```
// ARM CPU Team + Joseph Decuir
// 2/23/24
// start: ARM_L4DP, from 2020
// add: GPIO access
// add: single step and choice
// add manual reset
// add local wire display
// add local bus display
// add local ROM replacement
// add byte access - sequential

module ARM_DE10(
   input ADC_CLK_10, MAX10_CLK1_50, MAX10_CLK2_50,
   output           [7:0]     HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
   input            [1:0]     KEY,
   output           [9:0]     LEDR,
   input            [9:0]     SW,
   inout           [35:0]     ARMGPIO  );

wire clk, MemWrite;
reg [7:0] DB, Membus;   // display data bus
// manual clock generation, using RS-NAND logic
// start by pressing KEY[1]; delay resets it
wire DCLK, NCLK, CRES;  // RSNAND outputs
reg [31:0] CDIV;  // define 32-bit counter
always @(*) begin // start always delay block
   if (DCLK==0) CDIV = 0;  // if CLK off, reset counter
   else CDIV = CDIV+1;     // otherwise count up
   end                     // end clock counter
assign CRES = CDIV[28];    // fixed delay from 50 MHz clock
assign DCLK = ~(KEY[1] & NCLK);
assign NCLK = ~(CRES & DCLK);

// switch control clock
always @(*) if(SW[9]) clk = DCLK;
   else clk =  MAX10_CLK1_50;

assign reset = ~KEY[0];    // manual reset

always @(*) if(SW[8]) DB = SW[7:0];
   else DB = Membus;

// FSM inputs: Clock, Reset, Step
// manual FSM counter Reset
// logic clk, reset, MemWrite;      // main bus clock, reset and R/W
logic [31:0] WriteData, ReadData, DataAdr;   // 3 32-bit numbers
wire [7:0] MemBus;
// assign MemBus = ARMGPIO[7:0];

// main memory test code
initial DataAdr = 0;
always @(posedge AS) begin          // loop
DataAdr <= DataAdr + 1;
if (DataAdr == 96) DataAdr <= 0;
end

// memory test logic state machine (use before ARM)
initial MemWrite = 0;         // this test is read only
reg [5:0] bus;       // use 30 states = read rom, write ram, read ram
always @(posedge clk) begin
bus = (bus + 1);     //
if (bus==30) bus = 0;
case (bus)
// read from ROM
0: begin AS=0; RD=0; WR=0; WE=0; ROM <= 1; RAM <= 0; end
1: AS=1;    // assert Address strobe
2: RD=1;    // assert Read
9: MemBus <= ARMGPIO[7:0]; // capture 1st byte
// write to RAM
10: begin AS=0; RD=0; WR=0; WE=0; ROM <=0; RAM <=1; end
11: AS=1;      // assert Address strobe
12: WR=1;      // assert Read
19: ARMGPIO[7:0] <= MemBus;
```

```
// read from RAM
20: begin AS=0; RD=0; WR=0; WE=0; ROM <=0; RAM <=1; end
21: AS=1;      // assert Address strobe
22: RD=1;      // assert Read
29: MemBus <= ARMGPIO[7:0];
endcase
end

logic ROM, RAM, RD, WR, WE, AS;     // chip selects, data strobes, address strobe

assign ARMGPIO[35] = clk;
//    ARMBusGPIO[34] not define yet
assign ARMGPIO[33] = ~AS;  // AS, active low
// reserved for Reset button
assign ARMGPIO[32] = DataAdr[16];      // A16
// assign reset = ~ARMGPIO[31];  // RES, active low
assign ARMGPIO[30] = ~WE;  // WE, active low
assign ARMGPIO[29] = ~ROM;
assign ARMGPIO[28] = ~RAM;
assign ARMGPIO[27] = ~RD;  // RD, active low
assign ARMGPIO[26] = ~WR;  // WR, active low
//    ARMBusGPIO[25:24] not defined yet
assign ARMGPIO[23:8] = DataAdr[15:0];  // A15-A0
assign ARMGPIO[7:0] = 8'bZ;   // set Data bus to high Z, for reads
//always @(posedge AS) MemBus = ARMGPIO[7:0];

// display AB15-AB0, DB7-DB0 on Hex digits
seg7 D5(DataAdr[15:12], HEX5);// A15-A12
seg7 D4(DataAdr[11:8], HEX4); // A11-A8
seg7 D3(DataAdr[7:4], HEX3);  // A7-A4
seg7 D2(DataAdr[3:0], HEX2);  // A3-A0
seg7 D1(DB[7:4], HEX1); // D7-D4
seg7 D0(DB[3:0], HEX0); // D3-D0

assign LEDR[9] = clk;
assign LEDR[8] = AS;
assign LEDR[7] = WR;
assign LEDR[6] = RD;
assign LEDR[5] = ROM;
assign LEDR[4] = RAM;

endmodule // ARM_DE10


// original ARM_L4DP module
// eventually substitute for MemTest

module arm_L4DP(input  logic        clk, reset,    // rename from top
           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

// replace imem & dmem with phyical memory access (above)

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

// replace imem & dmem with phyical memory access (above)

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; // ALUControl was 2 bits
  logic [2:0] ALUControl;     // changed to 3 bits

  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,       // change to 3 bit
                  output logic         MemWrite, MemtoReg,
                  output logic         PCSrc);

  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);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite);
endmodule

module decode(input  logic [1:0] Op,
              input  logic [5:0] Funct,
              input  logic [3:0] Rd,
              output logic [1:0] FlagW,   // this needs to decode these
              output logic PCS, RegW, MemW,
              output logic MemtoReg, ALUSrc,
              output logic [1:0] ImmSrc, RegSrc,
              output logic [2:0] ALUControl);      // change to 3 bits

  logic [9:0] controls;
  logic       Branch, ALUOp, Test;  // Branch, ALUOp & Test
  assign Test = (Funct[4]&~Funct[3]);     // 4 test instructions

  // Main Decoder

  always_comb        // BEE425 project teams need to change here
   casex(Op)
                           // Test must inhibit controls[3] = RegW
     2'b00:                // Data processing immediate
            if (Funct[5])  controls = {6'b000010, ~Test, 3'b001};
                           // Data processing register
            else           controls = {6'b000000, ~Test, 3'b001};
                           // 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              expanded with new DP instructions
  always_comb
    if (ALUOp) begin                     // still needs FlagW setting
      case(Funct[4:1])
       4'b0000: ALUControl = 3'b010; // AND
       4'b0001: ALUControl = 3'b100; // EOR
       4'b0010: ALUControl = 3'b001; // SUB
       4'b0011: ALUControl = 3'bx;   // RSB not supported yet
       4'b0100: ALUControl = 3'b000; // ADD
       4'b0101: ALUControl = 3'b000; // ADC (need to enable CI)
       4'b0110: ALUControl = 3'b011; // SBC (need to enable CI)
       4'b0111: ALUControl = 3'bx;   // RSC not supported yet
       4'b1000: ALUControl = 3'b010; // TST does AND, write flags
       4'b1001: ALUControl = 3'b100; // TEQ does EOR, write flags
       4'b1010: ALUControl = 3'b001; // CMP does SUB, write flags
       4'b1011: ALUControl = 3'b000; // CMN does ADD, write flags
       4'b1100: ALUControl = 3'b011; // ORR
       4'b1101: ALUControl = 3'b101; // MOV sets ALU passthrough B, C
       4'b1110: ALUControl = 3'bx;   // BTC not supported
       4'b1111: ALUControl = 3'bx;   // MVN not supported
      endcase
      // update flags if S bit is set or Test instructions
   // (C & V only updated for arith or shift instructions)
      FlagW[1] = Funct[0] | Test; // FlagW[1] = S-bit or Test
   // FlagW[0] = S-bit & (ADD | SUB | MOV/shift)
      FlagW[0] = Test | (Funct[0] &       // ADD, SUB or MOV/Shift
        (ALUControl==3'b000 | ALUControl==3'b001 | ALUControl==3'b101));
    end else begin         // not ALUOp
      ALUControl = 3'b000; // add for non-DP instructions
      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 [2:0]  ALUControl, // change to 3 bit
                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 [31:0] ShiftData;     // new intermediate from shift module
  logic [3:0]  RA1, RA2;
  logic CI, CO;      // carry in and carry out from shift module

  // 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);

  // insert shift module here, intercepting WriteData -> ShiftData
  assign CI = ALUFlags[1];       // preset CI to existing C flag
  Shift       shift(Instr[11:4], WriteData, ShiftData,
               CI, CO); // add CI & CO. CO passes to ALU module

  // ALU logic
  mux2 #(32)  srcbmux(ShiftData, ExtImm, ALUSrc, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, CO,    // added from Shift
                  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 alu (input logic [31:0] a, b,
            input logic [2:0] ALUC,
            input logic CI,    // added
            output logic [31:0] Result,
            output logic [3:0] ALUF);

   logic c_out, math, mov;    // carry out, math op, mov op
   logic [31:0] sum;
   assign math = ~(ALUC[1] | ALUC[2]); // math = ADD or SUB
   assign mov = ALUC[2] & ~ALUC[1] & ALUC[2];   // decode MOV

   always_comb begin
      {c_out,sum} = a + (ALUC[0] ? ~b:b) + ALUC[0];  // sum
      casex(ALUC)
         3'b00x: Result = sum; //ADD or SUB
         3'b010: Result = a&b; //AND
         3'b011: Result = a|b; //OR
         3'b100: Result = a^b; //EOR
         3'b101: Result = b;   //MOV
         default: Result = 0;
      endcase
   end
   // ALUFlags -- gives information about the properties of the result
   assign ALUF[3] = Result[31];
   assign ALUF[2] = &(~Result);                      // investigate bug here
   assign ALUF[1] = (c_out & math) | (CI & mov);    // CO = math or mov
   assign ALUF[0] = ~(ALUC[0] ^ a[31] ^ b[31]) & (sum[31] ^ a[31]) & ~ALUC[1];
endmodule      // end alu module

module Shift(input logic [11:4] Inst,  // upper 8 bits of Src2 field
             input logic [31:0] RD2,    // from WriteData
             output logic [31:0] SHO,   // shift out to srcbmux
             input logic CI,            // maps from ALUFlags[1]
             output logic CO);          // maps to ALUFlags[1]

logic [31:0] SHE, SHI;  // shift extension and shift intermediate
logic [4:0] shamt5;  // 5 bit shift amount, instr[11:7]
logic [1:0] sh;      // 2 bit shift type, instr[6:5]
logic SS;            // 1 bit shift source, instr[4] (0)
assign {shamt5, sh, SS} = Inst[11:4];  // unpack instruction bits

always @(*) begin
      case (sh)            // decode sh bits
      2'b00:   if (shamt5==0) begin
               SHO <= RD2; CO <= 0; end   // MOV
               else begin  SHI <= 0;      // LSL
               {SHE, SHO} <= ({SHI, RD2} << shamt5);
               CO <= SHE[0];  // clip SHE LSB as carry out
               end
      2'b01:   begin    SHI <=0;        // LSR
               {SHE, SHO, CO} <= ({SHI, RD2, CI} >> shamt5);
               end                 // end LSR
      2'b10:   begin               // ASR
               if (RD2[31]==1)   SHI <= -1;     // sign extend
               else SHI <=0;
               {SHE, SHO, CO} <= ({SHI, RD2, CI} >> shamt5);
               end                 // end ASR
      2'b11:   if (shamt5==0) // RRX
               begin
               {SHO, CO} <= {CI, RD2};
               end                 // end RRX
               else  begin         // ROR
               {SHI, SHE, CO} <= ({RD2, RD2, CI} >> shamt5);
               SHO <= SHI | SHE; // recombine two parts of Rotated number
               end                 // end ROR
      endcase  // end decoding sh bits
   end         // end always
endmodule      // end shift module

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 seg7(input [3:0] hex, output [7:0] segment);
reg [7:0] leds;
always@(*) begin
case(hex)
   0: leds =  8'b00111111; // 0 image
   1: leds =  8'b00000110; // 1 image
   2: leds =  8'b01011011; // 2 image
   3: leds =  8'b01001111; // 3 image
   4: leds =  8'b01100110; // 4 image
   5: leds =  8'b01101101; // 5 image
   6: leds =  8'b01111101; // 6 image
   7: leds =  8'b00000111; // 7 image
   8: leds =  8'b01111111; // 8 image
   9: leds =  8'b01101111; // 9 image
   10: leds = 8'b01110111; // A image
   11: leds = 8'b01111100; // b image
   12: leds = 8'b00111001; // C image
   13: leds = 8'b01011110; // d image
   14: leds = 8'b01111001; // E image
   15: leds = 8'b01110001; // F image
   endcase
   end
   assign segment = ~leds;    // invert and copy to outputs
endmodule      // end of seg7
```