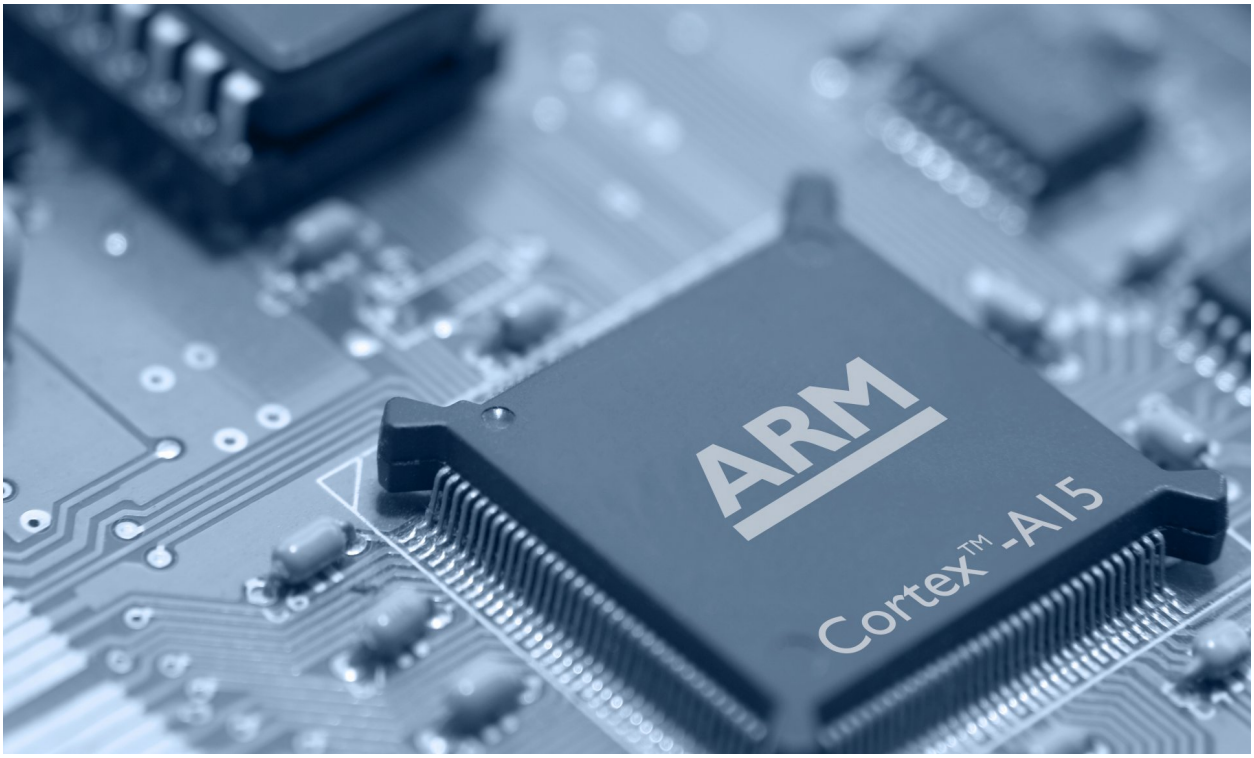
LAB 2 ARM Single-Cycle Processor

ECS154A

Zifei Cheng

May 28th 2022



Introduction

In this lab, we would like to implement a simplified ARM single-cycle processor by adding new instructions to the source codes in System Verilog. By critical thinking, we split this lab into 6 parts. To add new instructions, we first consider what we need to modify in our existing draft ARM single-cycle processor. Recall what we learned in class, a single-cycle ARM processor divides into two interacting parts: the Datapath including the ALU we built from LAB1, and the Control Unit computes by Decoder and Conditional Logic. Under the basic understanding of a limited subset (AND, SUB, AND, ORR) of ARM instruction sets in our source codes, we start to add XOR, LDRB, CMP, and LSR/LSL for data processing and the addressing modes in which the second source is either a register or an immediate. Thus, we first need to look through the source codes and make sure we're familiar with how the units perform in System Verilog. Second, we follow the LAB manual to add modifications including the ARM table and ARM processor schematic directly by hand. Third, after modifying any units in Datapath or the Control Unit as required, we start to make changes in System Verilog following our designed ARM table and schematic. Finally, by considering the correctness of our modified ARM processor, we make testbenches to check whether our new instructions work properly. At the end of this lab, we have a better understanding of how the ARM single-cycle processor produces and performs in a more straightforward way.

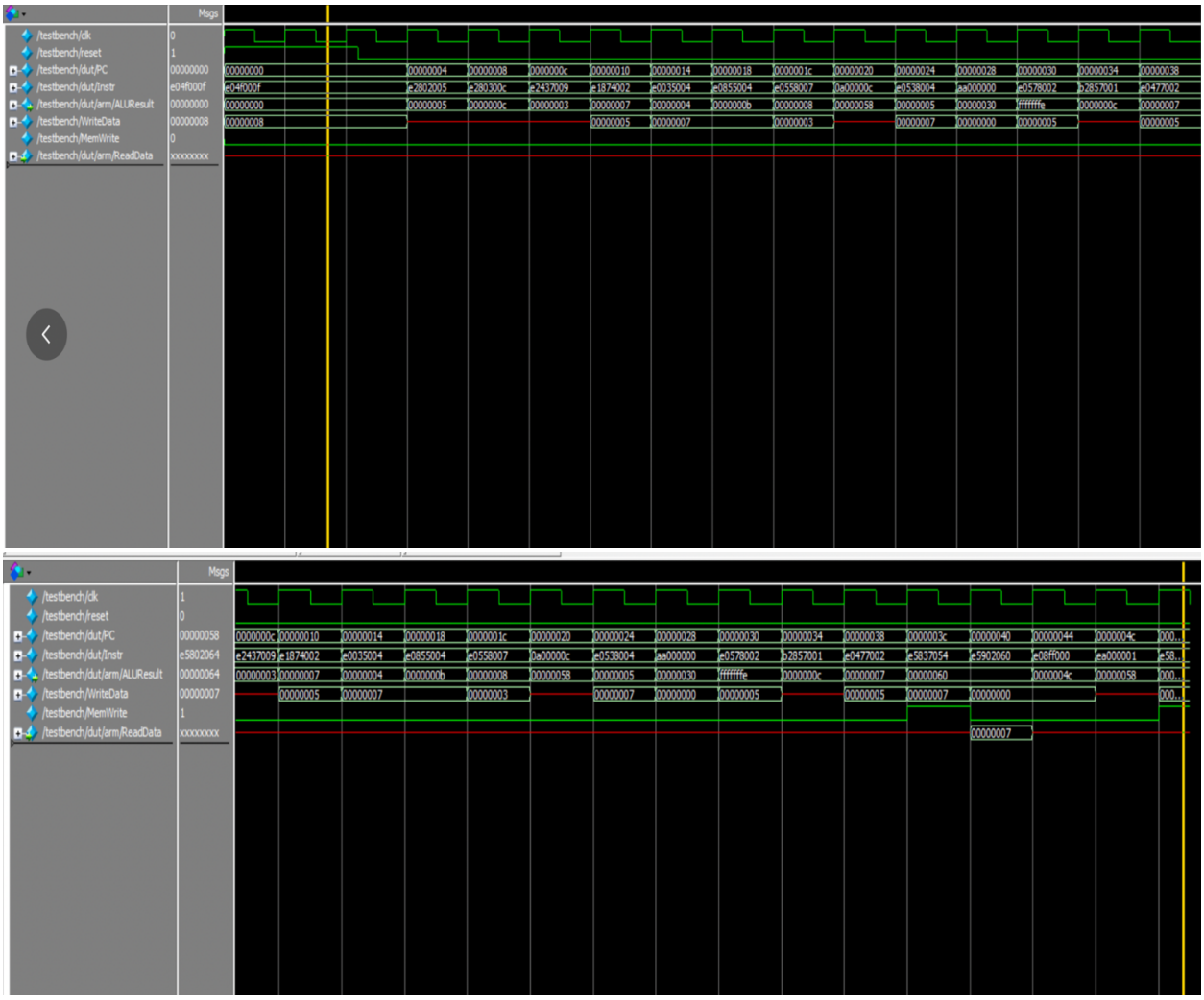
Part 1 & Part 2

ARM Single-Cycle Processor & 2 Testing the single-cycle ARM processor：
In part1, we look through our source code of the single-cycle ARM processor and try to understand why the modules are there and how each module works together. After fully understanding the entire system Verilog version of our simple single-cycle ARM processor, we start to add our ALU from lab1. In part2, we already have the basic single-cycle ARM processor that we need to modify later and now it is time to compile and simulate our source code.
To answer the question "What address, and data values are written by the final STR instruction?"， we fill out Table 1 by hand decoding and perform each instruction by hand. Then, we can know that the address is 128 and the data value is 254.
(a)
Table 1

Cycle	reset	PC	Instr	Src A	SrcB	Branch	AluResult
1	1	0	SUB R0, R15, R15 E04F000F	8	8	0	0
2	0	4	ADD R2, R0, #5 E2802005	0	5	0	5
3	0	8	ADD R3, R0, #12 E280300C	0	C	0	C
4	0	0C	SUB R7, R3, #9 E2437009	C	9	0	3
5	0	10	ORR R4, R7, R2 E1874002	3	5	0	7
6	0	14	AND R5, R3, R4 E0035004	C	7	0	4
7	0	18	ADD R5, R5, R4 E0855004	4	7	0	B
8	0	1C	SUBS R8, R5, R7 E0558007	B	3	0	8
9	0	20	BEQ END 0A00000C	28	30	1	58
10	0	24	SUBS R8, R3, R4 E0538004	C	7	0	5
11	0	28	BGE AROUND AA000000	30	0	1	30
skip	0	2C	E2805000				
12	0	30	SUBS R8, R7, R2 E0578002	3	5	0	FFFFFFFE
13	0	34	ADDLT R7, R5, #1 B2857001	B	1	0	C
14	0	38	SUB R7, R7, R2 E0477002	C	5	0	7
15	0	3C	STR R7, [R3, #84] E5837054	C	54	0	60
16	0	40	LDR R2, [R0, #96] E5902060	0	60	0	60
17	0	44	ADD R15, R15, R0 E08FF000	4C	0	0	4C
not execute	0	48	E280200E				
18	0	4C	B END EA000001	54	4	1	58
not execute	0	50	E280200D				

not execute	0	54	E280200A				
19	0	58	STR R2, [R0, #100] E5802064	0	64	0	64

(b)
Waveform



Part 3

Modifying the ARM single-cycle processor：

Now in part3, we start to modify our Arm processor by implementing two new instructions EOR and LDRB. When adding any instruction, we first consider where and how we need to modify our processor.

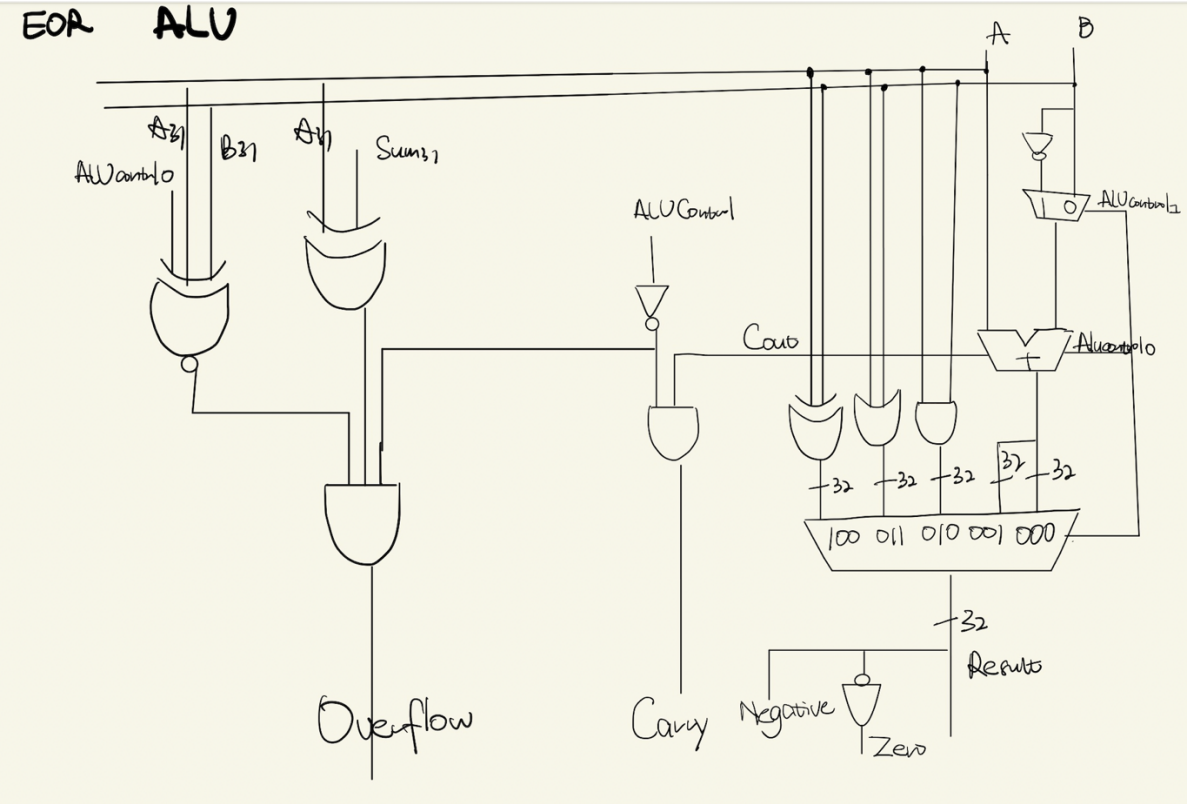
(c) EOR and LODR

1. EOR

First, we implement EOR, and we know that EOR has the same logic as xor gates, so we don't need to make other changes except for our ALU. To do so, we first add a new instruction to our ALU and modify ALU to perform XOR. And modifying the ALU truth table can help us to decode EOR performance in System Verilog.

Datapath won't change.

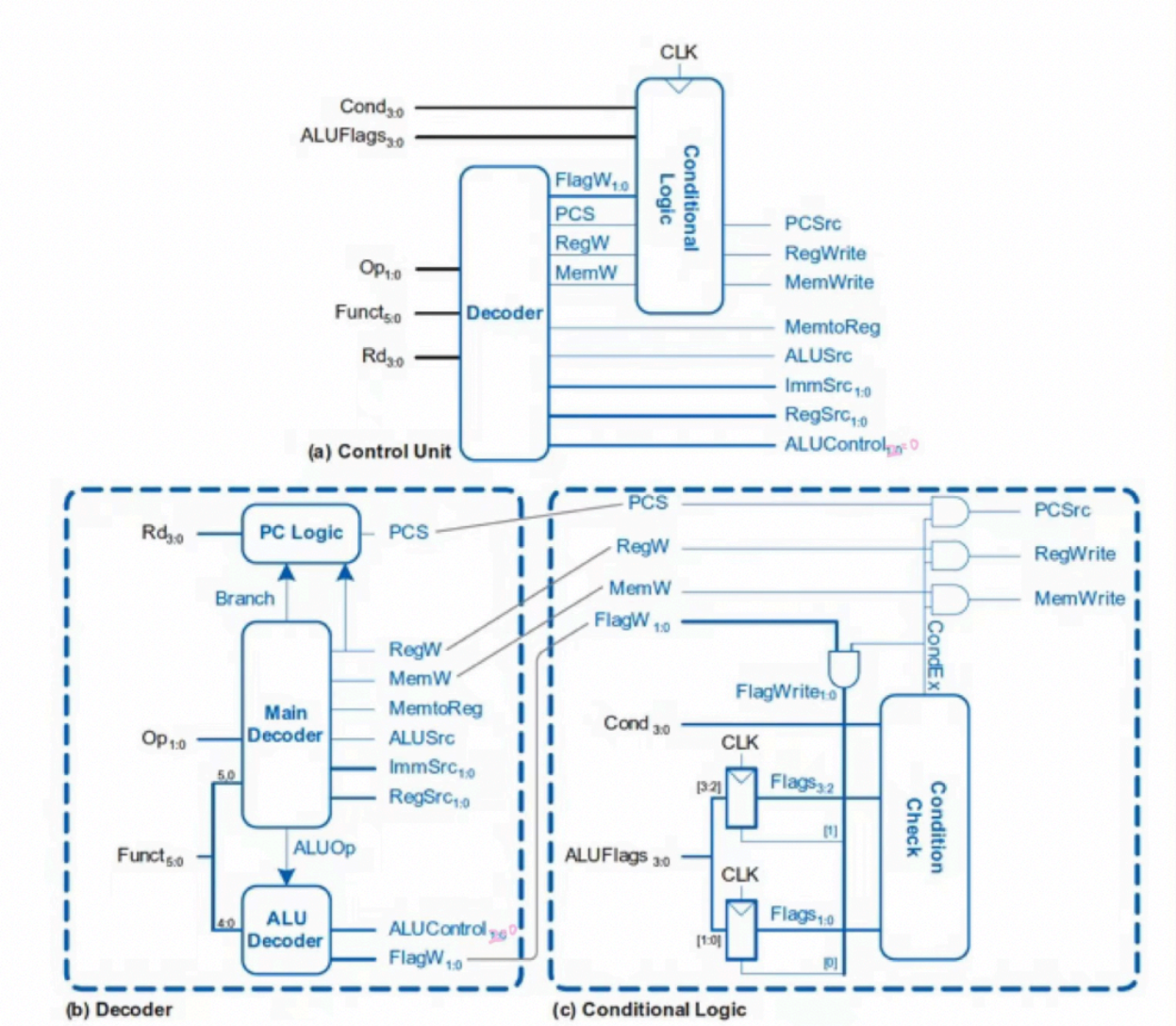
ALU for EOR



ALU Decoder truth table(main decoder table stay the same)

$ALUOp$	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10
	0001	0	EOR	11	00
		1			10

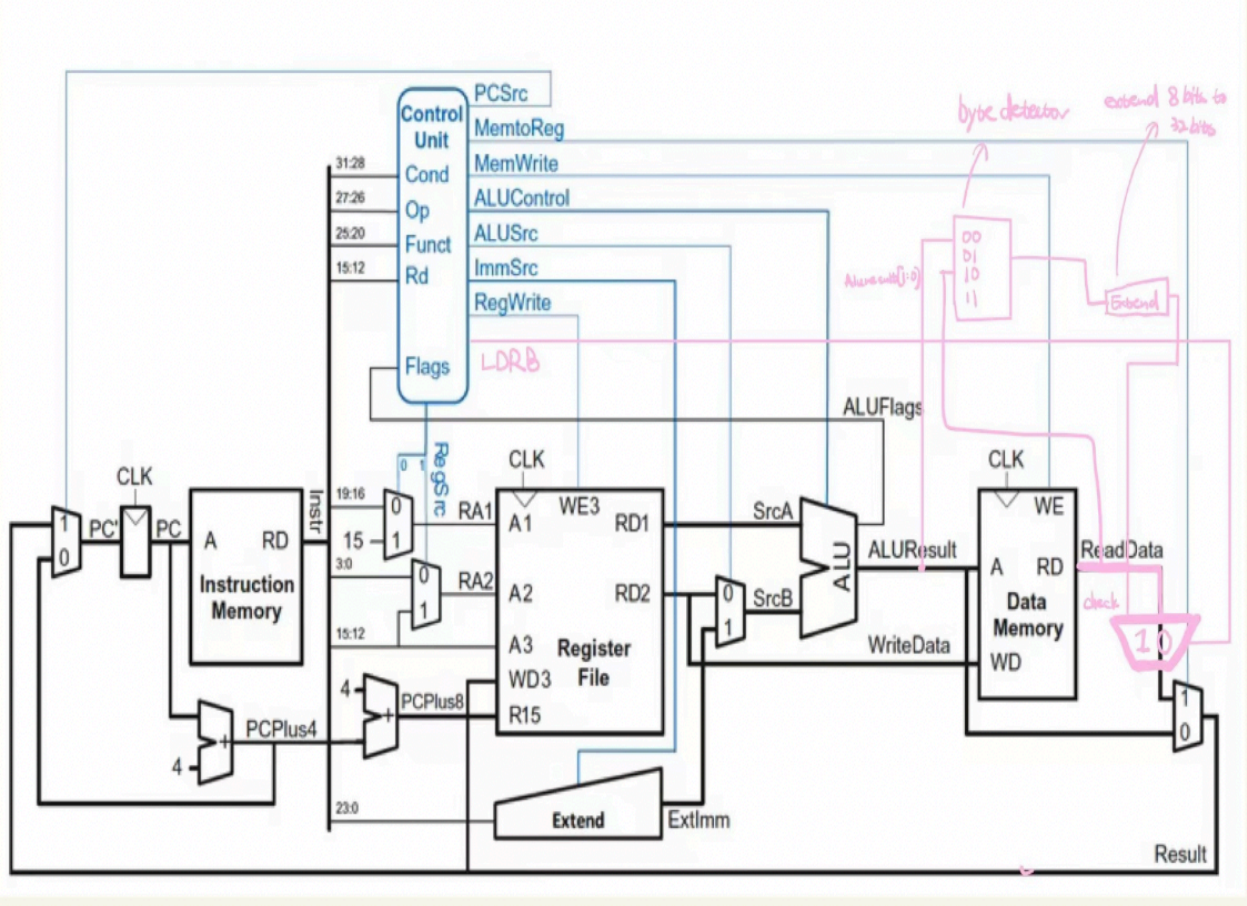
ALUControl changes from 1:0 to 2:0



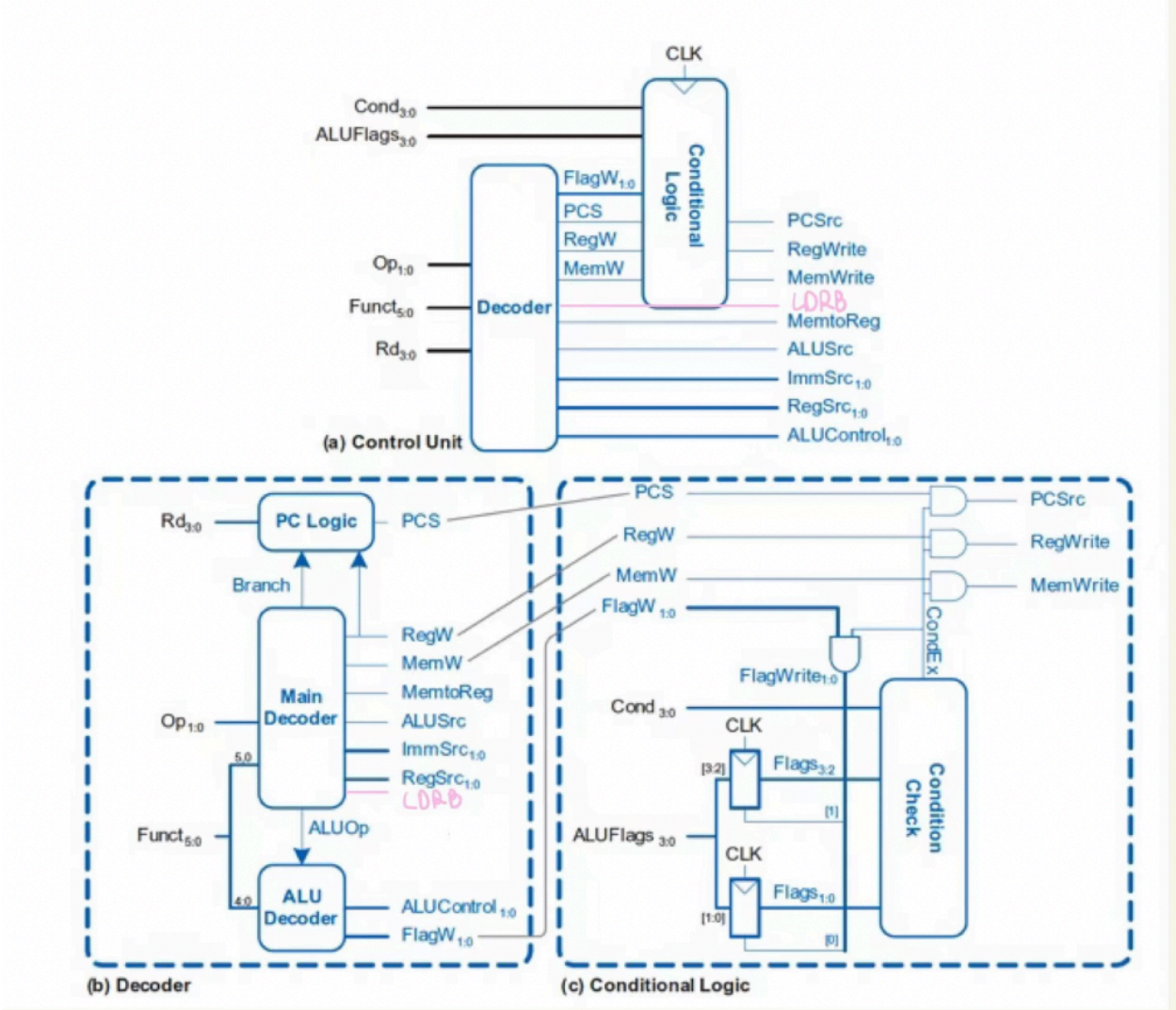
2. LDRB

Second, to implement LDRB, we notice that it's very similar to the LDR instruction which we are very familiar. Notice the difference between LDR and LDBR, we decide to have a Byte Detector to detect which bytes (8 bits) we want to load from the input register. And then adding a 2:1 multiplexer to select whether we want the ReadData or the LDBR result and here I set LDBR as a selector.

Datapath for LDRB



Add a wire LDRB to Main Decoder


CLK
Cond3:0
ALUFlags3:0
FlagW1:0
PCS
RegW
MemW
Conditional Logic
PCSrc
RegWrite
MemWrite
Op1:0
Funct5:0
Rd3:0
Decoder
LDRB
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
ALUControl1:0
(a) Control Unit
Rd3:0
PC Logic
PCS
Branch
RegW
MemW
Main Decoder
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
LDRB
Op1:0
5:0
Funct5:0
ALUOp
4:0
ALU Decoder
ALUControl1:0
FlagW1:0
(b) Decoder
PCS
RegW
MemW
FlagW1:0
PCSrc
RegWrite
MemWrite
CondEx
FlagWrite1:0
Cond3:0
CLK
[3:2]
Flags3:2
[1]
ALUFlags3:0
CLK
[1:0]
Flags1:0
[0]
Condition Check
(c) Conditional Logic

Main Decoder truth table for LDRB

Table 7.2 Main Decoder truth table

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
01	X	X	LDRB	0	1	0	1	01	1	X0	0

Table 7.3 ALU Decoder truth table

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

(d)System Verilog Code

```
//``````````````````````````````````````````````````````````
// arm_single.v
// David_Harris@hmc.edu and Sarah_Harris@hmc.edu 25 June 2013
//
// Single-cycle implementation of a subset of ARMv4
//
// run 210
// Expect simulator to print "Simulation succeeded"
// when the value 7 is written to address 100 (0x64)

// 16 32-bit registers
// Data-processing instructions
//   ADD, SUB, AND, ORR
//   INSTR<cond><S> rd, rn, #immediate
```

```
//   INSTR<cond><S> rd, rn, rm
//    rd <- rn INSTR rm                if (S) Update Status Flags
//    rd <- rn INSTR immediate         if (S) Update Status Flags
//   Instr[31:28] = cond
//   Instr[27:26] = op = 00
//   Instr[25:20] = funct
//                [25]:    1 for immediate, 0 for register
//                [24:21]: 0100 (ADD) / 0010 (SUB) /
//                         0000 (AND) / 1100 (ORR)
//                [20]:    S (1 = update CPSR status Flags)
//   Instr[19:16] = rn
//   Instr[15:12] = rd
//   Instr[11:8]  = 0000
//   Instr[7:0]   = imm8      (for #immediate type) /
//                {0000,rm} (for register type)
//
// Load/Store instructions
//   LDR, STR
//   INSTR rd, [rn, #offset]
//    LDR: rd <- Mem[rn+offset]
//    STR: Mem[rn+offset] <- rd
//   Instr[31:28] = cond
//   Instr[27:26] = op = 01
//   Instr[25:20] = funct
//                [25]:    0 (A)
//                [24:21]: 1100 (P/U/B/W)
//                [20]:    L (1 for LDR, 0 for STR)
//   Instr[19:16] = rn
//   Instr[15:12] = rd
//   Instr[11:0]  = imm12 (zero extended)
//
// Branch instruction (PC <= PC + offset, PC holds 8 bytes past Branch Instr)
//   B
//   B target
//    PC <- PC + 8 + imm24 << 2
//   Instr[31:28] = cond
//   Instr[27:25] = op = 10
//   Instr[25:24] = funct
//                [25]: 1 (Branch)
//                [24]: 0 (link)
//   Instr[23:0]  = imm24 (sign extend, shift left 2)
```

```
//   Note: no Branch delay slot on ARM
//
// Other:
//   R15 reads as PC+8
//   Conditional Encoding
//    cond  Meaning                  Flag
//    0000  Equal                   Z = 1
//    0001  Not Equal               Z = 0
//    0010  Carry Set               C = 1
//    0011  Carry Clear             C = 0
//    0100  Minus                   N = 1
//    0101  Plus                    N = 0
//    0110  Overflow                V = 1
//    0111  No Overflow             V = 0
//    1000  Unsigned Higher         C = 1 & Z = 0
//    1001  Unsigned Lower/Same     C = 0 | Z = 1
//    1010  Signed greater/equal    N = V
//    1011  Signed less             N != V
//    1100  Signed greater          N = V & Z = 0
//    1101  Signed less/equal       N != V | Z = 1
//    1110  Always                  any

//``````````````````````````````````````````````````````````
module testbench();

  logic       clk;
  logic       reset;

  logic [31:0] WriteData, DataAdr;
  logic       MemWrite;

  //``````````````````````````````````````````````
  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  //``````````````````````````````````````````````
  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end
```

```
  //````````````````````````````````````````````````
  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  //````````````````````````````````````````````````
  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 7) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) 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);

  //````````````````````````
  // instruction memory
  //
  imem imem(PC, Instr);
```

```
  //``````````````````````
  // data memory
  //
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

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

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

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

//``````````````````````````````````````````````````````````````
// the instruction memory loads memfile.dat into RAM
// to be executed
//
module imem(input  logic [31:0] a,
            output logic [31:0] rd);

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

  initial
     $readmemh("memfile1.dat",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

//``````````````````````````````````````````````````````````````
// This module is the arm CPU
//
module arm(input  logic        clk, reset,
           output logic [31:0] PC,
           input  logic [31:0] Instr,
```

```
        output logic        MemWrite,
        output logic [31:0] ALUResult, WriteData,
        input  logic [31:0] ReadData);

  logic [3:0] ALUFlags;
  logic       RegWrite,
              ALUSrc, MemtoReg, PCSrc;
  logic [1:0] RegSrc, ImmSrc;
  logic [2:0] ALUControl;

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc);

  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule

//``````````````````````````````````````````````````````````````
// The controller takes in the bits that define the instruction
// and outputs all of the control signals to control the
// execution of each instruction
//
module controller(input  logic        clk, reset,
                        input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic [1:0]   RegSrc,
                  output logic         RegWrite,
                  output logic [1:0]   ImmSrc,
                  output logic         ALUSrc,
                  output logic [2:0]   ALUControl,
                  output logic         MemWrite, MemtoReg,
                  output logic         PCSrc,
                  output logic         LDRB,
                  output logic         Shift
                  );
```

```
  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

  decode dec(Instr[27:26], Instr[25:20], Instr[15:12],
             FlagW, PCS, RegW, MemW,
             MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl, LDRB, NoWrite,Shift);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite, NoWrite);
endmodule

//``````````````````````````````````````````````````````````````````
module decode(input  logic [1:0] Op,
              input  logic [5:0] Funct,
              input  logic [3:0] Rd,
              output logic [1:0] FlagW,
              output logic       PCS, RegW, MemW,
              output logic       MemtoReg, ALUSrc,
              output logic [1:0] ImmSrc, RegSrc,
              output logic [2:0] ALUControl,
              output logic LDRB,
              output logic NoWrite,
              output logic Shift
              );

  logic [9:0] controls;
  logic Branch, ALUOp;
  assign LDRB = Funct[2];
  //``````````````````````
  // Main Decoder
  //
  always_comb
          casex(Op)
                              // Data processing immediate
            2'b00: if (Funct[5])  controls = 10'b0000101001;
                              // Data processing register
                   else           controls = 10'b0000001001;
                              // LDR
            2'b01: if (Funct[0])  controls = 10'b0001111000;
                              // STR
                   else           controls = 10'b1001110100;
```

```
                        // B
          2'b10:            controls = 10'b0110100010;
                        // Unimplemented
          default:          controls = 10'bx;
        endcase

assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
        RegW, MemW, Branch, ALUOp} = controls;

//``````````````````````
// ALU Decoder
//
always_comb
  if (ALUOp) begin                  // which DP Instr?
    case(Funct[4:1])
              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; // X_ORR
             4'b1010: ALUControl = 3'b001; // CMP
             4'b1101: ALUControl = 3'bx; // Shift
              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);
  end else begin
    ALUControl = 3'b000; // add for non-DP instructions
    FlagW      = 2'b00; // don't update Flags

  end
  always_comb
    if (Funct[4:1] == 4'b1010) begin
      NoWrite <= 1;
    end else begin
      NoWrite <= 0;
```

```
    end
   always_comb
    if (Funct[4:1] == 4'b1101) begin
      Shift <= 1;
    end else begin
      Shift <= 0;
    end

  //^^^^^^^^^^^^^^^^^^^^^^
  // PC Logic
  //
  assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
endmodule

//^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
module condlogic(input  logic       clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic       PCS, RegW, MemW,
                 output logic       PCSrc, RegWrite, MemWrite,
                 input  logic       NoWrite
                 );

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  //^^^^^^^^^^^^^^^^^^^^^^
  // write controls are conditional
  //
  condcheck cc(Cond, Flags, CondEx);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondEx & (~NoWrite);
  assign MemWrite  = MemW  & CondEx;
  assign PCSrc     = PCS   & CondEx;
```

```
endmodule

//``````````````````````````````````````````````````````````````
module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase
endmodule

//``````````````````````````````````````````````````````````````
module datapath(input  logic        clk, reset,
                input  logic [1:0]  RegSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic        ALUSrc,
                input  logic [2:0]  ALUControl,
                input  logic        MemtoReg,
```

```
            input  logic        PCSrc,
            output logic [3:0]  ALUFlags,
            output logic [31:0] PC,
            input  logic [31:0] Instr,
            output logic [31:0] ALUResult, WriteData,
            input  logic [31:0] ReadData,
            input  logic LDRB,
            input  logic Shift
            );

logic [31:0] PCNext, PCPlus4, PCPlus8;
logic [31:0] ExtImm, SrcA, SrcB, Result;
logic [3:0]  RA1, RA2;

// next PC logic
mux2 #(32)  pcmux(PCPlus4, Result, PCSrc, PCNext);
flopr #(32) pcreg(clk, reset, PCNext, PC);
adder #(32) pcadd1(PC, 32'b100, PCPlus4);
adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);

// register file logic
mux2 #(4)   ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
mux2 #(4)   ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);

regfile     rf(clk, RegWrite, RA1, RA2,
            Instr[15:12], Result, PCPlus8,
            SrcA, WriteData);


 //LDRB
 logic [31:0] check;
 logic [31:0] FinalData;
 always_comb
   case(ALUResult[1:0])
   2'b00: check = {24'b0, ReadData[7:0]};
   2'b01: check = {16'b0, ReadData[15:8],8'b0};
   2'b10: check = {8'b0, ReadData[23:16],16'b0};
   2'b11: check = {ReadData[31:24], 24'b0};
   endcase

 assign FinalData = LDRB ? check : ReadData;
```

```
  mux2 #(32)  resmux(ALUResult, FinalData, MemtoReg, Result);
  extend      ext(Instr[23:0], ImmSrc, ExtImm);

  //Shifter
  logic [31:0]shift_result;
  shifter Sh(WriteData,Instr[11:7],Instr[6:5],shift_result);

  logic [31:0] ALUResult1;
  // ALU logic
  mux2 #(32)  srcbmux(shift_result, ExtImm, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl,
                  ALUResult1, ALUFlags);
  //shift
  mux2 shift_mux(ALUResult1, SrcB, Shift, ALUResult);
endmodule

//``````````````````````````````````````````````````````````````
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [3:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[14:0];

  //``````````````````````
  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 15 reads PC+8 instead

  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];

endmodule

//``````````````````````````````````````````````````````````````
```

```
module extend(input  logic [23:0] Instr,
              input  logic [1:0]  ImmSrc,
              output logic [31:0] ExtImm);

  always_comb
    case(ImmSrc)
               // 8-bit unsigned immediate
      2'b00:   ExtImm = {24'b0, Instr[7:0]};
               // 12-bit unsigned immediate
      2'b01:   ExtImm = {20'b0, Instr[11:0]};
               // 24-bit two's complement shifted branch
      2'b10:   ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default: ExtImm = 32'bx; // undefined
    endcase
endmodule

//``````````````````````````````````````````````````````````````
module adder #(parameter WIDTH=8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule

//``````````````````````````````````````````````````````````````
module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

//``````````````````````````````````````````````````````````````
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);
```

```
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

//``````````````````````````````````````````````````````````````
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module shifter(
              input logic [31:0] ALUResult,
              input logic [6:0] shamt,
              input logic [1:0] sh,
              output logic [31:0] y);
        always_comb
          case(sh)
          //LSL
          2'b00 : y = ALUResult << shamt;
          //LSR
          2'b01 : y = ALUResult << shamt;
          //ASR
          2'b10 : y = ALUResult <<< shamt;
          //ROR
          2'b11 : y = ALUResult >>> shamt;

          endcase
endmodule
```

alu.sv

```
module alu(input logic [31:0] a, b, input  logic [2:0] ALUControl,
            output  logic [31:0] Result, output  logic [3:0] ALUFlags);

    logic [31:0]cout;
    logic [31:0]sum;
    logic [31:0] B;
    logic [31:0] d2;
```

```
    logic [31:0] d3;
    logic [31:0] d4;
    logic [31:0] d5;
    logic [31:0] d6;
    logic [31:0] d7;
    logic Xor;
    logic Xnor;

    assign B = ALUControl[0] ? ~b : b;

    N_bit_adder n (a, B, ALUControl[0], sum, cout);
    andfunction h (a, b, d2);
    orfunction s (a, b, d3);
    xorfunction k (a,b,d4);

    mux8 m (sum, sum, d2, d3, d4, d5, d6, d7,ALUControl,Result);

    //Overflow
    assign Xor = (a[31]^sum[31]);
    assign Xnor = (~(a[31]^b[31]^ALUControl[0]));
    assign ALUFlags[0] = ((~ALUControl[1])& Xor & Xnor);
    //Carry out
    assign ALUFlags[1] = (cout & (~ALUControl[1]));
    //Zero
    assign ALUFlags[2] = &(~Result);
    //Negative
    assign ALUFlags[3] = (Result[31]);

endmodule

module mux8(input logic [31:0] d0, d1, d2, d3, d4, d5, d6, d7,
        input logic [2:0] s,
        output logic [31:0] y);
        assign y = s[2] ? (s[1] ? (s[0] ? d7 : d6) : (s[0] ? d5 : d4) )
                        : (s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0) );
endmodule

module N_bit_adder(input1,input2, cin, answer, cout);
    parameter N=32;
    input [N-1:0] input1,input2, cin;
    output [N-1:0] answer, cout;
    logic [N-1:0]carry_out;
    logic [N-1:0] carry;
    genvar i;
    generate
   for(i=0;i<N;i=i+1)
     begin: generate_N_bit_Adder
   if(i==0)
```

```
    full_adder f(input1[0],input2[0], cin, answer[0],carry[0]);
   else
    full_adder f(input1[i],input2[i],carry[i-1],answer[i],carry[i]);
     end
  assign carry_out = carry[N-1];
  assign cout = carry_out;
   endgenerate
endmodule

module full_adder(x,y,c_in,s,c_out);
   input x,y,c_in;
   output s,c_out;
 assign s = (x^y) ^ c_in;
 assign c_out = (y&c_in)| (x&y) | (x&c_in);
endmodule

module andfunction(input logic[31:0]a, b,
                        output logic [31:0]y);
        assign y = a & b;
endmodule

module orfunction(input logic [31:0]a, b,
                        output logic [31:0]y);
        assign y = a | b;
endmodule
module xorfunction(input logic [31:0]a, b,
                        output logic [31:0]y);
        assign y = a ^ b;
endmodule
```

Part 4

Testing your modified ARM single-cycle processor：

In this part, we'd like to check whether our modified arm processor is well performed. So, we convert the given program to machine code and then modified the testbench to check whether our address and data value is the same as what we expected.

(e)memfile2.dat

E04F000F

E28010FF

E0812001

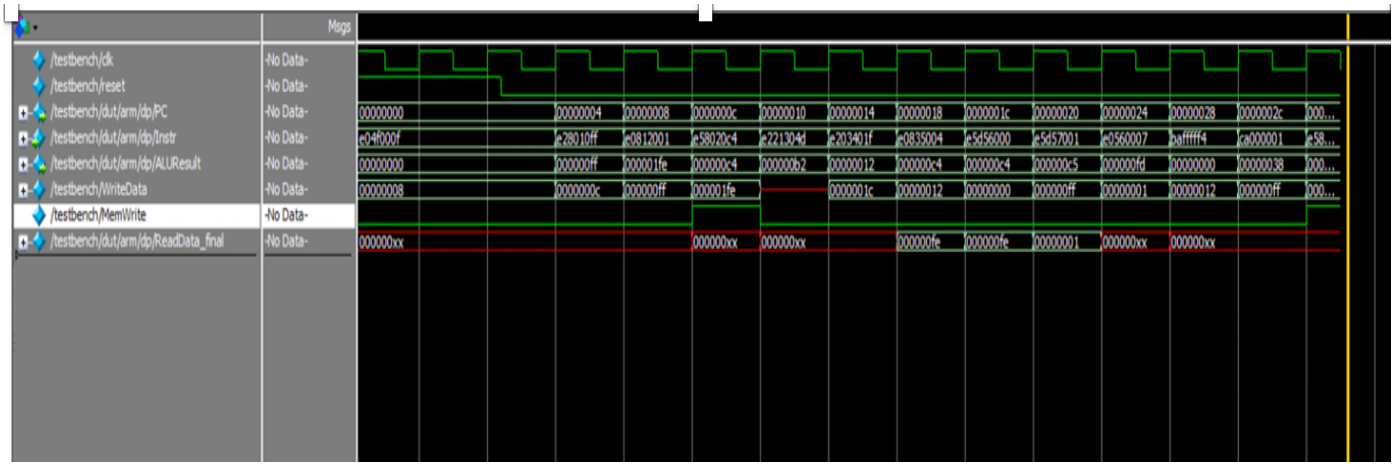
E58020C4

E221304D

E203401F

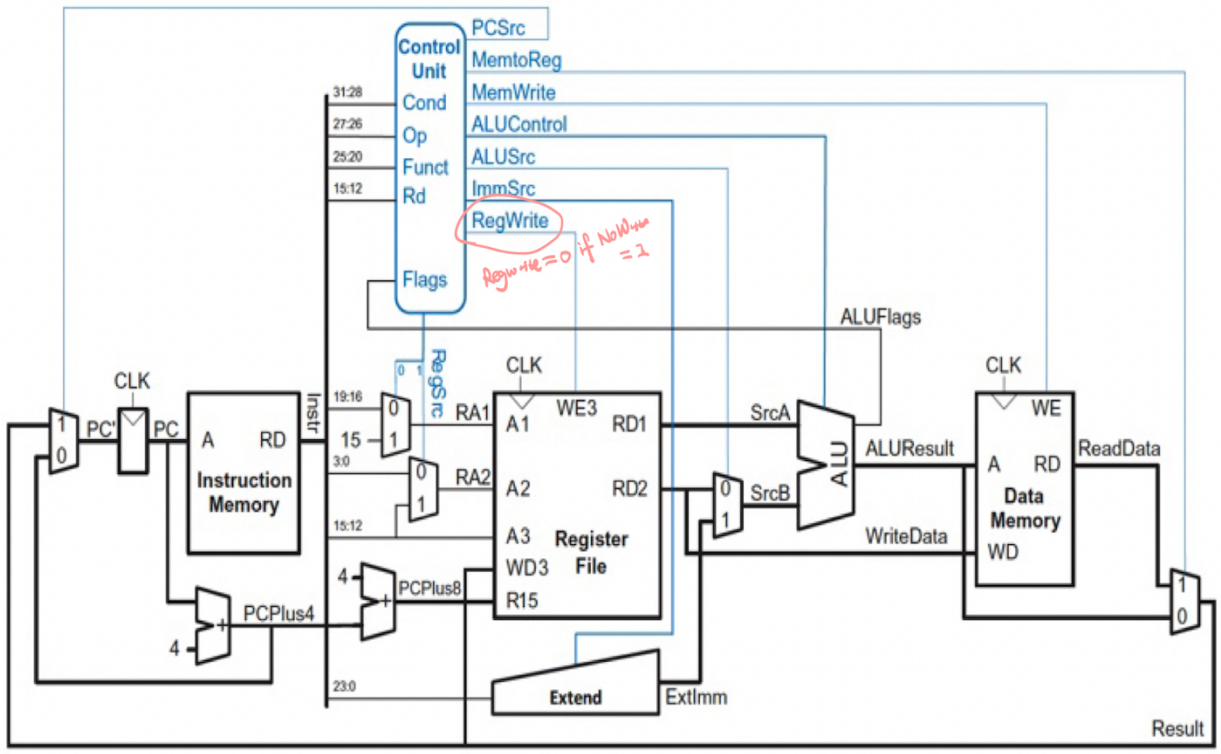
E0835004

E5D56000
E5D57001
E0560007
BAFFFFF4
CA000001
E584106E
EAFFFFF1
E584606E
(f)
By final STR, the address is 128 and the value is 254.
Here is the waveform.

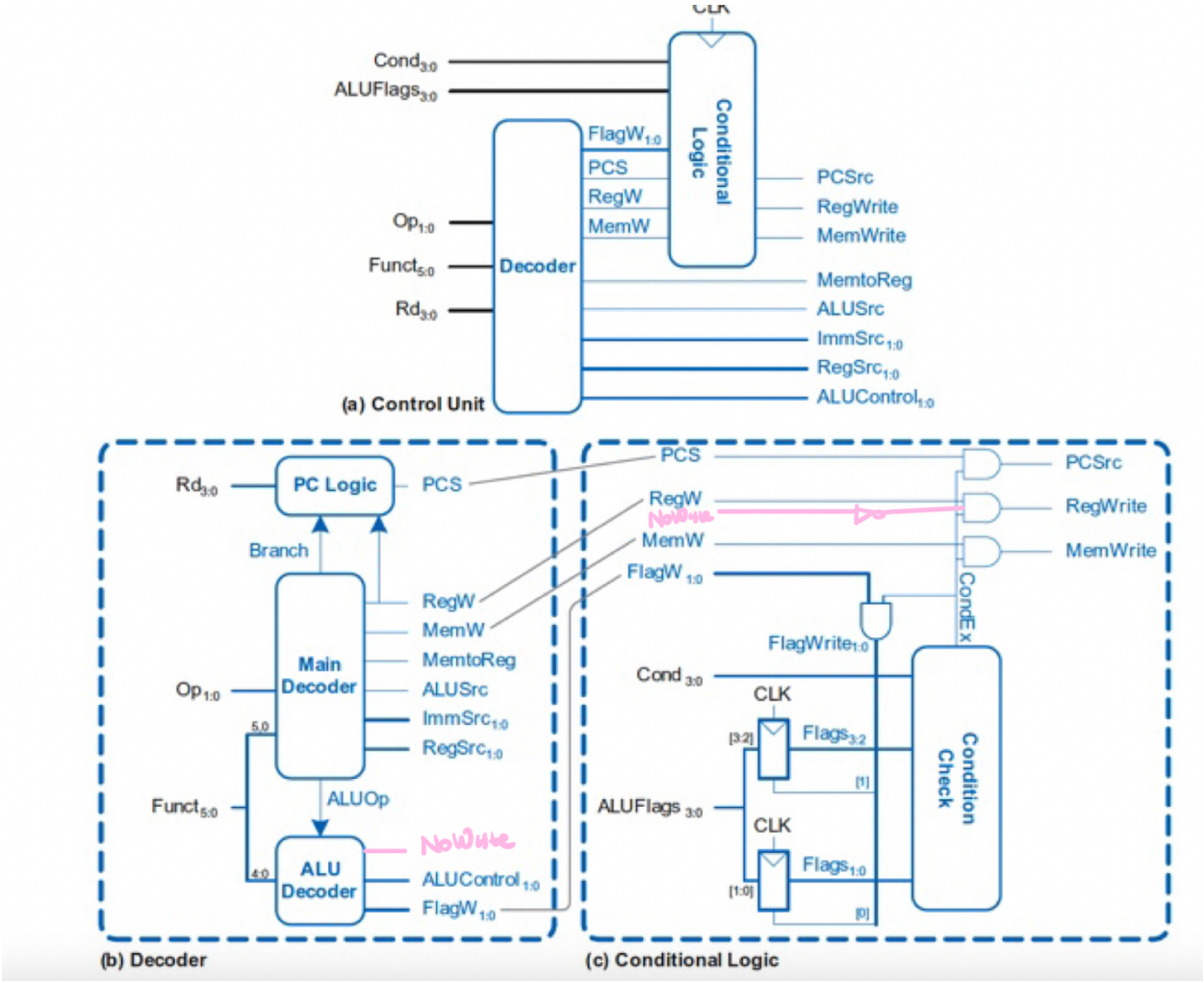


Part 5
Further modifications of your ARM single-cycle processor：
In part 5, we would like to implement two instructions LSL and LSR. But first, we should implement CMP instruction. To do so, we consider CMP should set flags and do the sub instruction. Then, we're supposed to add a CMP into our ALU Decoder and update the ALU truth table. Also, we should consider what is special to the CMP instruction. When the CMP instruction executes, the result won't be written into the register file. Thus, we should add a NOWrite signal that make the RegWrite unavalible.

Datapath for CMP



Decoder for CMP


CLK
Cond3:0
ALUFlags3:0
Conditional Logic
FlagW1:0
PCS
RegW
MemW
PCSrc
RegWrite
MemWrite
Op1:0
Funct5:0
Rd3:0
Decoder
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
ALUControl1:0
(a) Control Unit
Rd3:0
PC Logic
PCS
Branch
RegW
MemW
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
Op1:0
Main Decoder
5:0
Funct5:0
ALUOp
4:0
ALU Decoder
ALUControl1:0
FlagW1:0
(b) Decoder
PCS
RegW
MemW
FlagW1:0
PCSrc
RegWrite
MemWrite
CondEx
FlagWrite1:0
Cond3:0
CLK
[3:2]
Flags3:2
[1]
ALUFlags3:0
CLK
[1:0]
Flags1:0
[0]
Condition Check
(c) Conditional Logic

Decode Table for CMP

Table 7.2 Main Decoder truth table

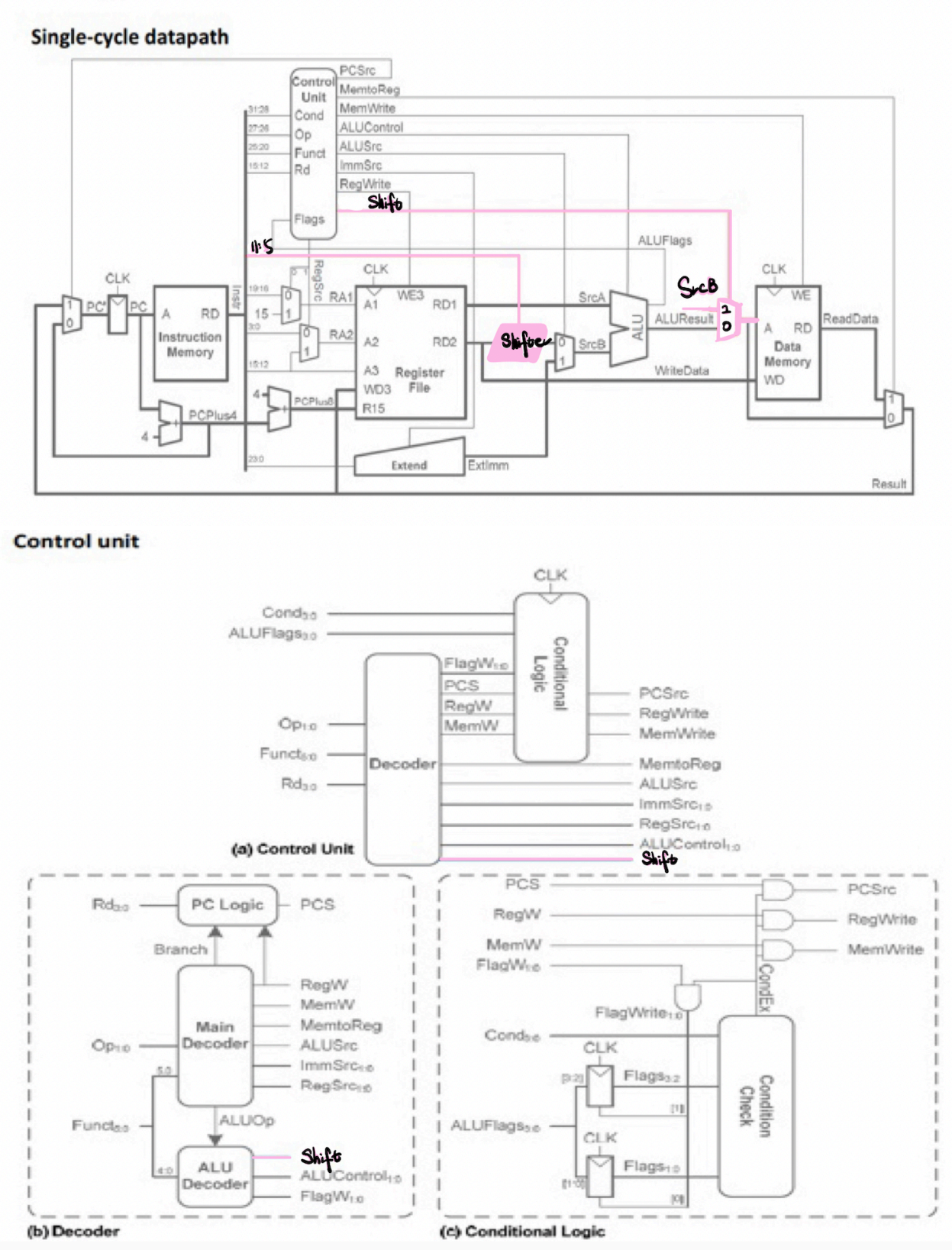
Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Table 7.3 ALU Decoder truth table

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10
1	1010	1	CMP	01	11

2. Shift(LSL and LSR)

Datapath for LSL and LSR



ALU Decoder truth table

ALU Decoder truth table

ALUOp	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	Notes	$ALUControl_{1:0}$	$FlagW_{1:0}$	Shifts
0	X	X	Not DP	00	00	0
1	0100	0	ADD	00	00	0
		1			11	0
	0010	0	SUB	01	00	0
		1			11	0
	0000	0	AND	10	00	0
		1			10	0
	1100	0	ORR	11	00	0
		1			10	0
	1101	0	LSL	XX	00	1
		1			10	1
1	1101	0	LSR	XX	00	1
		1			10	1

(h)

```
//`````````````````````````````````````````````````````````````````
// arm_single.v
// David_Harris@hmc.edu and Sarah_Harris@hmc.edu 25 June 2013
//
// Single-cycle implementation of a subset of ARMv4
//
// run 210
// Expect simulator to print "Simulation succeeded"
// when the value 7 is written to address 100 (0x64)

// 16 32-bit registers
// Data-processing instructions
//   ADD, SUB, AND, ORR
//   INSTR<cond><S> rd, rn, #immediate
//   INSTR<cond><S> rd, rn, rm
//    rd <- rn INSTR rm       if (S) Update Status Flags
//    rd <- rn INSTR immediate  if (S) Update Status Flags
//   Instr[31:28] = cond
//   Instr[27:26] = op = 00
//   Instr[25:20] = funct
//                  [25]:    1 for immediate, 0 for register
//                  [24:21]: 0100 (ADD) / 0010 (SUB) /
//                           0000 (AND) / 1100 (ORR)
//                  [20]:    S (1 = update CPSR status Flags)
//   Instr[19:16] = rn
```

```
//   Instr[15:12] = rd
//   Instr[11:8]  = 0000
//   Instr[7:0]   = imm8      (for #immediate type) /
//                  {0000,rm} (for register type)
//
// Load/Store instructions
//   LDR, STR
//   INSTR rd, [rn, #offset]
//    LDR: rd <- Mem[rn+offset]
//    STR: Mem[rn+offset] <- rd
//   Instr[31:28] = cond
//   Instr[27:26] = op = 01
//   Instr[25:20] = funct
//                  [25]:    0 (A)
//                  [24:21]: 1100 (P/U/B/W)
//                  [20]:    L (1 for LDR, 0 for STR)
//   Instr[19:16] = rn
//   Instr[15:12] = rd
//   Instr[11:0]  = imm12 (zero extended)
//
// Branch instruction (PC <= PC + offset, PC holds 8 bytes past Branch Instr)
//   B
//   B target
//    PC <- PC + 8 + imm24 << 2
//   Instr[31:28] = cond
//   Instr[27:25] = op = 10
//   Instr[25:24] = funct
//                  [25]: 1 (Branch)
//                  [24]: 0 (link)
//   Instr[23:0]  = imm24 (sign extend, shift left 2)
//   Note: no Branch delay slot on ARM
//
// Other:
//   R15 reads as PC+8
//   Conditional Encoding
//    cond  Meaning                      Flag
//    0000  Equal                        Z = 1
//    0001  Not Equal                    Z = 0
//    0010  Carry Set                    C = 1
//    0011  Carry Clear                  C = 0
//    0100  Minus                        N = 1
//    0101  Plus                         N = 0
//    0110  Overflow                     V = 1
//    0111  No Overflow                  V = 0
//    1000  Unsigned Higher              C = 1 & Z = 0
//    1001  Unsigned Lower/Same          C = 0 | Z = 1
//    1010  Signed greater/equal         N = V
//    1011  Signed less                  N != V
```

```
//    1100  Signed greater               N = V & Z = 0
//    1101  Signed less/equal            N != V | Z = 1
//    1110  Always                       any

//``````````````````````````````````````````````````````````````
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  //````````````````````````````````````````````````
  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

  //````````````````````````````````````````````````
  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  //````````````````````````````````````````````````
  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  //````````````````````````````````````````````````
  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 7) begin
          $display("Simulation succeeded");
          $stop;
        end else if (DataAdr !== 96) 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);

  //``````````````````````
  // instruction memory
  //
  imem imem(PC, Instr);

  //``````````````````````
  // data memory
  //
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

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

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

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

//````````````````````````````````````````````````````````````````
// the instruction memory loads memfile.dat into RAM
// to be executed
//
module imem(input  logic [31:0] a,
            output logic [31:0] rd);

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

  initial
      $readmemh("memfile.dat",RAM);
```

```
  assign rd = RAM[a[31:2]]; // word aligned
endmodule

//------------------------------------------------------------------
// This module is the arm CPU
//
module arm(input  logic        clk, reset,
           output logic [31:0] PC,
           input  logic [31:0] Instr,
           output logic        MemWrite,
           output logic [31:0] ALUResult, WriteData,
           input  logic [31:0] ReadData);

  logic [3:0] ALUFlags;
  logic       RegWrite,
              ALUSrc, MemtoReg, PCSrc;
  logic [1:0] RegSrc, ImmSrc;
  logic [2:0] ALUControl;

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc);

  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule

//------------------------------------------------------------------
// The controller takes in the bits that define the instruction
// and outputs all of the control signals to control the
// execution of each instruction
//
module controller(input  logic         clk, reset,
                  input  logic [31:12] Instr,
                    input  logic [3:0]   ALUFlags,
                    output logic [1:0]   RegSrc,
                    output logic         RegWrite,
                    output logic [1:0]   ImmSrc,
                    output logic         ALUSrc,
                    output logic [2:0]   ALUControl,
                    output logic         MemWrite, MemtoReg,
                    output logic         PCSrc,
                    output logic         LDRB,
```

```
                output logic         Shift
                );
  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

  decode dec(Instr[27:26], Instr[25:20], Instr[15:12],
             FlagW, PCS, RegW, MemW,
             MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl, LDRB, NoWrite,Shift);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite, NoWrite);
endmodule

//`````````````````````````````````````````````````````````````
module decode(input  logic [1:0] Op,
              input  logic [5:0] Funct,
              input  logic [3:0] Rd,
              output logic [1:0] FlagW,
              output logic       PCS, RegW, MemW,
              output logic       MemtoReg, ALUSrc,
              output logic [1:0] ImmSrc, RegSrc,
              output logic [2:0] ALUControl,
              output logic LDRB,
              output logic NoWrite,
              output logic Shift
              );

  logic [9:0] controls;
  logic Branch, ALUOp;
  assign LDRB = Funct[2];

  //``````````````````````
  // Main Decoder
  //
  always_comb
    casex(Op)
                            // Data processing immediate
      2'b00: if (Funct[5])  controls = 10'b0000101001;
                            // Data processing register
             else           controls = 10'b0000001001;
                            // LDR
      2'b01: if (Funct[0])  controls = 10'b0001111000;
                            // STR
             else           controls = 10'b1001110100;
                            // B
      2'b10:                controls = 10'b0110100010;
                            // Unimplemented
      default:              controls = 10'bx;
```

```
    endcase

  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
          RegW, MemW, Branch, ALUOp} = controls;

  //``````````````````````
  // ALU Decoder
  //
  always_comb
    if (ALUOp) begin                 // which DP Instr?
      case(Funct[4:1])
        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; // X_ORR
        4'b1010: ALUControl = 3'b001; // CMP
        4'b1101: ALUControl = 3'bx; // Shift
        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);
    end else begin
      ALUControl = 3'b000; // add for non-DP instructions
      FlagW      = 2'b00; // don't update Flags

    end
    always_comb
      if (Funct[4:1] == 4'b1010) begin
        NoWrite <= 1;
      end else begin
        NoWrite <= 0;
      end
    always_comb
      if (Funct[4:1] == 4'b1101) begin
        Shift <= 1;
      end else begin
        Shift <= 0;
      end

  //``````````````````````
  // PC Logic
  //
```

```
  assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
endmodule

//`````````````````````````````````````````````````````````````````
module condlogic(input  logic       clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic       PCS, RegW, MemW,
                 output logic       PCSrc, RegWrite, MemWrite,
                 input  logic       NoWrite
                 );

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  //``````````````````````
  // write controls are conditional
  //
  condcheck cc(Cond, Flags, CondEx);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondEx & (~NoWrite);
  assign MemWrite  = MemW  & CondEx;
  assign PCSrc     = PCS   & CondEx;
endmodule

//`````````````````````````````````````````````````````````````````
module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
```

```
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase
endmodule

//`````````````````````````````````````````````````````````````
module datapath(input  logic        clk, reset,
                input  logic [1:0]  RegSrc,
                input  logic        RegWrite,
                input  logic [1:0]  ImmSrc,
                input  logic        ALUSrc,
                input  logic [2:0]  ALUControl,
                input  logic        MemtoReg,
                input  logic        PCSrc,
                output logic [3:0]  ALUFlags,
                output logic [31:0] PC,
                input  logic [31:0] Instr,
                output logic [31:0] ALUResult, WriteData,
                input  logic [31:0] ReadData,
                input  logic LDRB,
                input  logic Shift
                );

  logic [31:0] PCNext, PCPlus4, PCPlus8;
  logic [31:0] ExtImm, SrcA, SrcB, Result;
  logic [3:0]  RA1, RA2;

  // next PC logic
  mux2 #(32)  pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder #(32) pcadd1(PC, 32'b100, PCPlus4);
  adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);

  // register file logic
  mux2 #(4)   ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4)   ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);

  regfile     rf(clk, RegWrite, RA1, RA2,
```

```
                Instr[15:12], Result, PCPlus8,
                SrcA, WriteData);


  //LDRB
  logic [31:0] check;
  logic [31:0] FinalData;
  always_comb
    case(ALUResult[1:0])
    2'b00: check = {24'b0, ReadData[7:0]};
    2'b01: check = {16'b0, ReadData[15:8],8'b0};
    2'b10: check = {8'b0, ReadData[23:16],16'b0};
    2'b11: check = {ReadData[31:24], 24'b0};
    endcase

  assign FinalData = LDRB ? check : ReadData;

  mux2 #(32)  resmux(ALUResult, FinalData, MemtoReg, Result);
  extend      ext(Instr[23:0], ImmSrc, ExtImm);

  //Shifter
  logic [31:0]shift_result;
  shifter Sh(WriteData,Instr[11:7],Instr[6:5],shift_result);

  logic [31:0] ALUResult1;
  // ALU logic
  mux2 #(32)  srcbmux(shift_result, ExtImm, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl,
                  ALUResult1, ALUFlags);
  //shift
  mux2 shift_mux(ALUResult1, SrcB, Shift, ALUResult);
endmodule

//------------------------------------------------------------------
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [3:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[14:0];

  //----------------------
  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 15 reads PC+8 instead
```

```
  always_ff @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];

endmodule

//`````````````````````````````````````````````````````````````````
module extend(input  logic [23:0] Instr,
              input  logic [1:0]  ImmSrc,
              output logic [31:0] ExtImm);

  always_comb
    case(ImmSrc)
               // 8-bit unsigned immediate
      2'b00:   ExtImm = {24'b0, Instr[7:0]};
               // 12-bit unsigned immediate
      2'b01:   ExtImm = {20'b0, Instr[11:0]};
               // 24-bit two's complement shifted branch
      2'b10:   ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default: ExtImm = 32'bx; // undefined
    endcase
endmodule

//`````````````````````````````````````````````````````````````````
module adder #(parameter WIDTH=8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule

//`````````````````````````````````````````````````````````````````
module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

//`````````````````````````````````````````````````````````````````
module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
```

```
              output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule

//`````````````````````````````````````````````````````````````````
module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module shifter(
              input logic [31:0] ALUResult,
              input logic [6:0] shamt,
              input logic [1:0] sh,
              output logic [31:0] y);
        always_comb
          case(sh)
          //LSL
          2'b00 : y = ALUResult << shamt;
          //LSR
          2'b01 : y = ALUResult << shamt;
          //ASR
          2'b10 : y = ALUResult <<< shamt;
          //ROR
          2'b11 : y = ALUResult >>> shamt;

          endcase
endmodule
```

(i)&(j)

E04F000F #SUB R0,R15,R15

E280101F #ADD R1,R0,#31

E1A02101 #LSL R2,R1,#2

E5802010 # STR R2,[R0,#16]

E1A031A1 #LSR R3,R1,#3

E1530002 # CMP R3,R2

E581302C #STR R3,[R1,#44]

(k)

First, we should implement CMP instruction. To do so, we consider CMP should set flags and do the sub instruction. Then, we're supposed to add a CMP into our ALU Decoder and update the ALU truth table. Also, we should consider what is special to the CMP instruction. When the CMP instruction executes, the result won't be written into the register file. Thus, we should add a NOWrite signal that makes the RegWrite unavailable. For shifter instructions, we can follow textbook question 7.4, we first consider adding a shifter to do the shifting job for us before we send the SrcB to the ALU. Also, the control logic should recognize when to pick the shift result instead of ALU result. So, we make a 2:1 multiplier to select whether we want ALUresult and SrcB by adding a selector called Shift. When the Shift is asserted, the SrcB which is the shifted result will be selected, otherwise the ALUresult will flow. Since there is no arithmetical logic, we don't need to modify ALU but the ALU Decoder truth table should be modified by adding the LSL and LSR instructions.

To detect whether our final single-cycle ALU is well performed, we should come up with all the instructions that our ALU should correctly calculate. So, I wrote 7 tests to check whether our ALU is correct after adding the two new instructions (LSL and LSR). Finally, by debugging some errors, we finished our simple single-cycle ARM processor!!

Final ALU Table

ALUop	Funct(4:1)	Funct(0)	Notes	ALUcontrol(2:0)	FlagW(1:0)	NoWrite
0	X	X	Not OP	000	00	0
1	0100	0	ADD	000	00	0
		1			11	0
1	0010	0	SUB	001	00	0
		1			11	0
1	0000	0	AND	010	00	0
		1			10	0
1	1100	0	ORR	011	00	0
		1			10	0
1	0001	0	EOR	111	00	0
		1			10	0
1	1010	0	CMP	001	11	0
		1				0
1	1101	0	LSL	000	00	0

		1			10	0
1	1101	0	LSR	000	00	0
		1			10	0