Gene Mary Cheruvathur and Ashley Guillard
EE 469
April 19, 2024

Lab 2 Report

Procedure

This lab was a continuation of the previous lab where we constructed an Arithmetic Logic Unit (ALU) and a Register File. These previous files were implemented into a partially finished ARM processor. With the addition of Lab 1's files, we were able to complete ARM Assembly commands such as MOV, ADD, SUB, AND, and ORR. The first task of this lab walked through how our ALU and register file fit into the design of the ARM processor. The second task had us add new instructions CMP (compare) and conditional branches.

Task #1 - Completing the Processor

To complete the first task for this lab, we needed to understand the connections between our ALU, register file, and the provided single-cycle ARM processor. Figure 1 shows how the register file (middle block) and the ALU (middle-right block) are integrated within the rest of the processor.

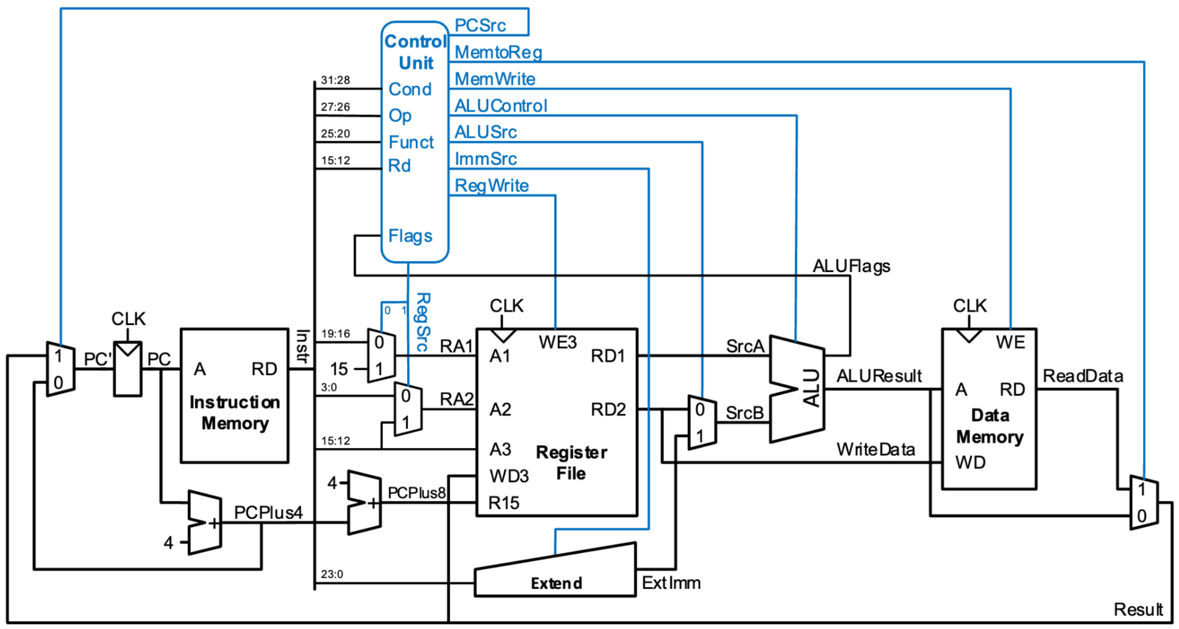

Figure 1: Single-cycle ARM Processor Schematic

As can be seen, a wire connects RA1 to the first address port of the register file and RA2 to the second read address port, whereas the write address is retrieved from bits 12 through 15 of the instruction since this holds what register we want the final result to go to. RA1 and RA2 are retrieved from a range of bits from the instruction, and the specific range is decided by what kind of instruction is being conducted. The write data connection is made to Result, which is either the read data from the data memory or the result of the ALU, decided by the type of instruction. Now the outputs of the register file connect to RD1 and RD2. The connection of RD1 goes directly into the SrcA of the ALU, and RD2 (depending on the instruction) goes to SrcB of ALU and WriteData of the data memory. For the ALU, the inputs are grabbed from SrcA and either the immediate value of the instruction or the second read data of the register file. The ALU will compute the results and input it to the data, and push the ALUFlags to the controller.

After connecting all the logics and ports, we needed to analyze and predict what we expected a set of assembly instructions, from the memfile.dat file, to output in the waveform of the simulation. To do this we used the tables from the lecture slides, which can be seen in Table 1, Table 2, Table 3, and Table 4 below.

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
11	X	X	B	1	0	0	1	10	0	X1	0

Table 1: The controller logics used for decisions in the datapath of the processor

Table 1 shows what controller logics are asserted depending on the assembly instruction. For example, if we needed to do some data-processing with an immediate, we would assert the ALUSrc to grab the immediate from the instruction and input it to the ALU instead of grabbing the data from the register file. The way to retrieve the immediate can be through the ImmSrc logic, which leads us to Table 2, where the first format of immediate needs to be grabbed since its specifically an immediate for data-processing.

$ImmSrc_{1:0}$	ExtImm	Description
00	{24'b0, $Instr_{7:0}$}	Zero-extended *imm8*
01	{20'b0, $Instr_{11:0}$}	Zero-extended *imm12*
10	{6{$Instr_{23}$}, $Instr_{23:0}$}	Sign-extended *imm24*

Table 2: The immediate data produced for the ExtImm depending on the type of instruction

We would also assert writing to the register file and set the RegSrc to X0, which will help decide which instruction bits we want to use to read from the register file. The table says the RegSrc is set to X0, where we, as the designer, get to decide what we want to set X to, it can be either 0 or 1. In the arm module, we set the X to be 0 since we don't ever use the data from the second read port of the register file. And finally, the ALUOp is asserted since we are doing some data-processing. When ALUOp is asserted, we will refer to Table 3, which connects to the Alu operations in Table 4.

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

Table 3: Instructions for the type of ALU operation to be conducted

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Table 4: ALU function in relation to the ALUControl

Like the example we stated, we follow the tables and the datapath diagram to predict all the outputs for the instructions set. The waveform of the processor is generated in ModelSIM, where we compare our predictions to the actual outputs.

Task #2 - Adding New Instructions

The second task for this lab was to introduce and implement comparative instructions (CMP) and conditional branching (BXX). To implement the conditional branching instructions, we needed to create a variable CondEx, which would determine if the instructions can be carried out or if the system would skip them. When CondEx is true, the instructions will be able to branch, write to a register, or write to memory. When CondEx is false, the instructions will be skipped and no changes will be made to the memory or registers. Table 5 shows the correlation between the XX mnemonic, within BXX, and the resulting conditional flags that would be required to carry out the branching instruction. This table was implemented within the code as a conditional case statement, see Appendix A.

$Cond_{3:0}$	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Table 5: Condition Mnemonics and CondEx value

The addition of CondEx would be meaningless if we could not store resulting flags from a compare (CMP) instruction. The purpose of the CMP instruction, for this lab, is to perform a subtraction and store the resulting flags. Since we are already familiar with how to conduct a subtraction and provide the flags with our ALU, the datapath section did not need to be adjusted for this new instruction. However, our controller needed to be updated to accommodate the CMP instructions. To start, we created a 4-bit register called FlagsReg. When the appropriate instruction is provided, the variable 2-bit FlagW will turn on and allow the previous flags to be replaced by current ones. The FlagW variable has the option to choose between storing the first two flags (NZ) with the first bit, and storing the second two flags (CV) with the second bit. Figure 2 shows this variable as well as the variable CondEx in more detail.

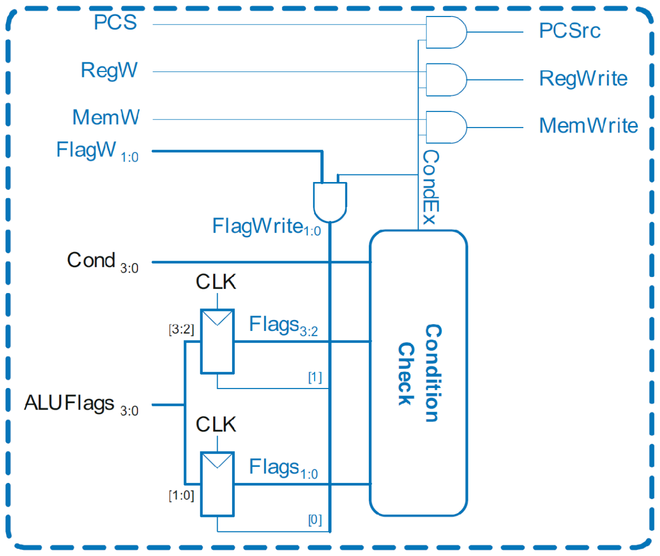

Figure 2: ARM Processor Controller with FlagW

With these adjustments to Task 1's ARM processor, we are able to handle conditional branching and compare instructions. This provides an opportunity to create if statements and loops, which could be very useful when creating a central processing unit (CPU). After the completion of Task 2, a table was created to understand order of the instructions and the values within each register, see Results Task #2 Table 7. These files were then simulated and compared to the table. The ModelSims and results will be explained later in this report.

Results

Task #1 - Completing the Processor

After writing down our predictions, we got the data in Table 6. The values with "0x" in-front indicate that they are written in hexadecimal format. The ones colored blue indicate they are data from the ALU and if it's colored green it comes from the data memory. The red X's are used to show that no data is currently stored or gathered.

Cycle	PC	Instr	SrcA	SrcB	ALUResult	WriteData	ReadData	MemWrite	RegWrite	Result
1	0x00	ADD R0, R15, #0	0x08	0x00	0x08	X	X	0	1	0x08
2	0x04	SUB R1, R0, R0	0x08	0x08	0x00	0x08	X	0	1	0x00
3	0x08	ADD R2, R1, #10	0x00	0x0A	0x0A	X	X	0	1	0x0A
4	0x0C	ADD R3, R0, R2	0x08	0x0A	0x12	0x0A	X	0	1	0x12
5	0x10	SUB R4, R2, #3	0x0A	0x03	0x07	0x12	X	0	1	0x07
6	0x14	SUB R5, R3, R4	0x12	0x07	0x0B	0x07	X	0	1	0x0B
7	0x18	ORR R6, R4, R5	0x07	0x0B	0x0F	0x0B	X	0	1	0x0F

8	0x1C	AND R7, R6, R5	0x0F	0x0B	0x0B	0x0B	X	0	1	0x0B
9	0x20	STR R7, [R1, #0]	0x00	0x00	0x00	0x0B	X	1	0	0x00
10	0x24	B SKIP	0x2C	0x04	0x30	0x00	X	0	0	0x30
11	0x30	LDR R8, [R1, #0]	0x00	0x00	0x00	X	0x0B	0	1	0x0B
12	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
13	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
14	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
15	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
16	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
17	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
18	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34
19	0x34	B LOOP	0x3C	-8	0x34	X	X	0	0	0x34

Table 6: First 19 cycles executing memfile.dat

The waveform of the ARM processor can be seen in Figure 3. In this waveform, we can see that the results are exactly the same as the predictions in Table 6. The SrcA, SrcB, WriteData, ReadData, etc., all update the expected clock cycle after the reset clock cycle. When we wrote the register file, we made it where the data will be stored the next clock cycle, which can be seen in the memory register file, the Result appears in the allocated register one clock cycle later. The only exception would be the first Result, but we believe this happened due to reset being asserted the previous clock cycle. The red lines for the WriteData and ReadData indicate that no data is present at whichever address the data is being grabbed from, proving our predictions accurate.

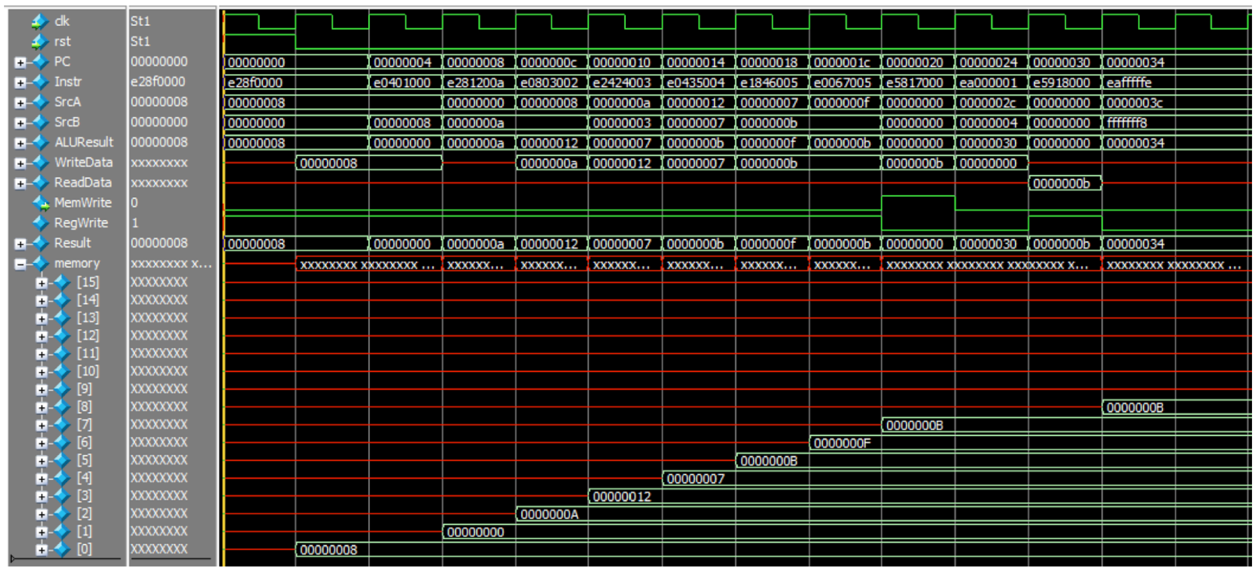

Figure 3: ModelSIM waveform of arm processor in Task 1

Task #2 - Adding New Instructions

Before simulating the ARM processor, we created a table of expected values which can be found below, Table 7. In the table, all values on the left side of the vertical border are in decimal format to make it easier to compare the expected values to the actual ones from ModelSim. The FlagsReg column is in binary to indicate which individual flags are on/off. The command column shows the current instructions that are read at each cycle. The Condition column shows if there was a condition and if the condition was false (skip the instructions) or true (execute the instructions). The PC sequence can be seen in the second column starting from 0 and ending at 116. Additionally, the final contents of each register can be seen in the final row of the table.

Cycle	PC	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R15	FlagsReg (NZCV)	Command	Condition
1	0	8	X	X	X	X	X	X	X	X	X	8	XXXX	ADD R0, R15, #0	ALWAYS
2	4	8	0	X	X	X	X	X	X	X	X	12	XXXX	SUB R1, R0, R0	ALWAYS
3	8	8	0	10	X	X	X	X	X	X	X	16	XXXX	ADD R2, R1, #10	ALWAYS
4	12	8	0	10	18	X	X	X	X	X	X	20	XXXX	ADD R3, R0, R2	ALWAYS
5	16	8	0	10	18	7	X	X	X	X	X	24	XXXX	SUB R4, R2, #3	ALWAYS
6	20	8	0	10	18	7	11	X	X	X	X	28	XXXX	SUB R5, R3, R4	ALWAYS
7	24	8	0	10	18	7	11	15	X	X	X	32	XXXX	ORR R6, R4, R5	ALWAYS
8	28	8	0	10	18	7	11	15	11	X	X	36	XXXX	AND R7, R6, R5	ALWAYS
9	32	8	0	10	18	7	11	15	11	X	X	40	XXXX	STR R7, [R1, #0]	ALWAYS
10	36	8	0	10	18	7	11	15	11	X	X	44	XXXX	B SKIP	ALWAYS
11	48	8	0	10	18	7	11	15	11	11	X	56	XXXX	LDR R8, [R1, #0]	ALWAYS
12	52	8	0	10	18	7	11	15	11	11	0	60	0110	CMP R9, R6, #15	ALWAYS
13	56	8	0	10	18	7	11	15	11	11	0	64	0110	BNE B_START	FALSE
14	60	8	0	10	18	7	11	15	11	11	4	68	0010	CMP R9, R5, R4	ALWAYS
15	64	8	0	10	18	7	11	15	11	11	4	72	0010	BNE BNETESTED	TRUE
16	72	8	0	10	18	7	11	15	11	11	-8	80	1000	CMP R9, R2, R3	ALWAYS
17	76	8	0	10	18	7	11	15	11	11	-8	84	1000	BGE BSTART	FALSE
18	80	8	0	10	18	7	11	15	11	11	8	88	0000	CMP R9, R3, R2	ALWAYS
19	84	8	0	10	18	7	11	15	11	11	8	92	0000	BGE BGETESTED	TRUE
20	92	8	0	10	18	7	11	15	11	11	8	100	0000	CMP R9, R3, R2	ALWAYS
21	96	8	0	10	18	7	11	15	11	11	8	104	0000	BLE BSTART	FALSE
22	100	8	0	10	18	7	11	15	11	11	-8	108	1000	CMP R9, R2, R3	ALWAYS

23	104	8	0	10	18	7	11	15	11	11	-8	112	1000	BLE BLETESTED	TRUE
24	112	8	0	10	18	7	11	15	11	1	-8	120	1000	ADD R8, R1, #1	ALWAYS
25	116	8	0	10	18	7	11	15	11	1	-8	124	1000	B LOOP	ALWAYS

Table 7: First 25 cycles executing memfile2.dat

After constructing the above table, Task 2's files were simulated, using a file labeled memfile.dat, to ensure the expected values were shown. The new instructions CMP and BXX were tested multiple times to ensure that each type of compared values and conditional branches were executed correctly. The new instructions start at cycle 12. The ModelSim transcript showed the correct output "Task 2 Passed". The below section will reference the ModelSim and table thoroughly.

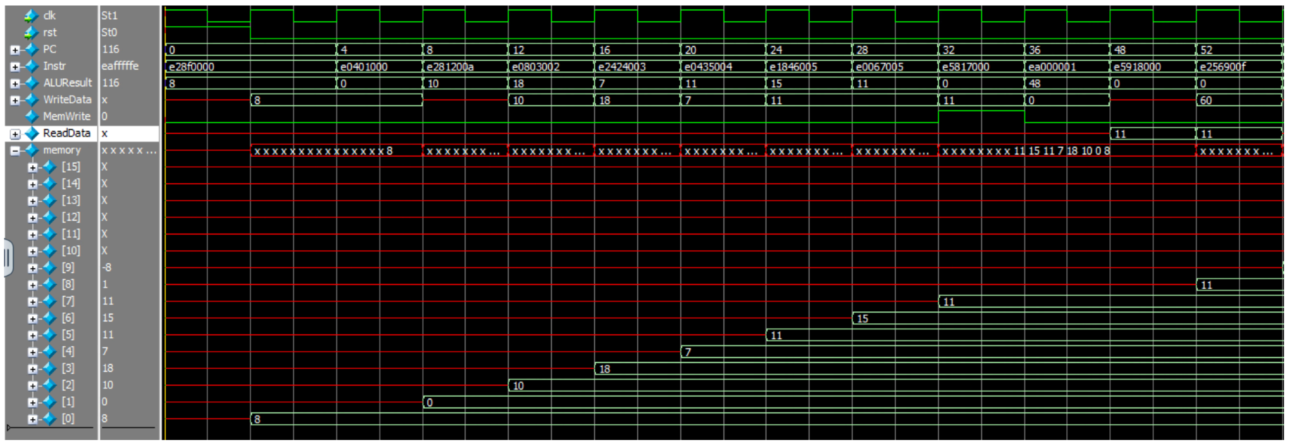

Figure 4: Cycles 0-12

Figure 4 shows the first 12 cycles including cycle 0 which refers to the reset cycle before any instructions have been read and is not shown in the table. Cycles 1-11 test the same functionalities as Task 1 to ensure that our processor can handle the previous and new instructions. The specific instructions can be found in the command column of Table 7. As can be seen, the ADD, SUB, ADD, ORR, and B (without condition) work as expected and all of the values match the table. All of the registers were updated when needed. The PC during this section increases by 4 (4-bytes per instruction) except for at PC=36 due to the "B SKIP" instructions.

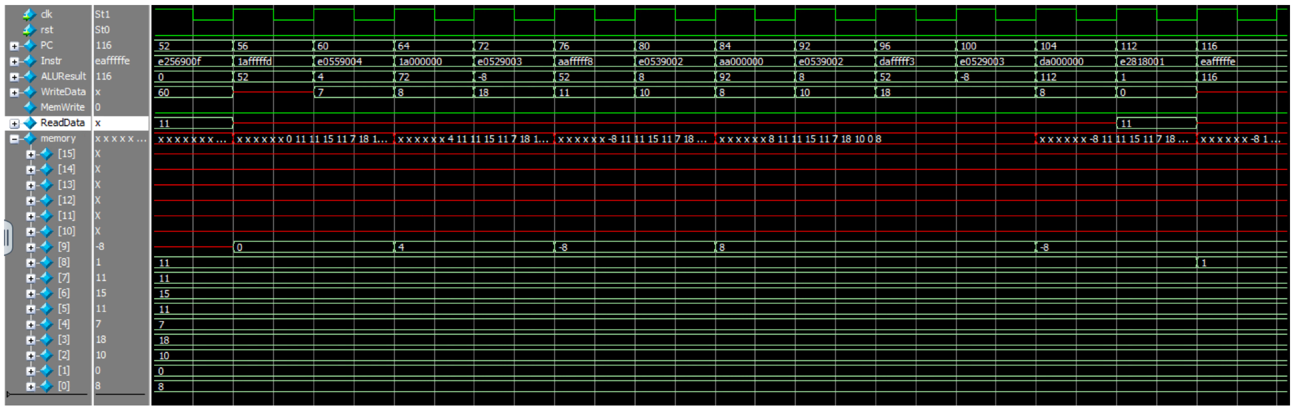

Figure 5: Cycles 12-25

The second section of the memfile2.dat can be seen in Figure 5. Starting at PC=52, we perform a comparison between R6 and the number 15. The sixth register holds the value 15 which means the stored flags would show that these numbers are equal, see Procedure Task #2 Table 5. Since the next instruction, at PC=56, is to branch if the compared values were not equal, this branching instruction is skipped and the PC the normal PCPlus4 (PC=60). Comparing the values and PC sequence in Table 7 to the ModelSim screenshot, we can see that the PC values only branch when the branching condition is met. This is the correct response for this system, and proves that both the compare and conditional branching instructions are working as they are supposed to. An additional test in cycle 24 shows that the last command before looping, "AND R8, R1 #1" executes and changes the final value stored in R8 at the end of the simulation. The final values shown in the waveform match those written in Table 7 above.

Final Product

For this lab, we were instructed to edit the necessary provided files to complete a simple ARM processor that could perform addition, subtraction, ANDing, ORing, branching (with or without conditions), and comparing. Only one of the given files needed to be edited to complete this lab, the arm.sv. This file held our ALU and reg_file as well as the controller we would need to update for conditional operations. When we completed the required revisions, two sets of instructions were provided to be tested, memfile.dat (Task 1) and memfile2.dat (Task 2). After simulating each task, the results were compared to a table of expected results and found to be accurate. In addition, the arm.sv file passed the provided test conditions to warrant the messages "Task 1 Passed" and "Task 2 Passed" in ModelSim's console. Overall, the files worked as expected and produced the correct outputs.

It was interesting to find that when implementing the CMP instructions, we were actually carrying out an instruction more similar to SUBS which is a subtraction with flags. This meant that we did not need to create a variable "NoWrite" as shown in one of the more recent lectures, 05b_single_cycle_controller. Additionally, after adding the flags to subtraction, it seemed fitting to add them for addition, ANDing, and ORing, when the instructions had S=1 in case they are needed in the future to perform operations like ANDS, see Appendix A. Since we only edited the arm.sv file, our appendix will only hold the files: arm.sv, reg_file.sv, alu.sv, adder.sv, and FullAdder.sv.

Appendix A - Processor

1) 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.
*/

// clk - system clock
// rst - system reset
// Instr - incoming 32 bit instruction from imem, contains opcode, condition,
addresses and or immediates
// ReadData - data read out of the dmem
// WriteData - data to be written to the dmem
// MemWrite - write enable to allowed WriteData to overwrite an existing dmem
word
// PC - the current program count value, goes to imem to fetch instruciton
// ALUResult - result of the ALU operation, sent as address to the dmem

module arm (
    input  logic        clk, rst,
    input  logic [31:0] Instr,
    input  logic [31:0] ReadData,
    output logic [31:0] WriteData,
    output logic [31:0] PC, ALUResult,
    output logic        MemWrite
);

    // datapath buses and signals
    logic [31:0] PCPrime, PCPlus4, PCPlus8; // pc signals
    logic [ 3:0] RA1, RA2;                  // regfile input addresses
    logic [31:0] RD1, RD2;                  // raw regfile outputs
    logic [ 3:0] ALUFlags;                  // alu combinational flag outputs
    logic [31:0] ExtImm, SrcA, SrcB;        // immediate and alu inputs
    logic [31:0] Result;                    // computed or fetched value to be
written into regfile or pc
       logic [ 3:0] FlagsReg;                  // stores the flags from the
most recent CMP command

    // control signals
    logic PCSrc, MemtoReg, ALUSrc, RegWrite, CondEx;
    logic [1:0] RegSrc, ImmSrc, ALUControl, FlagsWrite, FlagW;


    /* 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.
```

```
    */

//-----------------------------------------------------------------------------
---
    //                                          DATAPATH

//-----------------------------------------------------------------------------
---


    assign PCPrime = (PCSrc & CondEx) ? Result : PCPlus4;  // mux, use either
default or newly computed value
    assign PCPlus4 = PC + 'd4;                    // default value to access
next instruction
    assign PCPlus8 = PCPlus4 + 'd4;               // value read when reading
from reg[15]

    // update the PC, at rst initialize to 0
    always_ff @(posedge clk) begin
        if (rst) PC <= '0;
        else     PC <= PCPrime;
    end

    // determine the register addresses based on control signals
    // RegSrc[0] is set if doing a branch instruction
    // RefSrc[1] is set when doing memory instructions
    assign RA1 = RegSrc[0] ? 4'd15        : Instr[19:16];
    assign RA2 = RegSrc[1] ? Instr[15:12] : Instr[ 3: 0];

    // Register File (16x32)
    // Stores the data for registers 0 through 15.
       // 2 asynchronous read ports and 1 synchronous write port.
    reg_file u_reg_file (
        .clk       (clk),
        .wr_en     (RegWrite),
        .write_data(Result),
        .write_addr(Instr[15:12]),
        .read_addr1(RA1),
        .read_addr2(RA2),
        .read_data1(RD1),
        .read_data2(RD2)
    );

    // two muxes, put together into an always_comb for clarity
    // determines which set of instruction bits are used for the immediate
    always_comb begin
        if      (ImmSrc == 'b00) ExtImm = {{24{Instr[7]}},Instr[7:0]};
// 8 bit immediate - reg operations
```

```
        else if (ImmSrc == 'b01) ExtImm = {20'b0, Instr[11:0]};
// 12 bit immediate - mem operations
        else                        ExtImm = {{6{Instr[23]}}, Instr[23:0],
2'b00}; // 24 bit immediate - branch operation
    end

    // WriteData and SrcA are direct outputs of the register file, wheras SrcB
is chosen between reg file output and the immediate
    assign WriteData = (RA2 == 'd15) ? PCPlus8 : RD2;           // substitute
the 15th regfile register for PC
    assign SrcA      = (RA1 == 'd15) ? PCPlus8 : RD1;           // substitute
the 15th regfile register for PC
    assign SrcB      = ALUSrc        ? ExtImm  : WriteData;     // determine
alu operand to be either from reg file or from immediate

    // ALU (Arethmatic Logic Unit)
    // Preforms addition, subtraction, ANDing, as well as ORing.
       // Outputs result of the operation and flags (NZCV).
    alu u_alu (
        .a          (SrcA),
        .b          (SrcB),
        .ALUControl (ALUControl),
        .Result     (ALUResult),
        .ALUFlags   (ALUFlags)
    );

    // determine the result to run back to PC or the register file based on
whether we used a memory instruction
    assign Result = MemtoReg ? ReadData : ALUResult;    // determine whether
final writeback result is from dmemory or alu

       // DFF for storing flags from CMP command
       assign FlagsWrite = {2{CondEx}} & FlagW;
       always_ff @(posedge clk) begin
                  FlagsReg[3:2] <= FlagsWrite[1] ? ALUFlags[3:2] :
FlagsReg[3:2];
                  FlagsReg[1:0] <= FlagsWrite[0] ? ALUFlags[1:0] :
FlagsReg[1:0];
       end


    /* 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.
    */
```

```
//--------------------------------------------------------------------------------
---
    //                                      CONTROL

//--------------------------------------------------------------------------------
---

    always_comb begin
        casez (Instr[27:20])

            // ADD (Imm or Reg)
            8'b00?_0100_? : begin   // note that we use wildcard "?" in bit
25. That bit decides whether we use immediate or reg, but regardless we add
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = Instr[25]; // may use immediate
                RegWrite = CondEx;
                               FlagW = Instr[20] ? 2'b11 : 2'b00;
                RegSrc   = 'b00;
                ImmSrc   = 'b00;
                ALUControl = 'b00;
            end

            // SUB (Imm or Reg) OR CMP
            8'b00?_0010_? : begin   // note that we use wildcard "?" in bit
25. That bit decides whether we use immediate or reg, but regardless we sub
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = Instr[25]; // may use immediate
                RegWrite = CondEx;
                               FlagW = Instr[20] ? 2'b11 : 2'b00; // keep
flags (CMP), don't keep flags (SUB)
                RegSrc   = 'b00;
                ImmSrc   = 'b00;
                ALUControl = 'b01;
            end

            // AND
            8'b000_0000_? : begin
                PCSrc    = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc   = 0;
                RegWrite = CondEx;
                               FlagW = Instr[20] ? 2'b10 : 2'b00;
                RegSrc   = 'b00;
                ImmSrc    = 'b00;     // doesn't matter
                ALUControl = 'b10;
            end

            // ORR
            8'b000_1100_? : begin
                PCSrc     = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc    = 0;
                RegWrite = CondEx;
                               FlagW = Instr[20] ? 2'b10 : 2'b00;
                RegSrc    = 'b00;
                ImmSrc    = 'b00;     // doesn't matter
                ALUControl = 'b11;
            end

            // LDR
            8'b010_1100_1 : begin
                PCSrc     = 0;
                MemtoReg = 1;
                MemWrite = 0;
                ALUSrc    = 1;
                RegWrite = CondEx;
                               FlagW = 2'b00;
                RegSrc    = 'b10;     // msb doesn't matter
                ImmSrc    = 'b01;
                ALUControl = 'b00;   // do an add
            end

            // STR
            8'b010_1100_0 : begin
                PCSrc     = 0;
                MemtoReg = 0; // doesn't matter
                MemWrite = CondEx ? 1 : 0;
                ALUSrc    = 1;
                RegWrite = 0;
                               FlagW = 2'b00;
                RegSrc    = 'b10;     // msb doesn't matter
                ImmSrc    = 'b01;
                ALUControl = 'b00;   // do an add
            end

            // B
            8'b1010_???? : begin
                    PCSrc     = CondEx;
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc    = 1;
```

```
                    RegWrite = 0;
                                      FlagW = 2'b00;
                    RegSrc   = 'b01;
                    ImmSrc   = 'b10;
                    ALUControl = 'b00;  // do an add
            end

                  default: begin
                        PCSrc    = 0;
                 MemtoReg = 0; // doesn't matter
                 MemWrite = 0;
                 ALUSrc   = 0;
                 RegWrite = 0;
                        FlagW = 2'b00;
                 RegSrc   = 'b00;
                 ImmSrc   = 'b00;
                 ALUControl = 'b00;  // do an add
                  end
          endcase
      end

        // Condition Check
        always_comb begin
            case (Instr[31:28])

                  4'b0000: CondEx = FlagsReg[2];
// EQ (Equal)
                  4'b0001: CondEx = ~FlagsReg[2];
// NE (Not equal)
                  4'b0010: CondEx = FlagsReg[1];
// CS / HS (Carry set / Unsigned higher or same)
                  4'b0011: CondEx = ~FlagsReg[1];
// CC / LO (Carry clear / Unsigned lower)
                  4'b0100: CondEx = FlagsReg[3];
// MI (Minus / Negative)
                  4'b0101: CondEx = ~FlagsReg[3];
// PL (Plus / Positive pf Zero)
                  4'b0110: CondEx = FlagsReg[0];
// VS (Overflow / Overflow set)
                  4'b0111: CondEx = ~FlagsReg[0];
// VC (No overflow / Overflow clear)
                  4'b1000: CondEx = ~FlagsReg[2] & FlagsReg[1];
// HI (Unsigned higher)
                  4'b1001: CondEx = FlagsReg[2] | ~FlagsReg[1];
// LS (Unsigned lower or same)
                  4'b1010: CondEx = ~(FlagsReg[3] ^ FlagsReg[0]);
// GE (Signed greater than or equal)
                  4'b1011: CondEx = FlagsReg[3] ^ FlagsReg[0];
// LT (Signed less than)
```

```
                    4'b1100: CondEx = ~FlagsReg[2] & ~(FlagsReg[3] ^
FlagsReg[0]); // GT (Signed greater than)
                    4'b1101: CondEx = FlagsReg[2] | (FlagsReg[3] ^ FlagsReg[0]);
// LE (Signed less than or equal)
                    4'b1110: CondEx = 1;
// AL / none (Always / unconditional)
                    default: CondEx = 0;

            endcase
       end


endmodule
```

Appendix B - Register File

1) reg_file.sv

```
//This module is created for Lab 1 Task 2, where we design a 16x32
//register file with 2 asynchronous read ports and 1 synchronous
//write port. The inputs are the write and two read addresses,
//the write data and if write is enabled. The outputs are the data
//at the specified read addresses.
//
//Inputs: clk, wr_en, write_data, write_addr, read_addr1, read_addr2
//Outputs: read_data1, read_data2
module reg_file(clk, wr_en, write_data, write_addr, read_addr1,
                                          read_addr2, read_data1, read_data2);
      input logic clk, wr_en;
      input logic [31:0] write_data;
      input logic [3:0] write_addr;
      input logic [3:0] read_addr1, read_addr2;
      output logic [31:0] read_data1, read_data2;

      //16x32 registers holding data as memory
      logic [15:0][31:0] memory;

      //writes data to specified address on memory at positive edge of
      //clock cycles (synchronous)
      always_ff @(posedge clk) begin
            //only write if wr_en is true
            if (wr_en) begin
                  memory[write_addr] <= write_data;
            end
      end

      //gives the data immediately when both read data is specified (asynch)
      assign read_data1 = memory[read_addr1];
      assign read_data2 = memory[read_addr2];
endmodule
```

Appendix C - ALU

1) FullAdder.sv

```
// Adds three 1-bit inputs, A, B, and cin. Produces the sum and carry out.
//
// Inputs: A, B, cin (carry in)
// Outputs: sum, cout (carry out)
module FullAdder (A, B, cin, sum, cout);

      input logic A, B, cin;
      output logic sum, cout;

      assign sum = A ^ B ^ cin;
      assign cout = A&B | cin & (A^B);

endmodule
```

2) adder.sv

```
// Adds two N-bit numbers and produces the output and carry out.
// Generates an instantiations of the FullAdder module for each
// of the N bits. The initial carry in is used for the first
// Full Adder.
//
// Inputs: a, b, cin (initial carry in)
// Outputs: result, cout0 (final carry out)
module adder #(parameter N = 3) (
      input logic [N-1:0] a, b,
      input logic cin,
      output logic [N-1:0] result,
      output logic cout0
      );

      // cin and cout
      wire [N-1:0]cout;
      assign cout0 = cout[N-1];

      genvar i;
      generate
            for(i=0; i<N; i++) begin : Adders
                  if (i==0) FullAdder FA0 (.A(a[i]), .B(b[i]), .cin(cin),
.sum(result[i]), .cout(cout[i]));
                  else FullAdder FA (.A(a[i]), .B(b[i]), .cin(cout[i-1]),
.sum(result[i]), .cout(cout[i]));
            end
      endgenerate
```

```
endmodule
```

3) alu.sv

```
// This module holds the ALU for lab 1 Task 3 which operates on two N-bit
inputs.
// This ALU is able to change operations based on the 2-bit "ALUControl"
input. The
// possible operations are: Addition (00), Subtraction (01), AND (10), as well
as
// OR (11). Based on the given input, this module will output the correct
result of
// the selected operation. This module additionally outputs flags (within the
"ALUFlags"
// variable) when the result is negative (ALUFlags[3]) and result is zero
([2]). When
// adding or subtracting, the ALUFlags[1] and ALUFlags[0] will show whether
the adder
// produced a carry out or experienced an overflow, respectively.
//
// Inputs: a, b, ALUControl
// Outputs: Result, ALUFlags
module alu #(parameter N = 32)(

      // Inputs a and b (N-bit)
      input logic [N-1:0] a, b,

      // Controls the operation
      input logic [1:0] ALUControl,

      // Output result from operation
      output logic [N-1:0] Result,

      // ALU flags:
      // [0] = Adder results in overflow
      // [1] = Adder produced a carry out
      // [2] = Result is 0
      // [3] = Result is negative
      output logic [3:0] ALUFlags
      );

      // Results from each operation
      logic [N-1:0] sum, AND, OR, b_new;

      // Cout for adder
      logic cout;
```

```
	// add/sub b value MUX
	always_comb begin
		if (ALUControl[0]) b_new = ~b; // If subtracting, use ~b
		else b_new = b;                 // If adding, use b
	end

	// Instantiate add module
	// Uses Full Adders to add each individual bits together. Produces a
Cout value.
	adder #(.N(N)) adding (.a, .b(b_new), .cin(ALUControl[0]), .result(sum),
.cout0(cout));

	// Store result of ANDing
	assign AND = a & b;

	// Store result from ORing
	assign OR = a | b;

	// Ending result MUX
	always_comb begin
		case(ALUControl)
			2'b10: Result = AND;
			2'b11: Result = OR;
			default: Result = sum;
		endcase
	end

	// ALU Flags output circuit
	assign ALUFlags[0] = ~( (ALUControl[0]) ^ (a[N-1]) ^ (b[N-1]) ) & (
(a[N-1]) ^ (sum[N-1]) ) & ~( ALUControl[1] );
	assign ALUFlags[1] = ~ALUControl[1] & cout;
	assign ALUFlags[2] = (Result == 0);
	assign ALUFlags[3] = Result[N-1];

endmodule
```