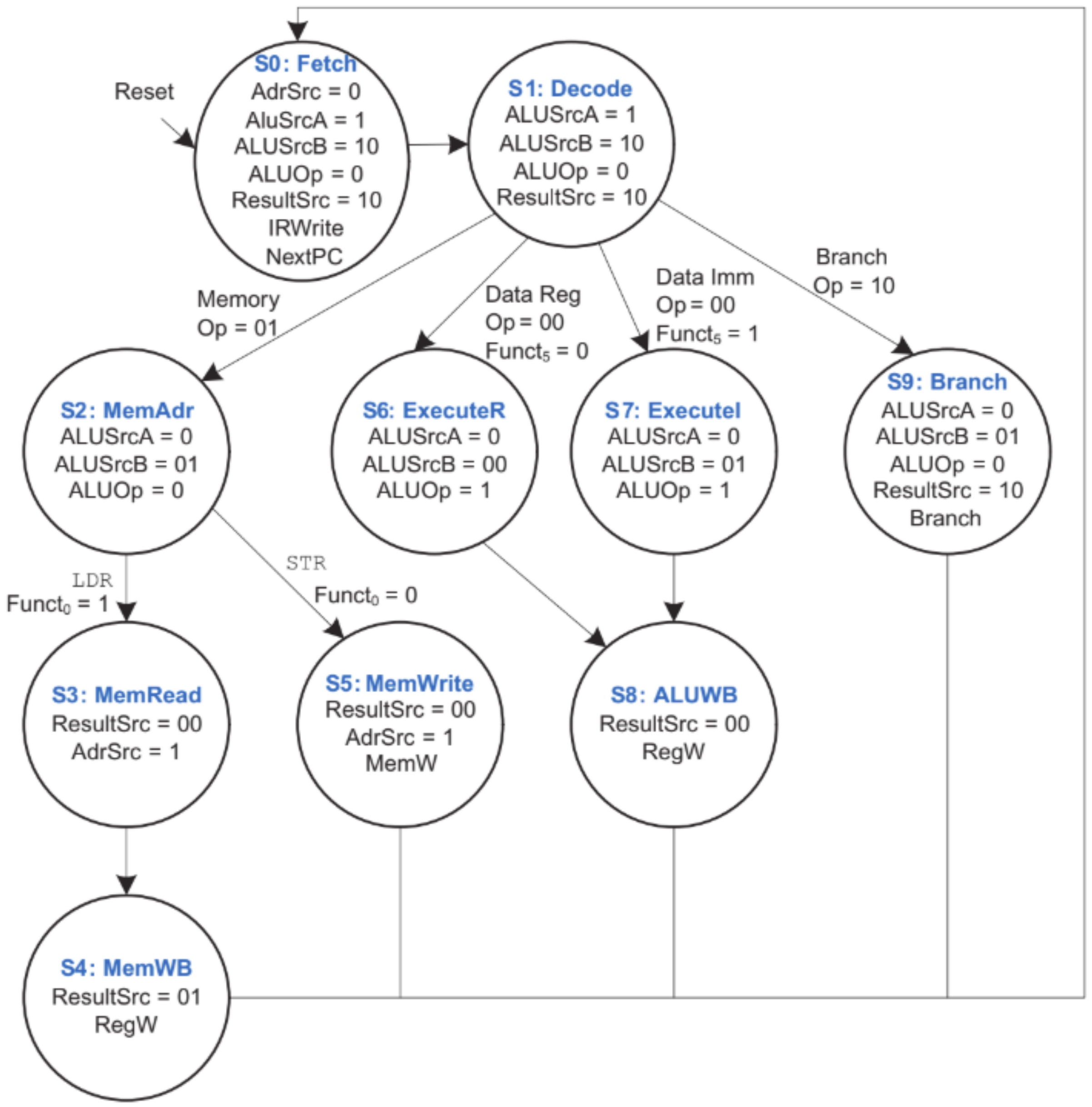


Figure 7.41 Complete multicycle control FSM



State	**Datapath μOp**
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC+4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset

Table 6.3 Condition mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Table 7.3 ALU Decoder truth table

ALUOp	*Funct*$_{4:1}$ (*cmd*)	*Funct*$_0$ (*S*)	Type	*ALUControl*$_{1:0}$	*FlagW*$_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

Table 7.6 Instr Decoder logic for *RegSrc* and *ImmSrc*

Instruction	*Op*	*Funct*$_5$	*Funct*$_0$	*RegSrc*$_1$	*RegSrc*$_0$	*ImmSrc*$_{1:0}$
LDR	01	X	1	X	0	01
STR	01	X	0	1	0	01
DP immediate	00	1	X	X	0	00
DP register	00	0	X	0	0	00
B	10	X	X	X	1	10

HDL Example 7.3 Decoder

```
module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic       PCS, RegW, MemW,
               output logic       MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RedSrc, ALUControl);

      logic [9:0] controls;
      logic             Branch, ALUOp;

      //Main Decoder
      always_comb
            casex(Op)
                                      // Data-processing immediate
                  2'b00: if(Funct[5]) controls = 10'b0000101001;
                                      // Data-processing register
                        else          controls = 10'b0000001001;
                                      // LDR
                  2'b01: if(Funct[0]) controls = 10'b0001111000;
                                      // STR
                        else          controls = 10'b1001110100;
                                      // B
                  2'b10:              controls = 10'b0110100010;
                                      // Unimplemented
                  default:            controls = 10'bx;
            endcase

      assign {RegSrc,ImmSrc,ALUSrc,MemtoReg,RegW,MemW,Branch,ALUOp}=controls;

      // ALU Decoder
      always_comb
      if(ALUOp) begin // which DP Instr
            case(Funct[4:1])
                  4'b0100: ALUControl = 2'b00; // ADD
                  4'b0010: ALUControl = 2'b01; // SUB
                  4'b0000: ALUControl = 2'b10; // AND
                  4'b1100: ALUControl = 2'b11; // ORR
                  default: ALUControl = 2'bx;  // unimplemented
            endcase

            // update flags if S bit is set (C&V only for arith)
            FlagW[1] = Funct[0];
            FlagW[0] = Funct[0]&(ALUControl==2'b00|ALUcontrol==2'b01);
      end else begin
            ALUControl = 2'b00; // add for non-DP instructions
            FlagW      = 2'b00; // don't update Flags
      end

      // PC Logic
      assign PCS = ((Rd==4'b1111)&RegW)|Branch;
endmodule
```

HDL Example 7.4 Conditional Logic

```
module condlogic(input  logic       clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic             PCS, RegW, MemW,
                 output logic             PCSrc, RegWrite, MemWrite);

      logic [1:0] FlagWrite;
      logic [3:0] Flags;
      logic       CondEx;

      flopenr #(2)flagreg1(clk,reset,FlagWrite[1],ALUFlags[3:2],Flags[3:2]);
      flopenr #(2)flagreg0(clk,reset,FlagWrite[0],ALUFlags[1:0],Flags[1:0]);

      //write controls are conditional
      condcheck cc(Cond, Flags, CondEx);
      assign FlagWrite = FlagW & {2{CondEx}};
      assign RegWrite  = RegW  & CondEx;
      assign MemWrite  = MemW  & CondEx;
      assign PCSrc     = PCS   & CondEx;
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

      logic neg, zero, carry, overflow, ge;

      assign {neg, zero, carry, overflow} = Flags;
      assign ge = (neg == overflow);

      always_comb
            case(Cond)
                  4'b0000: CondEx = zero;               // EQ
                  4'b0001: CondEx = ~zero;              // NE
                  4'b0010: CondEx = carry;              // CS
                  4'b0011: CondEx = ~carry;             // CC
                  4'b0100: CondEx = neg;                // MI
                  4'b0101: CondEx = ~neg;               // PL
                  4'b0110: CondEx = overflow;           // VS
                  4'b0111: CondEx = ~overflow;          // VC
                  4'b1000: CondEx = carry&~zero;        // HI
                  4'b1001: CondEx = ~(carry&~zero);     // LS
                  4'b1010: CondEx = ge;                 // GE
                  4'b1011: CondEx = ~ge;                // LT
                  4'b1100: CondEx = ~zero&ge;           // GT
                  4'b1101: CondEx = ~(~zero&ge);        // LE
                  4'b1110: CondEx = 1'b1;               // Always
                  default: CondEx = 1'bx;               // undefined
            endcase
endmodule
```