ECE 469 Microprocessor System Design

Lab 2

ARM Single-Cycle Processor

Lab Objectives:

1. Develop and understand the peripheral components of a single cycle processor that join major pieces into a cohesive unit.
2. Implement new instructions into an existing framework.

Big Picture:

One of the major projects within this course is the design and simulation of a simple pipelined processor. Lab 1 had you build the two integral components of the processor. Lab 2 will have you link these components, as well as some additional logic to create a complete single cycle processor. Lab 3 will have you add pipelining to your processor and resolve the issues that come with the performance enhancement. For that reason, it is important you do not fall behind on this lab.

Introduction:

In this lab you will build a simplified ARM single-cycle processor using SystemVerilog. You will combine your ALU and Register File from Lab 1 with the code for the rest of the processor located in the lab folder provided by the class. Then you will load a test program and confirm that the system works. Next, you will implement two new instructions, and then run 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.

Before starting this lab, you should be familiar with the single-cycle implementation of the ARM processor. A schematic diagram is shown 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`.

Instruction	Format	Description	Bits
`ADD`	`ADD R1, R2, R3`	`R[D] = R[A] + R[B]`	`1110 000 0100 0 AAAA DDDD 0000 0000 BBBB`
	`ADD R1, R2, #10`	`R[D] = R[A] + I`	`1110 001 0100 0 AAAA DDDD 0000 IIII IIII`
`SUB`	`SUB R1, R2, R3`	`R[D] = R[A] - R[B]`	`1110 000 0010 0 AAAA DDDD 0000 0000 BBBB`
	`SUB R1, R2, #10`	`R[D] = R[A] - I`	`1110 001 0010 0 AAAA DDDD 0000 IIII IIII`
`AND`	`AND R1, R2, R3`	`R[D] = R[A] & R[B]`	`1110 000 0000 0 AAAA DDDD 0000 0000 BBBB`
`ORR`	`ORR R1, R2, R3`	`R[D] = R[A] | R[B]`	`1110 000 1100 0 AAAA DDDD 0000 0000 BBBB`
`LDR`	`LDR R1, [R2, #10]`	`R[D] = MEM[R[A] + 10]`	`1110 010 1100 1 AAAA DDDD IIII IIII IIII`
`STR`	`STR R1, [R2, #10]`	`MEM[R[A] + I] = R[D]`	`1110 010 1100 0 AAAA DDDD IIII IIII IIII`
`B`	`B TAG`	`PC = PC+(I<<2)`	`1110 1010 IIII IIII IIII IIII IIII IIII`

Table 1: Processor Base Instruction Set

ARM Single-Cycle Processor

Our arm processor and memory system are made up of 3 major blocks: the arm processor, the instruction memory and the data memory. Each is stored in its own distinct file, and they are all tied together with a top-level module as well as a testbench. We implore you to investigate each file and understand its contents:

top.sv

top is a structurally made toplevel module. It consists of 3 instantiations, as well as the signals that link them. It is almost totally self-contained, with no outputs and two system inputs: clk and rst. clk represents the clock the system runs on, with one instruction being read and executed every cycle. rst is the system reset and should be run for at least a cycle when simulating the system.

imem.sv

imem is the read only, 64 word x 32 bit per word instruction memory for our processor. Its module is written in RTL, and it strongly resembles a ROM (read only memory) or LUT (look up table). This memory has no clock, and cannot be written to, but rather it asynchronously reads out the word stored in its memory as soon as an address is given. The address and memory are byte aligned, meaning that the bottom two bits are discarded when looking for the word. One important line to note is the

```
Initial $readmemb(“memfile.dat”, memory);
```

which determines the contents of the memory when the system is initialized. You will alter this line to use programs given to you as a part of this lab.

dmem.sv

dmem is a more traditional, albeit very uninteresting, random access 64 word x 32 bit per word memory. This module is also written in RTL, and likely strongly resembles your own register file except for a few minor differences. The first is that there is only a single read port, compared to the register file’s two read ports. The other difference is that the dmem is also byte aligned, and therefore discards the bottom two bits of the address when doing a read or write.

arm.sv

arm is the spotlight of the show and contains the bulk of the datapath and control logic. This module is split into two parts, the datapath and control.
The datapath consists of a PC as well as a series of muxes to make decisions about which data words to pass forward and operate on. It is noticeably missing the register file and alu, which you will fill in using the modules made in lab 1. To correctly match up signals to the ports of the register file and alu take some time to study and understand the logic and flow of the datapath.
The control conists of a large decoder, which evaluates the top bits of the instruction and produces the control bits which become the select bits and write enables of the system. The write enables (RegWrite, MemWrite and PCSrc) are especially important because they are representative of your processors current state.
The decoder uses a case statement you may not have seen before, the casez. The casez is just like the case statement however it allows you to use wildcard bits so that the pattern matches regardless of what value is in the “?” position of the word. For more information on this and more see here.

testbench.sv

testbench is a simulation module which simply instantiates the processor system and runs 50 cycles of instructions before terminating. At termination, specific register file values are checked to verify the processors’ ability to execute the implemented instructions.

Task 1: Completing the Processor

Adding the Register File and ALU:

As mentioned previously, your first task will be to insert you register file and alu modules into the datapath of the arm processor. Make sure you fully understand what each signal represents prior to doing this and utilize schematics to visually verify or justify the port connections you've made. An important note to make is that there is additional logic in place to replace reads from R15 of the register file with that of a modified program counter. Keep this in mind when studying and visualizing the datapath.

Simulating the Processor:

In this section you will test and verify the processor through modelsim simulation. As of this moment the processor should be able to execute `ADD, SUB, AND, ORR, LDR, STR` and `B` instructions, with both immediate and register format for `ADD` and `SUB`. To accomplish this, the program given below utilizes every instruction to gradually fill the register file, culminating in the loading of a single word into the register. If any single of the instructions don't work, the wrong value will be loaded into the register file and the testbench will report a failure.

```
MAIN        ADD R0, R15, #0
            SUB R1, R0, R0
            ADD R2, R1, #10
            ADD R3, R0, R2
            SUB R4, R2, #3
            SUB R5, R3, R4
            ORR R6, R4, R5
            AND R7, R6, R5
            STR R7, [R1, #0]
            B SKIP
            STR R1, [R1, #0]
            B LOOP
SKIP        LDR R8, [R1, #0]
LOOP        B LOOP
```

Figure 1: Base Processor Testing Program "memfile.dat"

Before running a program, you should have expectations of what is supposed to be happening each cycle. To do this, fill in Table 3 in the appendix of this lab with values you expect to see on each cycle during execution. This table should then be checked against the simulation outputs after running the simulation.

While you may want to view many internal signals when debugging the simulation, your screenshot of the simulation must have these signals from top to bottom: `clk`, `reset`, `PC`, `Instr`, `ALUResult`, `WriteData`, `MemWrite`, `ReadData`. Additionally, the contents of the register file must be visible at the moment that the final `LDR` command is executed.

Task 2: Adding New Instructions:

Instruction	Format	Description	Bits
`CMP`	`CMP R1, R2, R3`	`R[D] = R[A] - R[B], Flag`	`1110 000 0010 1 AAAA DDDD 0000 0000 BBBB`
	`CMP R1, R2, #10`	`R[D] = R[A] – I, Flag`	`1110 001 0010 1 AAAA DDDD 0000 IIII IIII`
`BXX`	`BXX TAG`	`PC = PC+(I<<2) if COND`	`COND 1010 IIII IIII IIII IIII IIII IIII`
	`B`	`Unconditional`	`1110`
	`BEQ`	`Equal`	`0000`
	`BNE`	`Not Equal`	`0001`
	`BGE`	`Greater or Equal`	`1010`
	`BGT`	`Greater`	`1100`
	`BLE`	`Less or Equal`	`1101`
	`BLT`	`Less`	`1011`

Table 2: Added command and branching conditions

Adding the CMP/SUBS and Conditional Branching:

The processor, as it stands, has no specific purpose for the `ALUFlags` that it is computing during each operation. In the arm architecture, each instruction has four bits allocated to a condition. Every instruction up till now has executed unconditionally, but real programs often have if and else statements, as well as loops and functions which require conditional execution of instructions. To accommodate this, we introduce the `SUBS/CMP` instruction, which preforms the normal `SUB` instruction but also stores the resulting flags into a special flag register. This flag register can then be read and used to evaluate condition logic, to determine whether to take specific branches or to pass over the instruction without modifying any part of the processor state. The reason this command is used, is because subtracting two operands lets us figure out their relation relative to one another using the flags.

To accommodate these new instructions, we need make a few additions to the arm module. First, we must add a new register, `FlagsReg`, to the arm code. This register will store the flags from the most recent `CMP` command and should for that reason be set in an `always_ff` whenever a new control signal, `FlagWrite`, is asserted. At the same time, we need to improve the control to support this new instruction, either by adding a new entry or updating the `SUB` instruction.

Once we can save flags for future instructions, we need to use those flag bits to compute each condition defined by `EQ, NE, GE, GT, LE,` and `LT.` With these instructions we can finally update the branch instruction in the control to check that the instruction condition (`Instr[31:28]`) is valid before deciding to execute the branch, or otherwise ignoring the instruction completely. To ignore an instruction, we simply have to deassert any control logic that may update state such as `MemWrite, RegWrite, PCSrc` and `FlagWrite`.

Simulating the Updated Processor:

Simulating the updated processor should be just as straightforward as the base. In this case however, the program is much longer to effectively verify not only the original instructions but variations of the new conditional instructions as well.

```
MAIN        ADD R0, R15, #0
            SUB R1, R0, R0
            ADD R2, R1, #10
            ADD R3, R0, R2
            SUB R4, R2, #3
            SUB R5, R3, R4
            ORR R6, R4, R5
            AND R7, R6, R5
            STR R7, [R1, #0]
            B SKIP
            STR R1, [R1, #0]
            B LOOP
SKIP        LDR R8, [R1, #0]
B_START     CMP R9, R6, #15
            BNE B_START
            CMP R9, R5, R4
            BNE BNE_TESTED
            B B_START
BNE_TESTED  CMP R9, R2, R3
            BGE B_START
            CMP R9, R3, R2
            BGE BGE_TESTED
            B B_START
BGE_TESTED  CMP R9, R3, R2
            BLE B_START
            CMP R9, R2, R3
            BLE BLE_TESTED
            B B_START
BLE_TESTED  ADD R8, R1, #1
LOOP        B LOOP
```

Figure 2: Updated Processor Testing Program "memfile2.dat"

Before running a program, you should have expectations of which branches are expected to be taken and which are expected to be passed over. While there is no table to fill out for this section, in your report submission you should list the expected sequence of PC. This sequence will demonstrate the flow of the program, highlighting which instructions are skipped over and in which order.

While you may want to view many internal signals when debugging the simulation, your screenshot of the simulation must have these signals from top to bottom: `clk`, `rst`, `PC`, `Instr`, `ALUResult`, `WriteData`, `MemWrite`, `ReadData`. Additionally, the contents of the register file must be visible at the moment that the final `ADD` command is executed.

Deliverables:

Please turn in each of the following items:

1. A detailed lab report that includes the procedure and results for tasks 1 and 2, as well as an Appendix with the updated arm.sv, your alu.sv, your reg_file.sv as well as any other files you added or used to simulate the processor outside of the given files. The report should also include:
   - A complete version of Table 3.
   - The PC sequence for the memfile2.dat program.
   - The simulation waveforms embedded into the results sections. These should be ordered as described in the simulation sections, as well as highlight the final register file contents, specifically the final transition of register 8. Any other proof of functionality waveform images should be included as well.
   - Your design files (alu.sv, reg_file.sv, arm.sv) as well as any others you may have made and used pasted and formatted in the report Appendix.
   - Follow the report format as outlined in "lab report outline" and "lab report template" documents on canvas.
2. Each of the design files (alu.sv, reg_file.sv, arm.sv) as well as any others you may have made and used, submitted as distinct files into canvas.

Cycle	PC	Instr	SrcA	SrcB	ALUResult	WriteData	ReadData	MemWrite	RegWrite	Result
1	00	`ADD R0, R15, #0`	8	0	8	Don't Care	X	0	1	8
2	04	`SUB R1, R0, R0`	8	8	0	8	X	0	1	0
3	08	`ADD R2, R1, #10`	0	A	A	Don't Care	X	0	1	A
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										
17										
18										
19										

Table 3. First nineteen cycles of executing memfile.dat

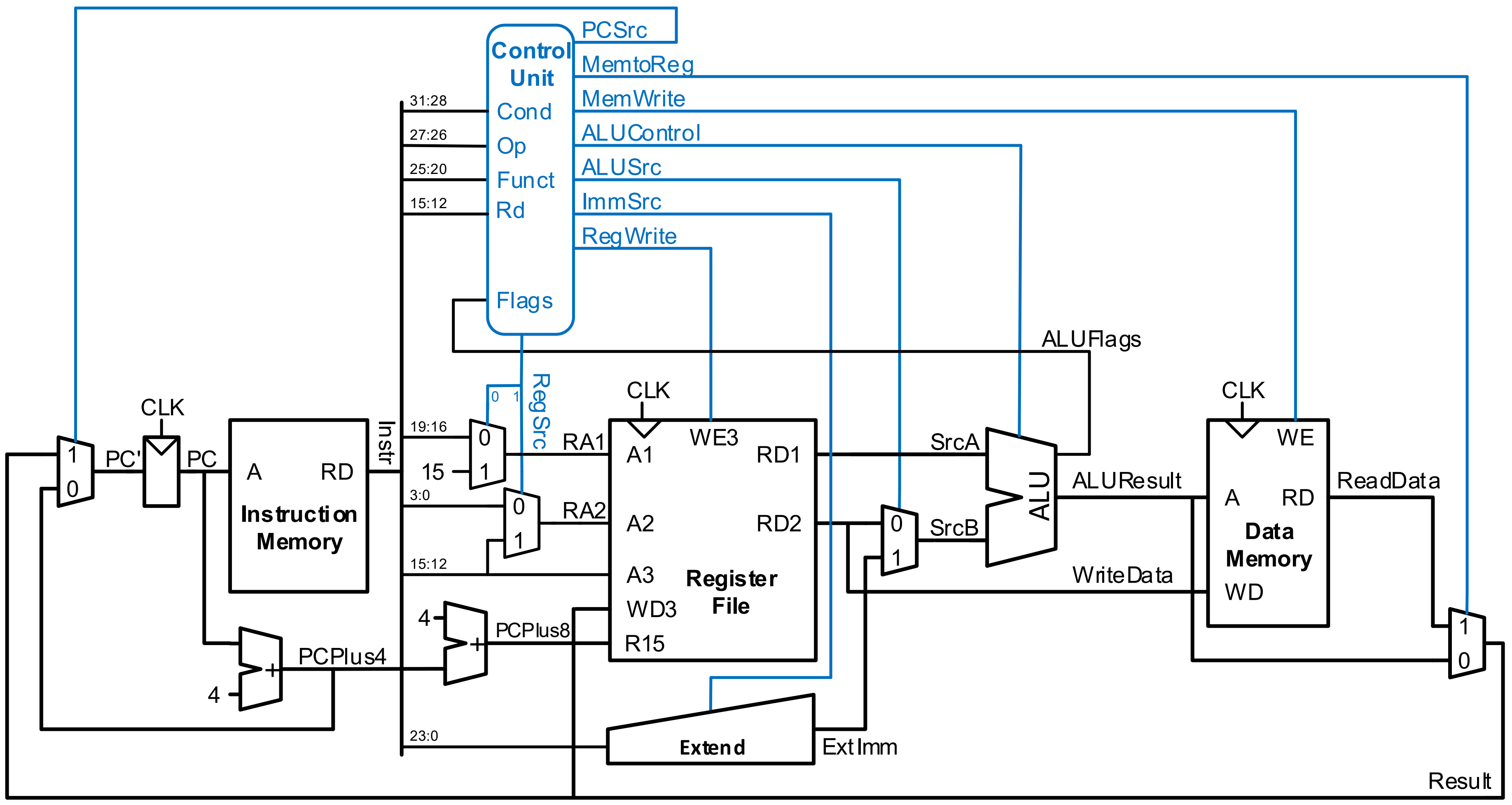

Figure 3. Single-cycle ARM processor