Digital Design and Computer Architecture

Lab 9: ARM Single-Cycle Processor

Introduction

In this lab you will build a simplified ARM single-cycle processor using SystemVerilog. You will combine your ALU from Lab 5 with the code for the rest of the processor taken from the textbook. Then you will load a test program and confirm that the system works. Next, you will implement two new instructions, and then write a new test program that confirms the new instructions work as well. By the end of this lab, you should thoroughly understand the internal operation of the ARM single-cycle processor.

Please read and follow the instructions in this lab carefully. In the past, many students have lost points for silly errors like not printing all the signals requested.

Before starting this lab, you should be very familiar with the single-cycle implementation of the ARM processor described in Section 7.3 of the Chapter 7 ARM draft, *Digital Design and Computer Architecture*. The single-cycle processor schematic from the text is repeated at the end of this lab assignment for your convenience. This version of the ARM single-cycle processor can execute the following instructions: `ADD`, `SUB`, `AND`, `ORR`, `LDR`, `STR`, and `B`.

Our model of the single-cycle ARM processor divides the machine into two major units: the control and the datapath. Each unit is constructed from various functional blocks. For example, as shown in the figure on the last page of this lab, the datapath contains the 32-bit ALU that you designed in Lab 5, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

1. ARM Single-Cycle Processor

The SystemVerilog single-cycle ARM module is given in Section 7.6 of the text. Use the electronic versions of all these files are in the class directory. Copy them to your own lab9_xx folder.

Study the files until you are familiar with their contents. Look in arm.sv. The top-level module (named `top`) contains the arm processor (`arm`) and the data and instruction memories (`dmem` and `imem`). Now look at the processor module (called `arm`). It instantiates two sub-modules, `controller` and `datapath`. Now take a look at the `controller` module and its submodules. It contains two sub-modules: `decode` and `condlogic`. The `decode` module produces all but three control signals. The `condlogic` module produces those remaining three control signals that update architectural state (`RegWrite`, `MemWrite`) or determine the next PC (`PCSrc`). These three signals depend on the condition mnemonic from the instruction

(Cond$_{3:0}$) and the stored condition flags (Flags$_{3:0}$) that are internal to the condlogic module. The condition flags produced by the ALU (ALUFlags$_{3:0}$) are updated in the flags registers dependent on the S bit (FlagW$_{1:0}$) and on whether the instruction is executed (again, dependent on the condition mnemonic Cond$_{3:0}$ and the stored value of the condition flags Flags$_{3:0}$). Make sure you thoroughly understand the controller module. Correlate signal names in the SystemVerilog code with the wires on the schematic.

After you thoroughly understand the controller module, take a look at the datapath SystemVerilog module. The datapath has quite a few submodules. Make sure you understand why each submodule is there and where each is located on the ARM single-cycle processor schematic. You'll notice that the alu module is not defined. Copy your ALU from Lab 5 into your lab9_xx directory. Be sure the module name matches the instance module name (alu), and make sure the inputs and outputs are in the same order as in they are expected in the datapath module.

The instruction and data memories instantiated in the top module are each a 64-word × 32-bit array. The instruction memory needs to contain some initial values representing the program. The test program is given in Figure 7.60 of the draft textbook. Study the program until you understand what it does. The machine language code for the program is stored in memfile.dat.

2. Testing the single-cycle ARM processor

In this section, you will test the processor with your ALU.

In a complex system, if you don't know what to expect the answer should be, you are unlikely to get the right answer. Begin by predicting what should happen on each cycle when running the program. Complete the chart in Table 1 at the end of the lab with your predictions. What address will the final STR instruction write to and what value will it write?

Simulate your processor with ModelSim. Refer to your earlier lab handouts if you need a refresher on how to use ModelSim. Be sure to add all of the .sv files, including the one containing your ALU. Add all of the signals from Table 1 to your waves window. (Note that many are not at the top level; you'll have to drill down into the appropriate part of the hierarchy to find them.)

Run the simulation. If all goes well, the testbench will print "Simulation succeeded." Look at the waveforms and check that they match your predictions in Table 1. If they don't, the problem is likely in your ALU or because you didn't properly add all of the files.

If you need to debug, you'll likely want to view more internal signals. However, on the final waveform that you turn in, show ONLY the following signals in this order: clk, reset, PC, Instr, ALUResult, WriteData, MemWrite, and ReadData. **All the values need to be output in hexadecimal and must be readable to get full credit.**

After you have fixed any bugs, print out your final waveform.

3. Modifying the ARM single-cycle processor

You now need to modify the ARM single-cycle processor by adding the EOR and LDRB instructions. First, modify the ARM processor schematic/ALU at the end of this lab to show what changes are necessary. You can draw your changes directly onto the schematics. Then

modify the main decoder and ALU decoder as required. Show your changes in the tables at the end of the lab. Finally, modify the SystemVerilog code as needed to include your modifications.

4. Testing your modified ARM single-cycle processor

Next, you'll need a test program to verify that your modified processor works. The program should check that your new instructions work properly and that the old ones didn't break. Use memfile2.asm below.

```
; memfile2.asm
; david_harris@hmc.edu and sarah_harris@hmc.edu 3 April 2014
MAIN        SUB  R0, R15, R15
            ADD  R1, R0, #255
            ADD  R2, R1, R1
            STR  R2, [R0, #196]
            EOR  R3, R1, #77
            AND  R4, R3, #0x1F
            ADD  R5, R3, R4
            LDRB R6, [R5]
            LDRB R7, [R5, #1]
            SUBS R0, R6, R7
            BLT  MAIN
            BGT  HERE
            STR  R1, [R4, #110]
            B     MAIN
HERE        STR  R6, [R4, #110]
```

Figure 1. ARM assembly program: memfile2.asm

Convert the program to machine language and put it in a file named memfile2.dat. Modify imem to load this file. Modify the testbench to check for the appropriate address and data value indicating that the simulation succeeded. Run the program and check your results. Debug if necessary. When you are done, print out the waveforms as before and indicate the address and data value written by the final `STR` instruction.

What to Turn In

Please turn in each of the following items, clearly labeled and in the following order:

1. **Please indicate how many hours you spent on this lab.** This will not affect your grade (unless omitted), but will be helpful for calibrating the workload for next semester's labs.
2. A completed version of Table 1.
3. An image of the simulation waveforms showing correct operation of the processor. Does it write the correct value to address 100?

   The simulation waveforms should give the signal values in hexadecimal format and should be in the following order: `clk`, `reset`, `PC`, `Instr`, `ALUResult`, `WriteData`,

`MemWrite`, and `ReadData`. While you may print more signals during debug, do not display any other signals in the waveform you submit. Check that the waveforms are zoomed out enough that the grader can read your bus values. Unreadable waveforms will receive no credit. Use several pages and multiple images as necessary.

4. Marked up versions of the datapath schematic and decoder tables that adds the `EOR` and `LDRB` instructions.
5. Your SystemVerilog code for your modified ARM computer (including `EOR` and `LDRB` functionality) with the changes highlighted and commented in the code.
6. The contents of your memfile2.dat containing your machine language code.
7. An image of the simulation waveforms showing correct operation of your modified processor on the new program. What address and data values are written by the final `STR` instruction?

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	$Flags_{3:0}$ [NZCV]	CondEx	WriteData	MemWrite	ReadData
1	1	00	SUB R0, R15, R15 E04F000F	0	0	0	0	?	1	0	0	x
2	0	04	ADD R2, R0, #5 E2802005	0	5	0	5	?	1	5	0	x
3	0	08	ADD R3, R0, #12 E280300C	0	C	0	C	?	1	C	0	x
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												

Table 1. First nineteen cycles of executing armtest.asm (all in hexadecimal, except $Flags_{3:0}$ in binary)

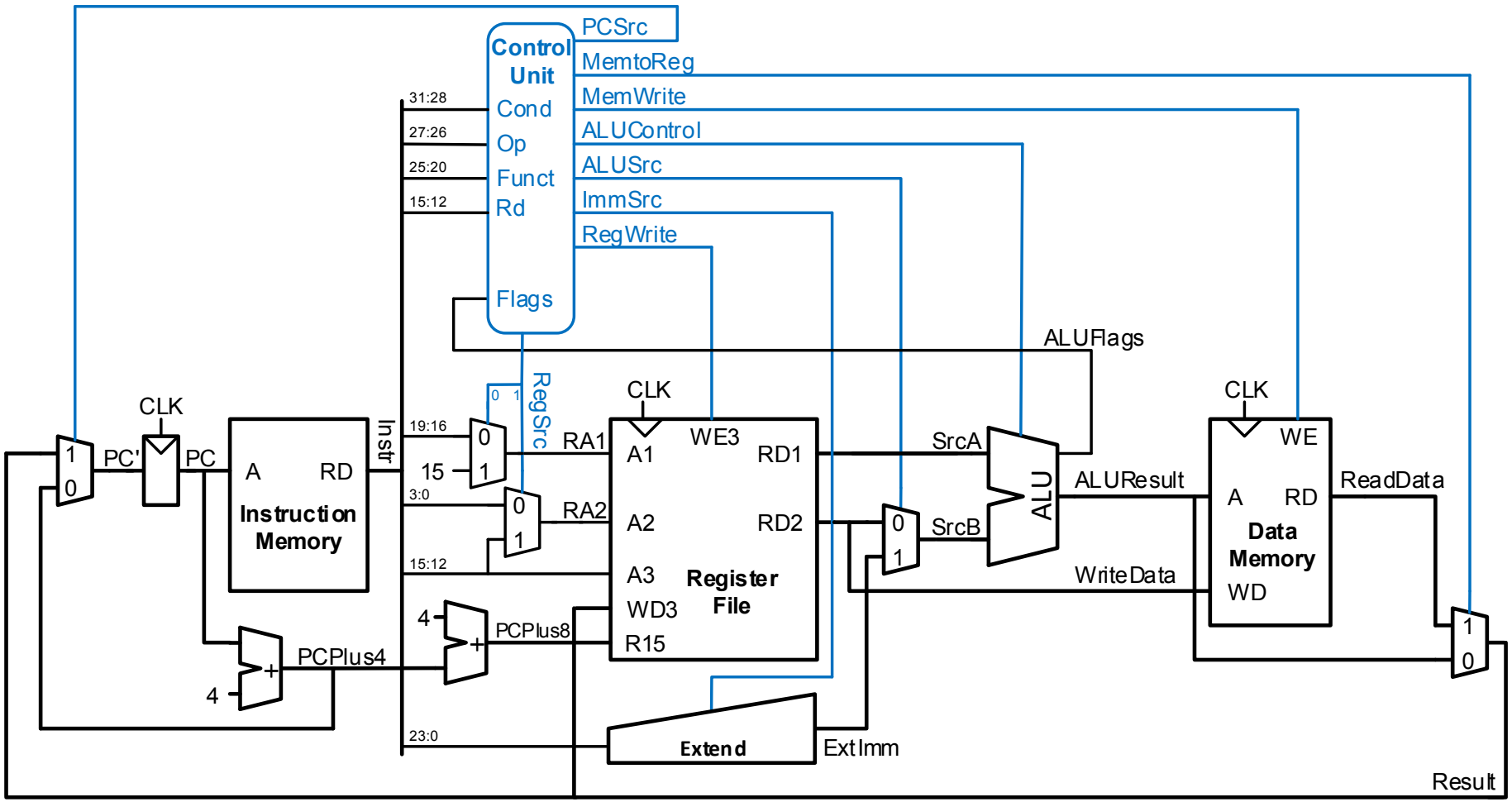

Figure 2. Single-cycle ARM processor

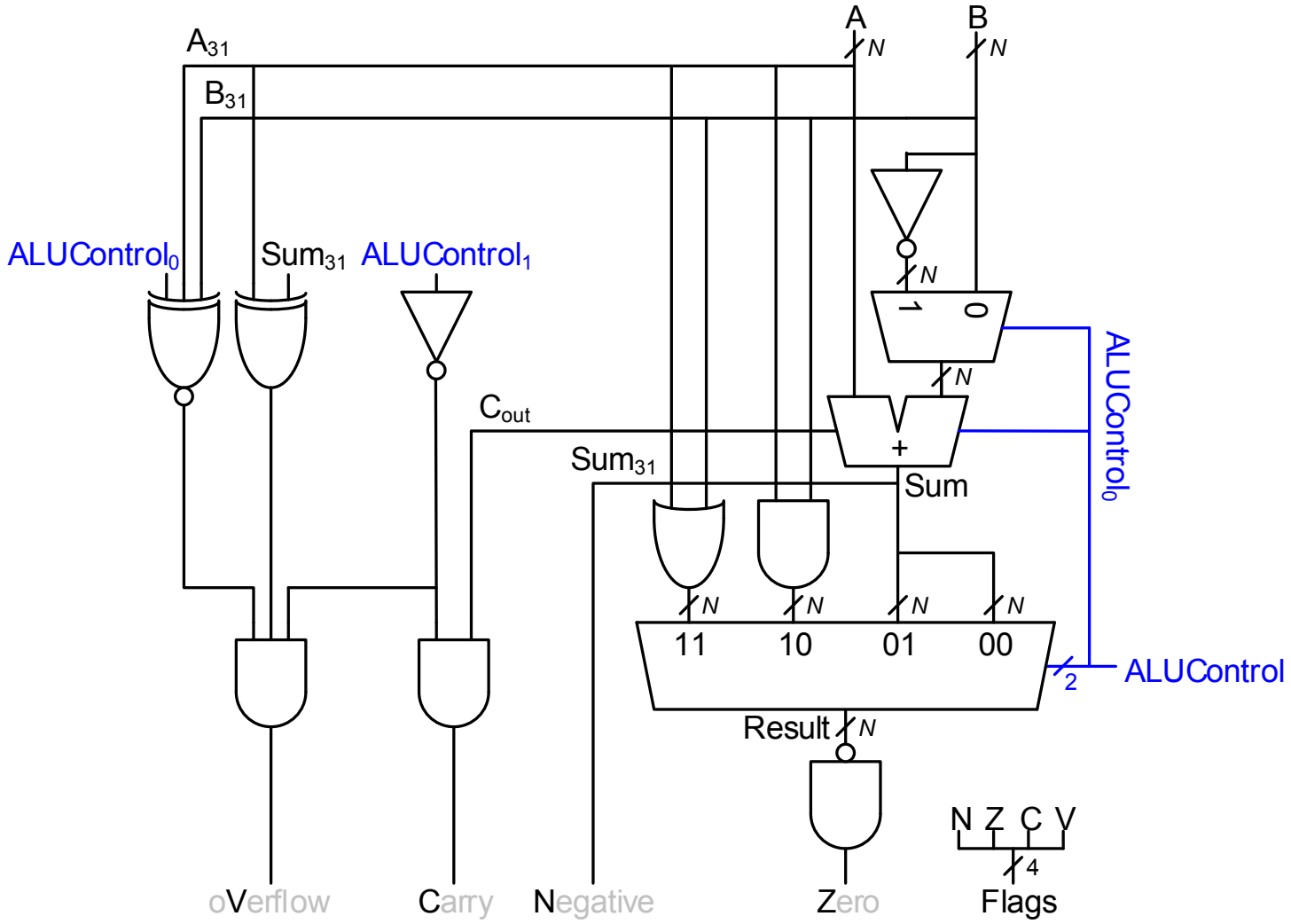

Figure 3. ARM ALU

Table 2. Extended functionality: Main Decoder

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Table 3. Extended functionality: ALU Decoder

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10