Lab 5: Microprocessor System Design

ARM Single-Cycle Processor

Miriam Mnyuku

June 04, 2018

Abstract
The aim of this lab was to build a simplified ARM single-cycle processor using SystemVerilog. We built up from lab 2 by adding the ALU code and other codes were provided from the textbook. A test program was created to test two new function that were added LDRB and EOR.

Introduction
A single-cycle processor module was added to the code from lab 2. It was then tested to confirm how the functions of the whole system works. The single-cycle module used divides the circuit schematic and the machine into two major units: the control and the data path. Each unit is constructed from various functional blocks, such as the datapath containing the 32-bit ALU created in lab 2, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

Materials
Quartus II version 15.1
ModelSim

Procedure
Part 1: The modules we studied to gain understanding of how the arm processor instruction and datapaths work.

Part 2: the single-cycle ARM processor was tested by using assembly language which was converted into machine language. The machine language written in hexadecimal was created and saved in a .dat file which was stored in the simulation and main folder of the lab 5 project library. To create the .dat file, first create a new text file, enter the machine language into the text file and then re-save the file so that it is named “memfile.dat.”
The table 1 below was filled out by guessing the results before confirming them by running the simulation. The results were compared after the simulation for verification.

Part 3: The ARM single-cycle processor was modified by adding the LDRB and EOR instructions. To accomodate for the EOR instruction, the ALU was revised to XOR datapath values ‘SrcA’ and ‘SrcB’. The circuit schematic for the single-cycle ARM processor was not revised other than changing the bus wires for the ALUControl path. The ARM ALU, however, was revised by increasing the 4-case multiplexer, to a 5-case, which required increasing to ALUControl bus wire to allow 3’bits. In order to revise the ALU to account for the XOR case statement, the ALUControl was increased to three bits (ALUControl[2:0]) and an additional case statement was created for the following bit combination (3’b100) for EOR case and ‘SrcA’ XOR ‘SrcB’.

```
always_comb
    begin
        case(ALUControl[2:0])
            3'b000: Result = sum;       //ALUControl = 0; add
            3'b001: Result = sum;       //ALUControl = 1; subtract
            3'b010: Result = a & b; //ALUControl = 2; and
            3'b011: Result = a | b; //ALUControl = 3; or
            3'b100: Result = a ^ b;     //ALUControl = 4; XOR
            default: Result = 32'bx;    //default
        endcase
    end
```

The extended functionality table for the ALUDecoder was then revised to the following:

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	000	00
1	0100	0	ADD	000	00
		1			11
	0010	0	SUB	001	00
		1			11
	0000	0	AND	010	00
		1			10
	1100	0	ORR	011	00
		1			10
1	1000	1	EOR	100	00
		0			10

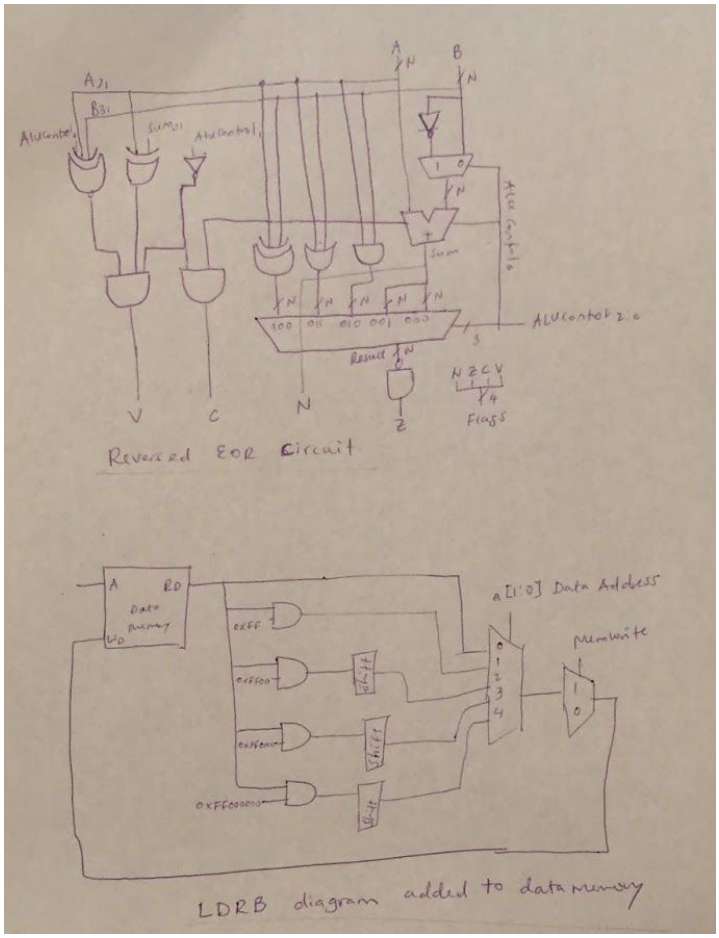


Table 2. Extended functionality: Main Decoder

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

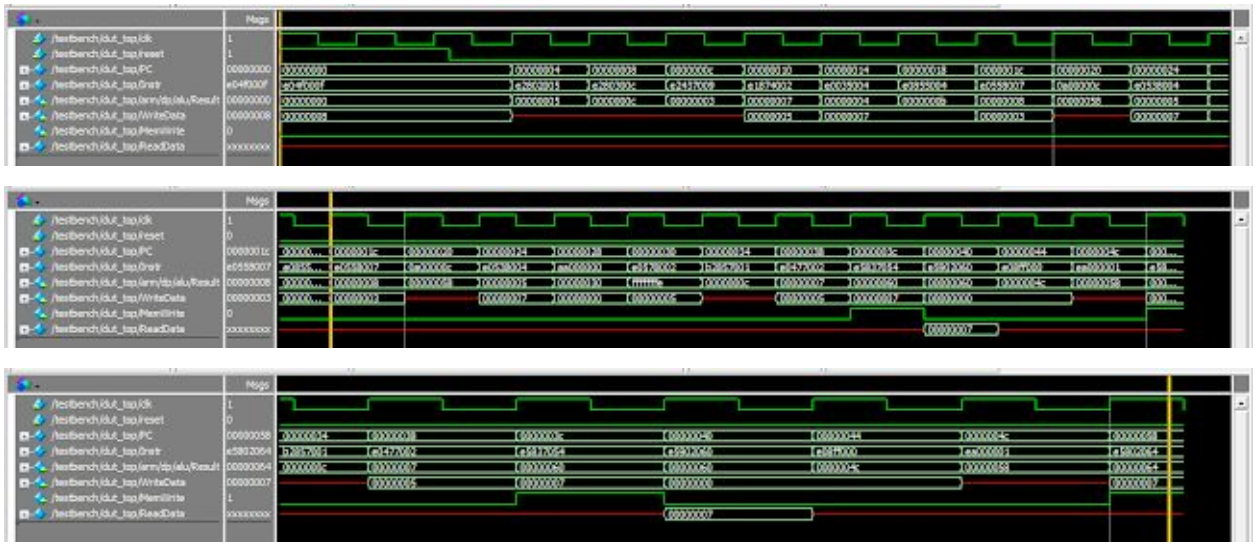
Part 4: Next, a second test program will be created to verify that the modified processor successfully works. The program should check that the instructions work properly and that the old ones did not break. The program written in assembly language was converted into the following machine language and stored in a file named memfile2.dat which was stored in the same location as the first test program. The program was run multiple times and results were verified.

```
E04F000F
E28010FF
E0812001
E58020C4
E221304D
E203401F
E0835004
E5D56000
E5D57001
E0560007
BAFFFFF4
CA000001
E584106E
EAFFFFF1
E584606E
```

Results

Part 2: tested the single-cycle ARM processor by using assembly language that was converted into machine language.

The following waveforms were outputted after testing the single-cycle ARM processor using the ALU developed in lab 2:



The final value that was stored by the last instruction was 0x07, based on the datapath label WriteData.
The primary issue that I kept running into was not correctly changing the port busses for each declaration of ALUControl to accommodate for 3 bits. I spent a good chunk of my time stuck, until I determined that I had actually just missed on pin assignment, which was stuck at [1:0] instead of [2:0].

This code was added to part 2:

Arm_single.sv code

```
// ALU Decoder
always_comb
  if (ALUOp) begin                 // which DP Instr?
    case(Funct[4:1])  // modified to 3 bits
     4'b0100: ALUControl = 3'b000; // ADD
     4'b0010: ALUControl = 3'b001; // SUB
      4'b0000: ALUControl = 3'b010; // AND
     4'b1100: ALUControl = 3'b011; // ORR
     4'b0001: ALUControl = 3'b100; // XOR              //modificaion for EOR
     default: ALUControl = 3'bx;  // unimplemented
    endcase
    // update flags if S bit is set
  // (C & V only updated for arith instructions)
    FlagW[1]      = Funct[0]; // FlagW[1] = S-bit
  // FlagW[0] = S-bit & (ADD | SUB)
    FlagW[0]      = Funct[0] &
      (ALUControl == 3'b000 | ALUControl == 3'b001);      //modified for EOR
  end else begin
    ALUControl = 3'b000; // add for non-DP instructions      // modified for EOR
```

ALU.sv code

```
always_comb
   begin
      case(ALUControl[2:0])
         3'b000: Result = sum;       //ALUControl = 0; add
         3'b001: Result = sum;       //ALUControl = 1; subtract
         3'b010: Result = a & b; //ALUControl = 2; and
         3'b011: Result = a | b; //ALUControl = 3; or
         3'b100: Result = a ^ b;     //ALUControl = 4; XOR
         default: Result = 32'bx;    //default
      endcase
   end
```

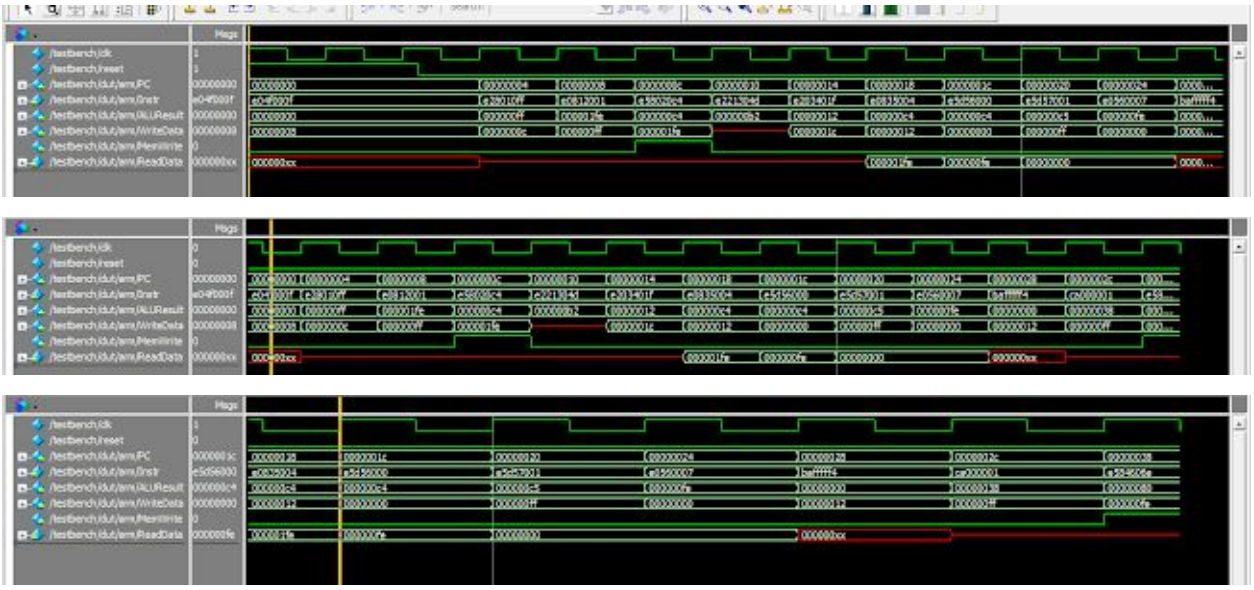
2. Table 1: First nineteen cycles of executing armtest.asm

Cycle	Reset	PC	Instr	SrcA	SrcB	Branch	ALUResult	Flags [NZCV]	CondEx	WriteData	MemWrite	ReadData
1	1	0	SUB R0, R15, R15 E04F000F	8	8	0	0	0110	1	8	0	x
2	0	4	ADD R2, R0, #5 E2802005	0	5	0	5	0000	1	x	0	x
3	0	8	ADD R3, R0 #12 E280300C	0	C	0	C	0000	1	x	0	x
4	0	C	SUB R7, R3, #9 E2437009	C	9	0	3	0010	1	x	0	x
5	0	10	ORR R4, R7, R2 E1874002	3	5	0	7	0000	1	5	0	x
6	0	14	AND R5, R5, R4 E0035004	C	7	0	4	0000	1	7	0	X
7	0	18	ADD R5, R5, R4 E0855004	4	7	0	B	0000	1	7	0	X
8	0	1C	SUBS R8, R5, R7 E0558007	B	3	0	8	0010	1	3	0	X
9	0	20	BEQ END 0A00000C	28	30	1	58	0000	0	X	0	X
10	0	24	SUBS R8, R3, R4 E0538004	C	7	0	5	0010	1	7	0	X
11	0	28	BGE AROUND AA000000	30	0	1	30	0000	1	0	0	X
12	0		ADD R5, R0, #0 E2805000 (SKIPPED)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
13	0	30	SUBS R8, R7, R2 E0578002	3	5	0	FFFFFFFE	1000	5	1	0	X
14	0	34	ADDLT R7, R5, #1 B2857001	B	1	0	C	0000	1	X	0	X
15	0	38	SUB R7, R7, R2 E0477002	C	5	0	7	0010	1	5	0	X
16	0	3C	STR R7, [R3, #84] E5837054	C	54	0	60	0000	1	7	1	X
17	0	40	LDR R2, [R0, #96] E5902060	0	60	0	60	0000	1	0	0	7
18	0	44	ADD R15, R15, R0 E08FF000	4C	0	0	4C	0000	1	0	0	X
19	0		ADD R2, R0, #14 (SKIPPED)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
20	0	48	B END EA000001	54	4	1	58	0000	1	X	0	X
21	0	4C	ADD R2, R0, #10 (SKIPPED)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
22	0		ADD R2, R0, #13 (SKIPPED)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
23		58	STR R2, [R0, #100] E5802064	0	64	0	64	0000	1	7	1	X

3. An image of the simulation waveforms showing correct operation of the processor. Does it write the correct value to address 100? Yes.

Part 3: involved modifying the ARM single-cycle processor by adding the EOR and LDRB instructions.

Waveforms results after testing the single-cycle ARM processor using the ALU developed in lab 2:



The previous SystemVerilog program was added and modified to:

```
  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
      //if(DataAdr === 100 & writeData === 7) begin
      // modified for second instruction set which includes LDRB
        if(DataAdr === 'h80 & WriteData === 'hFE) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 'hC4) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] PC, Instr, ReadData;

  // instantiate processor and memories
  arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
          WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData, Instr[22]); //add B: modification for LDRB
endmodule

module dmem(input  logic        clk, we,
            input  logic [31:0] a, wd,
            output logic [31:0] rd,
            input logic B);      // add B

  logic [31:0] RAM[63:0];


  //modification for LDRB
  always_comb begin
  rd = RAM[a[31:2]]; // word aligned

  //LDRB logic: if B or Instr[22] (Funct[2]) is 1, then use the multiplexer before entering modified dmem
     if(B == 1)
        case(a[1:0])
```

```
  //modification for LDRB
  always_comb begin
  rd = RAM[a[31:2]]; // word aligned

  //LDRB logic: if B or Instr[22] (Funct[2]) is 1, then use the multiplexer before entering modified dmem
     if(B == 1)
        case(a[1:0])
          2'b00: rd = rd & 8'hFF;
          2'b01: rd = (rd & 8'hFF00);       // >> 8;
          2'b10: rd = (rd & 8'hFF0000)      >> 16;
          2'b11: rd = (rd & 8'hFF000000)    >> 24;
        endcase
  end
```

Shawnna helped implement this logic. The module was added after the data memory and in between the ‘ReadData’ wire bus and the multiplexer connected to ‘MemtoReg’.

The final STR instruction address and value are:
Address: 0xC4
Data Value: 0xFE

Conclusion:
This lab helped me understand how instructions are executed in the single-cycle ARM processor and it also helped me improve my debugging and system verilog skills.

Bibliography:
Shawnna
Instructor Joe
Textbook