Lab 4 – ARM Single-Cycle Processor

Introduction

In this lab[1] you will build and program a simplified ARM single-cycle processor using SystemVerilog. You will combine the ALU from the previous homework 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. 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.

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 a previous homework, 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. The electronic versions of all these files are on Moodle. Copy them to your own folder.

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

[1] This homework is derived from a lab proposed by S. Harris and D.M. Harris in Digital Design and Computer Architecture ARM Edition.

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. 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 `MyARM_MultiCycle.sv` 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 MyProgram.hex.

2. Interface with the Raspberry PI

The interface with the Raspberry PI is realized through a SPI bus. Refer to chapter 9, I/0 Systems, (available as a PDF file from the web site of the reference book). Sections 9.3.1 to 9.3.4 give a complete description of the SPI and the general-purpose digital I/0 of the Raspberry.

A SPI slave implementation in SystemVerilog and associated script in Python are provided for this lab.

3. Programming the single-cycle ARM processor

Use the assembler program "`MyProgram.s`" that contains two parts: the first part checks that the instructions work properly while the second part implements a simple low-frequency counter and displays the output on the 8-Led of the DE0-nano. The counting frequency is fixed by the Raspberry PI.

Use `AssDeb` to compile, debug and get the .hex file for Quartus. Copy `MyProgram.hex` to your Quartus folder

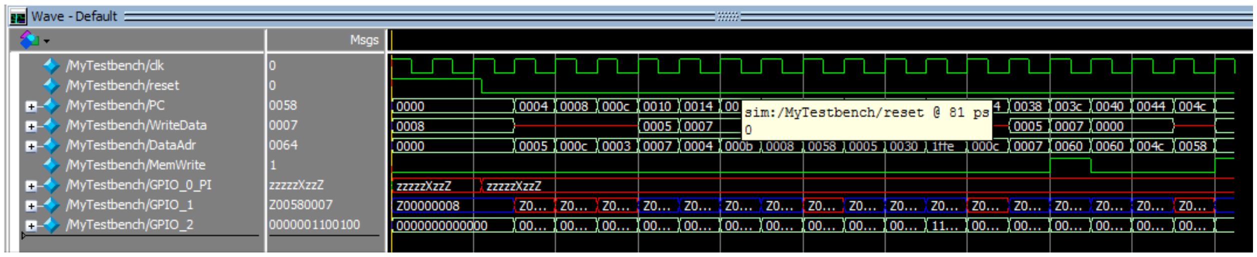
3. Testing the single-cycle ARM processor

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

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. 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, you need to debug your code and you'll likely want to view more internal signals.



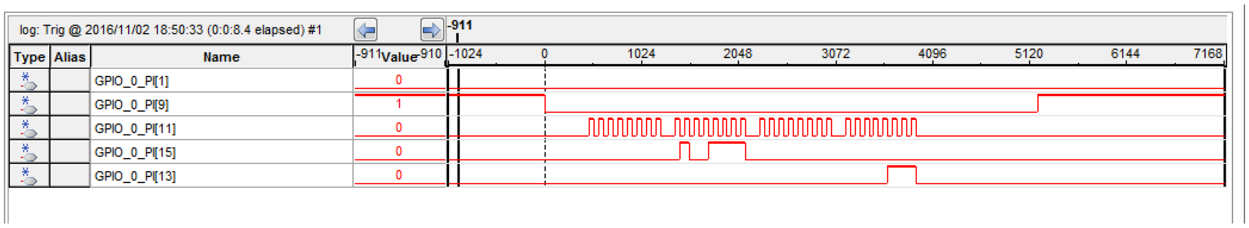
	Msgs
/MyTestbench/clk	0
/MyTestbench/reset	0
/MyTestbench/PC	0058
/MyTestbench/WriteData	0007
/MyTestbench/DataAdr	0064
/MyTestbench/MemWrite	1
/MyTestbench/GPIO_0_PI	zzzzzXzzZ
/MyTestbench/GPIO_1	Z00580007
/MyTestbench/GPIO_2	0000001100100

To set the frequency counting, use the following script

```
python3 MyARM.py
```

The script should return 0x00000007 (the value of R2 at the end of the first part)

SignalTap is an embedded logic analyzer and is a very useful tool to debug real-time systems. A configuration is provided to check the SPI signals.



log: Trig @ 2016/11/02 18:50:33 (0:0:8.4 elapsed) #1

Type	Alias	Name	Value
		GPIO_0_PI[1]	0
		GPIO_0_PI[9]	1
		GPIO_0_PI[11]	0
		GPIO_0_PI[15]	0
		GPIO_0_PI[13]	0

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	Flags$_{3:0}$ [NZCV]	CondEx	WriteData	MemWrite	ReadData
1	1	00	`SUB R0, R15, R15` `E04F000F`	8	8	0	0	?	1	8	0	x
2	0	04	`ADD R2, R0, #5` `E2802005`	0	5	0	5	?	1	x	0	x
3	0	08	`ADD R3, R0, #12` `E280300C`	0	C	0	C	?	1	x	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)

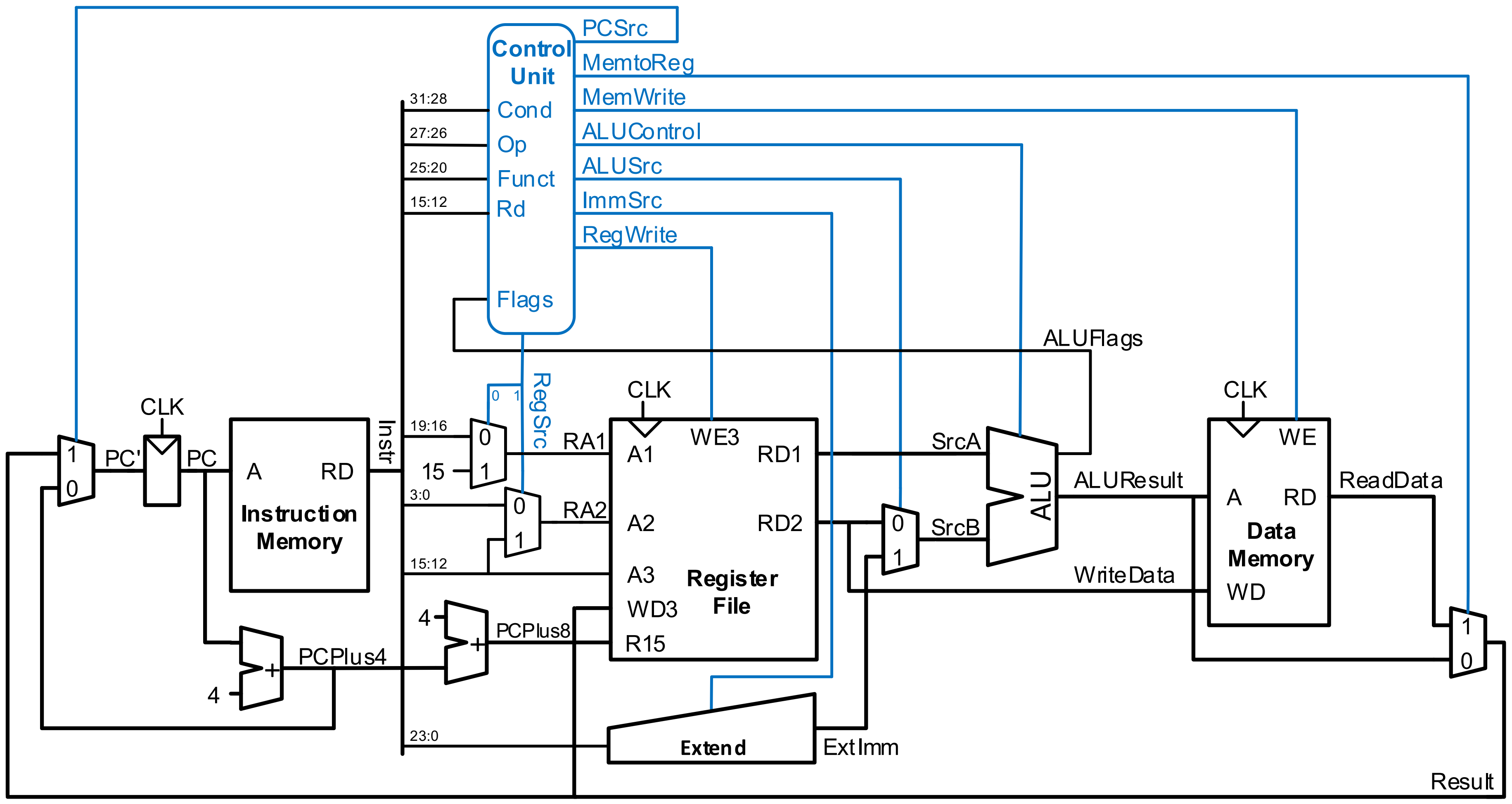


Figure 2. Single-cycle ARM processor

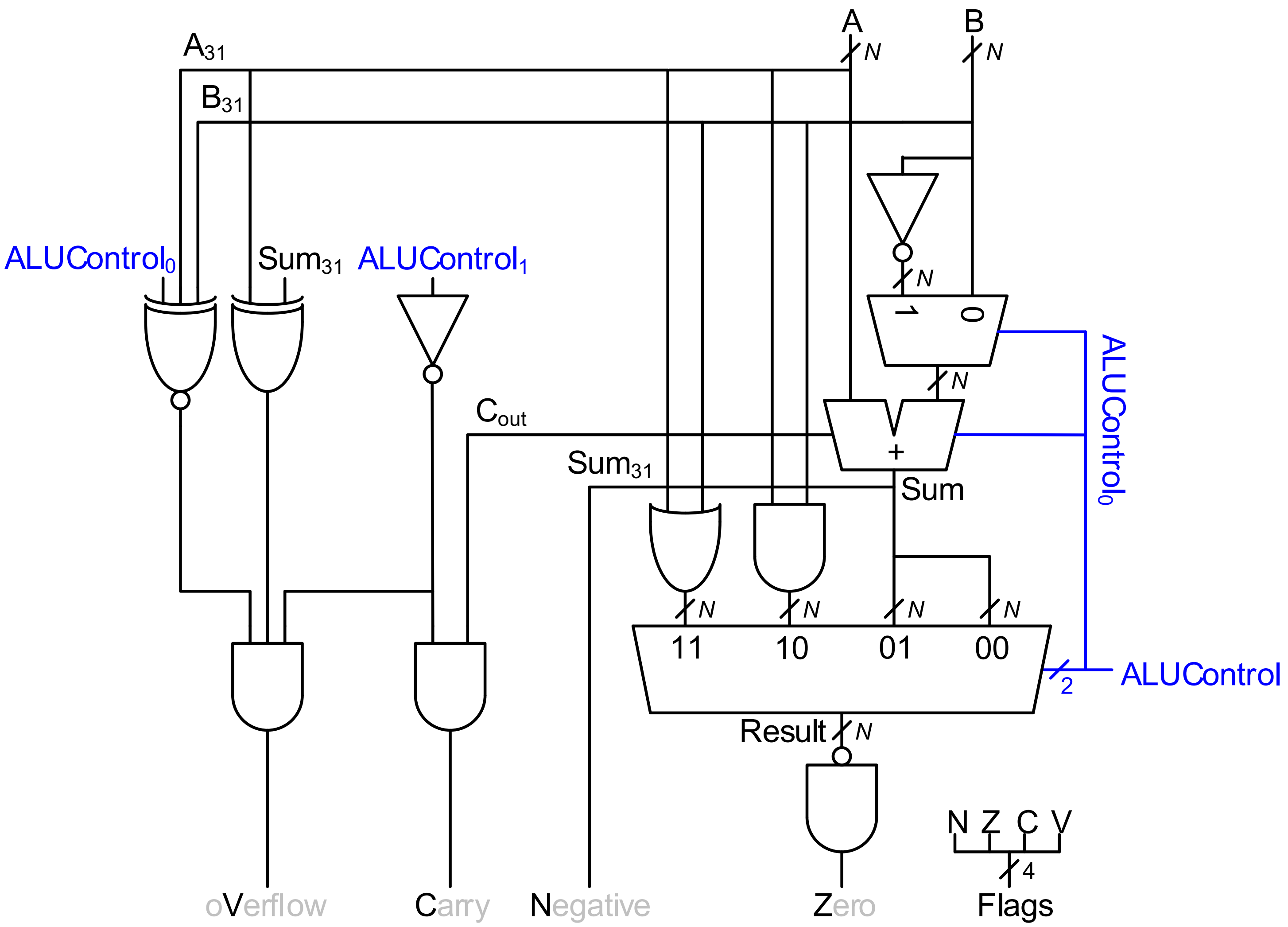


Figure 3. ARM ALU

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