Architecture of a single cycle Arm processor

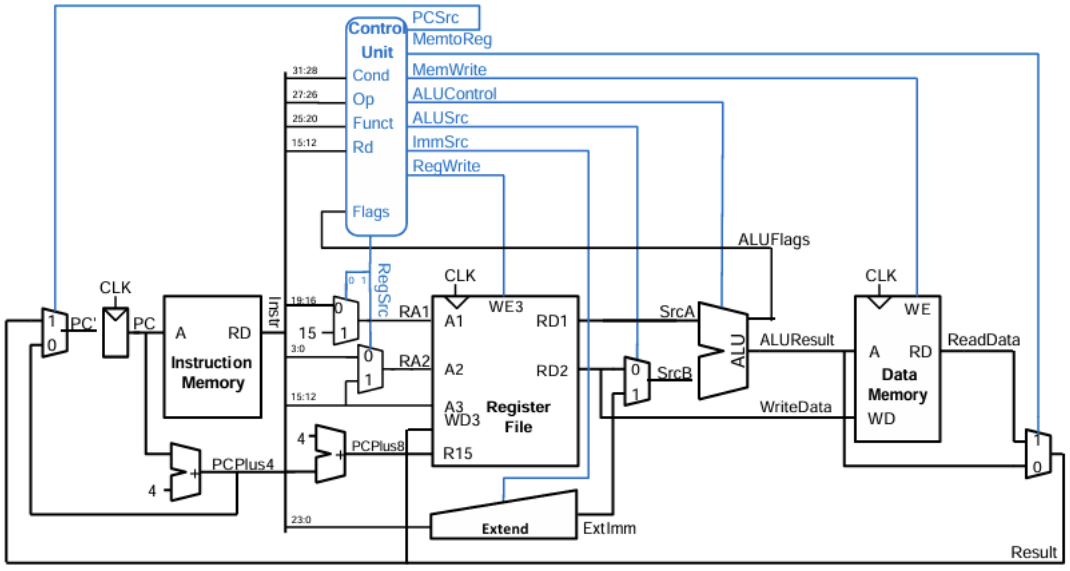


Fig: Single cycle ARM Processor

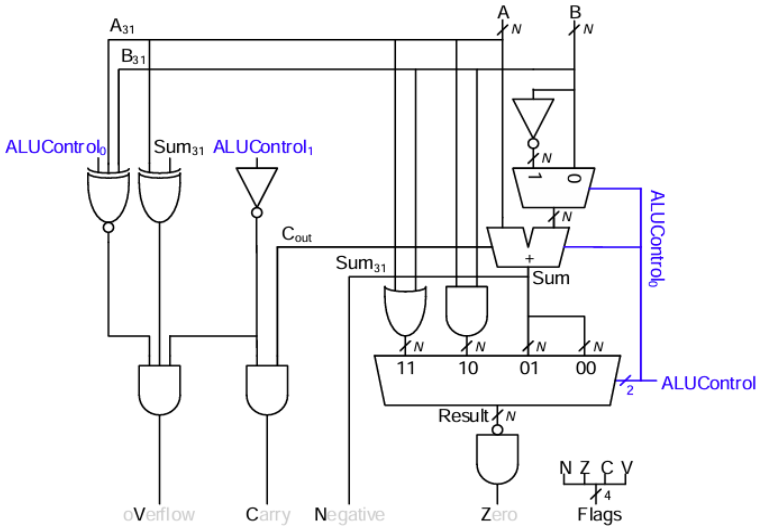


Fig: ARM ALU

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

Fig: Main Decoder

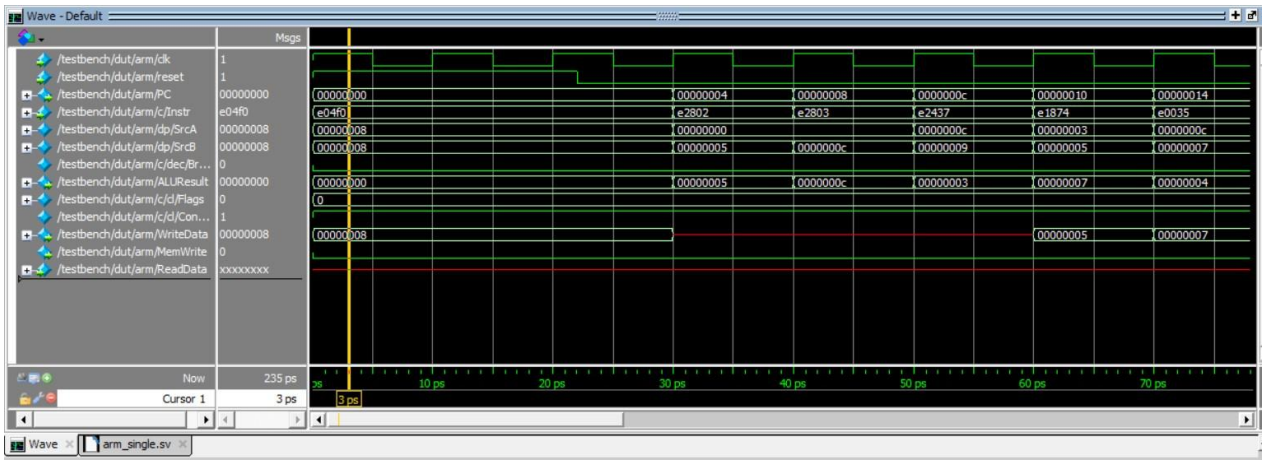
ALUOp	*Funct*$_{4:1}$ *(cmd)*	*Funct*$_0$ *(S)*	Notes	*ALUControl*$_{1:0}$	*FlagW*$_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

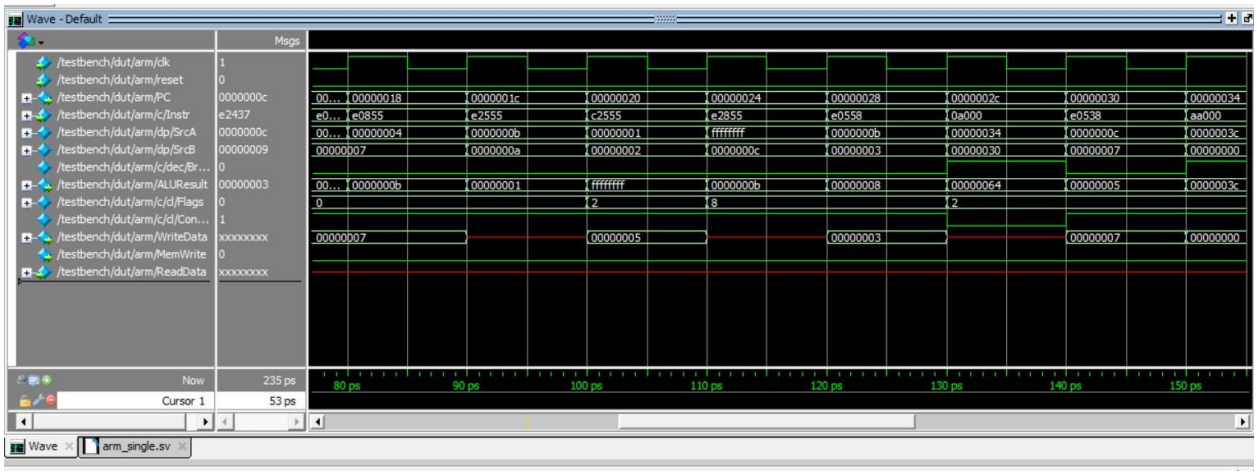
Fig: ALU Decoder

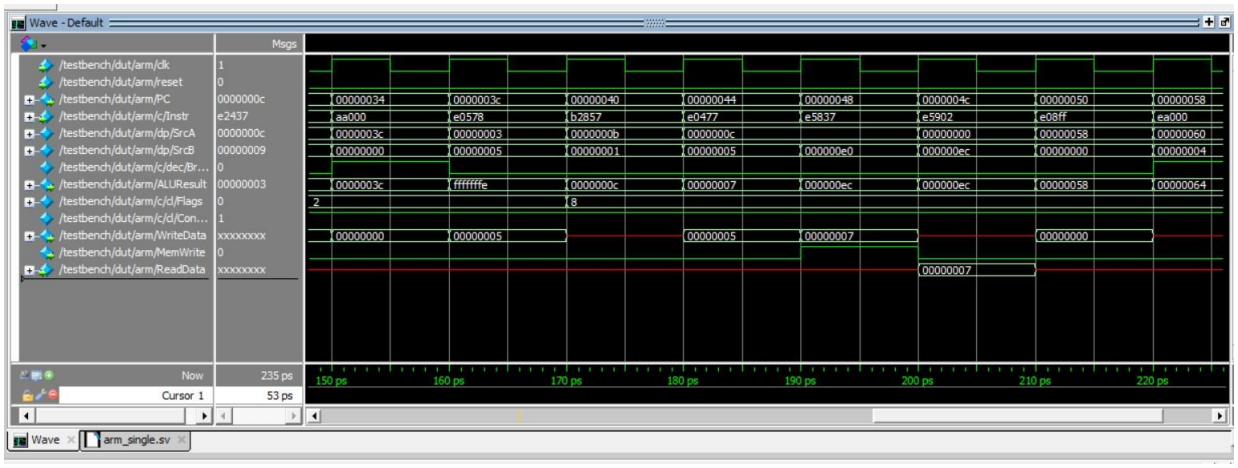
First nineteen cycles of executing different instructions in hexadecimal

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	Flags$_{3:0}$ [NZCV]	CondEx	WriteData	MemWrite	ReadData
1	1	00	SUB R0, R15, R15 E04F000F	8	8	0	0	?0000	1	8	0	x
2	0	04	ADD R2, R0, #5 E2802005	0	5	0	5	?0000	1	x	0	x
3	0	08	ADD R3, R0, #12 E280300C	0	C	0	C	?0000	1	x	0	x
4	0	C(12)	SUB R7, R3, #9 E2437009	C	9	0	3	0000	1	x	0	x
5	0	10(16)	ORR R4, R7, R2 E1874002	3	5	0	7	0000	1	5	0	x
6	0	14(20)	AND R5, R3, R4 E0035004	C(12)	7	0	4	0000	1	7	0	x
7	0	18(24)	ADD R5, R5, R4 E0855004	4	7	0	B(11)	0000	1	7	0	x
8	0	1C(28)	SUBS R5, R5, #10 E255500A	B(11)	A(10)	0	1	0000	1	x	0	x
9	0	20(32)	SUBSGT R5, R5, #2 C2555002	1	2	0	FFFFFFFF (-1)	0000	1	5	0	x
10	0	24(36)	ADD R5, R5, #12 E285500C	FFFFFFFF (-1)	C(12)	0	B(11)	0010	1	x	0	x
11	0	28(40)	SUBS R8, R5, R7 E0558007	B(11)	3	0	8	1000	1	3	0	x
12	0	2C(44)	BEQ END 0A00000C	34(52)	30(48)	FFFFFFFF (-1)	64(100)	1000	0	x	0	x
13	0	30(48)	SUBS R8, R3, R4 E0538004	C(12)	7	0	5	0010	1	7	0	x
14	0	34(52)	BGE AROUND AA000000	3C(60)	0	FFFFFFFF (-1)	3C(60)	0010	1	0	0	x
15	0	3C(60)	SUBS R8, R7, R2 E0578002	3	5	0	FFFFFFFE (-2)	0010	1	5	0	x
16	0	40(64)	ADDLT R7, R5, #1 B2857001	B(11)	1	0	C(12)	1000	1	x	0	x
17	0	44(68)	SUB R7, R7, R2 E0477002	C(12)	5	0	7	1000	1	5	0	x
18	0	48(72)	STR R7, [R3, #224] E58370E0	C(12)	E0(224)	0	EC(236)	1000	1	7	1	x
19	0	48(76)	LDR R2, [R0, #236] E59020EC	0	EC(236)	0	EC(236)	1000	1	x	0	7

Simulation waveform (matches with the expected values from the table) :

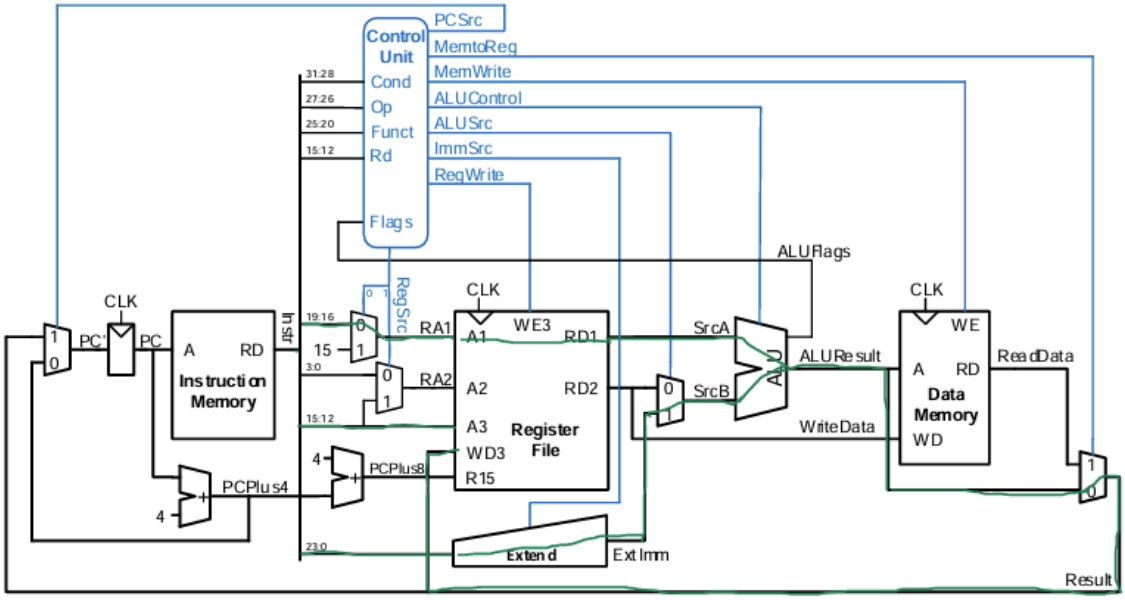






Modify the ARM single-cycle processor by adding the EOR instruction:

Modified architecture for EOR:



System verilog code modified:

```
// arm_single.sv
// 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 62 (0x3e)

// 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 === (62<<2)  & 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);
  imem imem(PC, Instr);
  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

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

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; //updated

  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
```

```
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, //updated
                  output   logic    MemWrite,   MemtoReg,
                  output logic      PCSrc);

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

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

module decoder(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);  //updated

  logic [9:0] controls;
  logic       Branch, ALUOp;

  // Main Decoder

  always_comb
            case(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 = 2'b000; // ADD
                4'b0010: ALUControl = 2'b001; // SUB
               4'b0000: ALUControl = 2'b010; // AND
                4'b1100: ALUControl = 2'b011; // ORR
              4'b0001: ALUControl = 2'b110; // updated 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 == 2'b00 | ALUControl == 2'b01);
    end else begin
      ALUControl = 3'b000; // add for non-DP instructions //updated
      FlagW      = 2'b00; // don't update Flags
    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);

  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;
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 [1:0]  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);

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

  // ALU logic
  mux2 #(32)  srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl,
                  ALUResult, ALUFlags);
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 alu(input  logic [31:0] a, b,
           input  logic [2:0]  ALUControl, //updated
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags);

  logic        neg, zero, carry, overflow;
  logic [31:0] condinvb;
  logic [32:0] sum;

  assign condinvb = ALUControl[0] ? ~b : b;
```

```
  assign sum = a + condinvb + ALUControl[0];

  always_comb
    casex (ALUControl[1:0])
    2'b0?: Result = sum;
    2'b10: Result = a & b;
    2'b11: Result = a | b;
    endcase

  assign neg      = Result[31];
  assign zero     = (Result == 32'b0);
  assign carry    = (ALUControl[1] == 1'b0) & sum[32];
  assign overflow = (ALUControl[1] == 1'b0) &
                    ~(a[31] ^ b[31] ^ ALUControl[0]) &
                    (a[31] ^ sum[31]);
  assign ALUFlags    = {neg, zero, carry, overflow};
endmodule
```

Testing the modified ARM signle cycle processor

ARM assembly Program: memfile2.asm:

```
MAIN    SUB R0, R15, R15
        ADD R1, R0, #255
        ADD R2, R1, R1
        STR R2, [R0, #196]
        EOR R3, R1, #77
        AND R4, R3, #0x1F
        ADD R5, R3, R4
        SUBS R0, R6, R7
        BLT MAIN
        BGT HERE
        STR R1, [R4, #110]
        B MAIN
HERE    STR R6, [R4, #110]
```

Machine language code of memfile2.dat:

E04F000F
E2802005
E280300C
E2437009
E1874002
E0035004
E0855004
E255500A
C2555002
E285500C
E0558007
0A00000C
E0538004
AA000000
E2805000
E0578002
B2857001
E0477002
E58370E0
E59020EC
E08FF000
E280200E
EA000001
E280200D
E280200A
E58020F8

Simulation waveform of the modified structure: