Mini S08

ECE 643 Final Project

Dan Wagner
Kansas State University
College of Engineering
Department of Electrical and Computer Engineering

Dr. John Devore
Professor
Department of Electrical and Computer Engineering

December 8, 2017

Introduction

The Mini S08 project was designed to emulate a subset of instructions from the Freescale S08 microcontroller on an Altera DE2 FPGA development board. The subset was thorough and allowed the Mini S08 to function exactly like the microcontroller except for a few instructions (such as the cmp instruction) in order to keep the project simplistic. The Mini S08 was constrained to be coded in either the Verilog or AHDL languages. The seven segment displays on the board were also required to be used for displaying the Instruction Register, Data Bus, and Address Bus values.

A Floating Point Unit (FPU) was incorporated into the project to perform two operations: division and multiplication. The Mini S08 was able to take two IEEE 754 Floating Point numbers and perform a division and/or multiplication on them. The FPU was built upon the project's previous functionality, as the core framework was completed.

IEEE 754 Standard

The IEEE 754 Floating Point Standard [1] specifies the format of 32-bit floating point numbers. Each number adheres to the following format:

1. The most significant bit is the sign bit which is true when the number is negative.
2. The next eight bits are the biased exponent bits. The bias is 127.
3. The remaining 23 bits are the normalized fractional bits of the number.

When performing arithmetic, the end result may require rounding. Two main methods were investigated: round to even and round to odd. In round to even, the number is rounded as to make the least significant bit a zero (rounds to the nearest even number). If round to odd is used, the least significant bit of the number is made to be a one (rounds to the nearest odd number).

Data Flow

The data flow consists of several components and can be seen in the following figure.

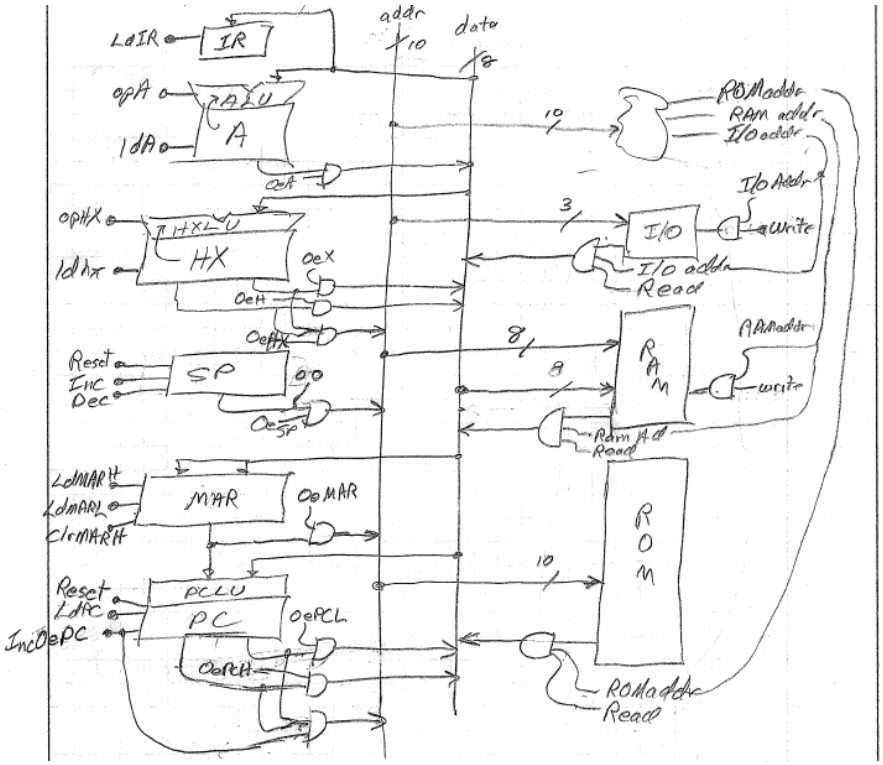

Figure 1: Data Flow of the Mini S08 Project (excluding FPU).

The Mini S08 contains two buses for information: an 8-bit data bus and a 10-bit address bus. The address bus is ten bits to accommodate the ROM memory space size. Information on the address bus pertains to memory locations where the program or registers need to access in order to retrieve the required information. Data bus accesses are for transporting pieces of data found at these locations, or directly given (in the case of Immediate addressing).

Each register obtains their values from the data bus in different forms. The accumulator (A) is fed its value from the arithmetic and logic unit (ALU), which gets its value from the bus. The H:X register pair obtains its value in a similar fashion, as does the Program Counter (PC). Finally, the Memory Address Register (MAR) pulls its data directly from the bus.

Interaction with the address bus are more complicated. The information placed on this bus depends upon the addressing mode and the specific instruction. In Indexed addressing, the data the instruction needs is located in the memory location denoted by the HX register; thus, the HX register is put onto the address bus. Immediate and Relative addressing access the memory location pointed at by the PC, while Direct accesses the location pointed at by the MAR. Stack addressing and the rts instruction retrieves data from the location pointed at by the Stack Pointer (SP).

Input and output are coordinated between the buses and four modules: FPU, RAM, ROM, and SCI. In Figure 1, the FPU is not shown, and the SCI module is renamed I/O. Data flows to these modules from the address bus (accessing these modules requires addressing each). These modules transfer information to the data bus to be used by the program.

The control signals coordinating these transactions will be discussed in the next section.

Control Units

The Mini S08's operation was modeled as a state machine with five states. These states are shown in Figure 2.

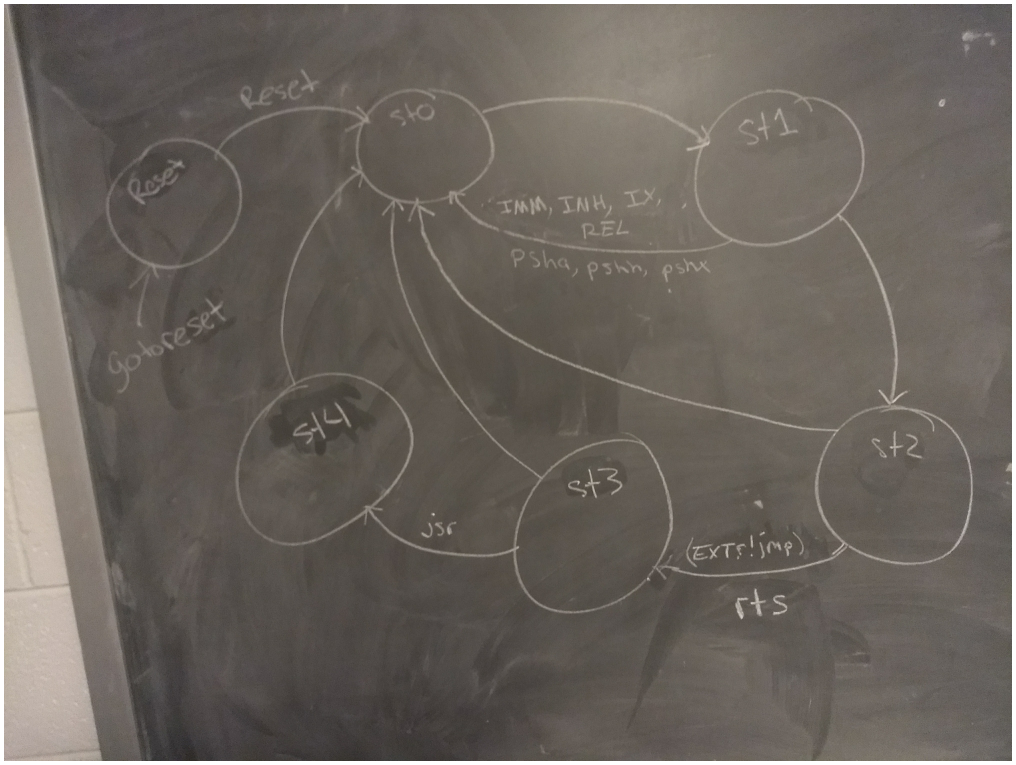

Figure 2: State Machine for the Mini S08 Project.

Reset is the initial state that the MiniS08 is in when programmed. When the reset push button is pressed, the program returns to this state and transitions to state zero. State zero is where the instruction is fetched into the IR. After fetching, the next state is state one.

State one decodes the fetched instruction and transitions to different states depending on the address mode. If the instruction is an Immediate, Inherent, Indexed, Relative, or one of the push instructions, then the instruction is completed in one cycle and the system transitions to the state zero. Otherwise, additional cycles are required for complete the execution and flow transitions to state two.

State two continues executing instructions that failed to take one cycle to complete. If the instruction is an Extended or jmp instruction, or an rts instruction, then another state is required to complete it (state three). Otherwise, the instruction finishes and the system transitions back to state zero.

State three, similar to state two, resumes instruction execution that failed to complete in the prior state(s). Most instructions finish executing here, but jsr requires transitioning to the final state.

State four is required for the jsr instruction, as it requires four cycles to complete. After completion, the system transitions back to state zero.

The diagram in Figure 2 fulfilled the operating requirements for the S08 without the FPU. When the FPU was added to the project, two additional state diagrams were required: one for multiplication, and one for addition. Shown below, the multiply state machine required three states depending on the operation's result.

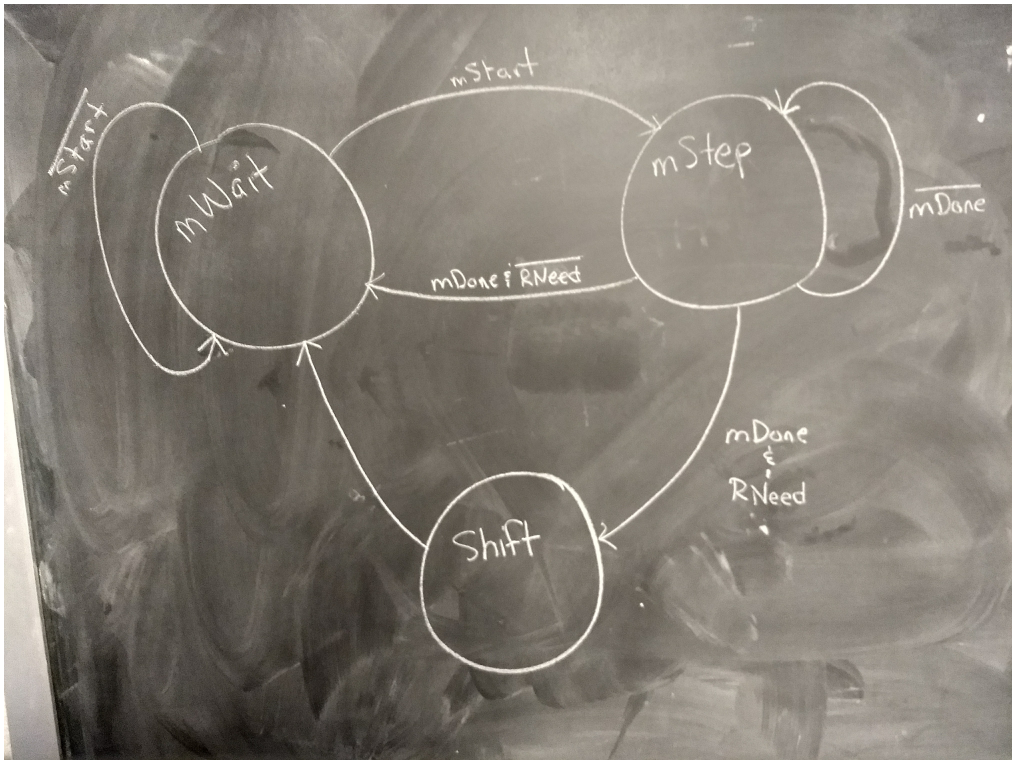

Figure 3: Multiply State Machine for the Mini S08 Project.

The multiply state machine begins in the waiting state. The FPU waits for all data to be input from the S08 before beginning the operation. The FPU receives both numbers to multiply and the operation to perform from the S08 data bus. Once all information is received, the start signal enables and the system transitions to the step state.

Most of the work is done within the step state. On each iteration, a portion of the multiplication is performed (mainly, shifting and adding). The system remains in this state until the multiplication operation is complete. If rounding is required, the system transitions to the shift state and shifts the product right by one bit. Otherwise, it returns to the waiting state and the cycle begins anew.

The division state machine is simpler than either the multiplication or S08 machines. It consists of two states, as shown in Figure 4.

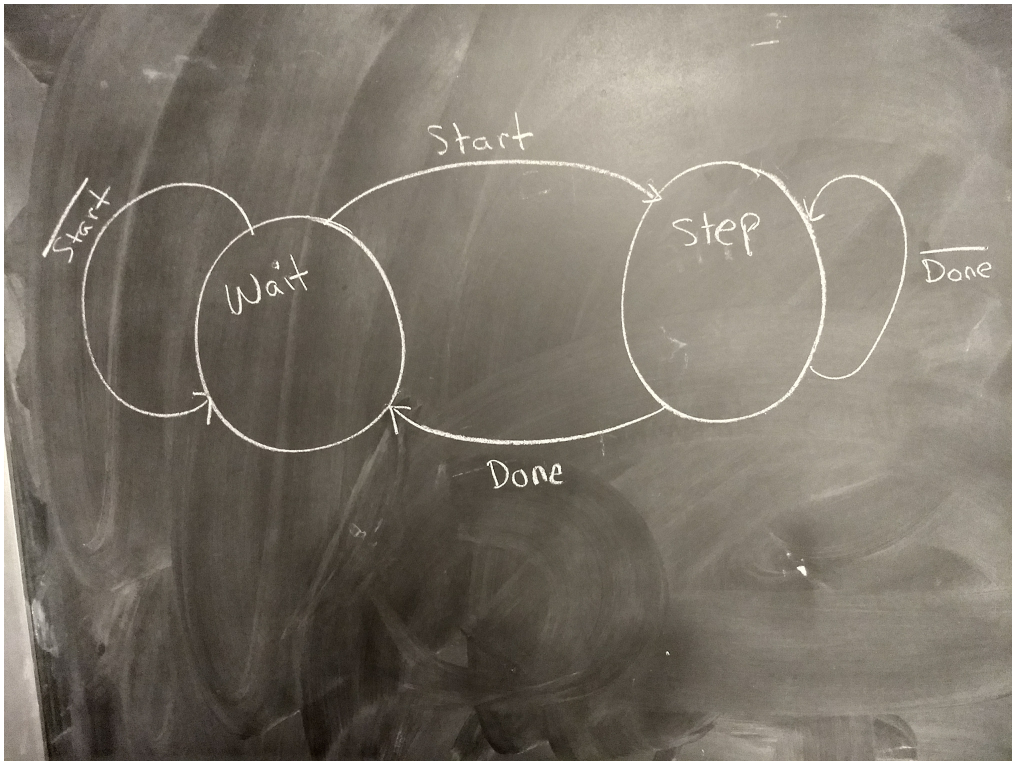

Figure 4: Divide State Machine for the Mini S08 Project.

The system remains in the waiting state like the multiply state. It waits for both floating point values and the operation to be entered. Once these requirements are met, the system transitions into the step state.

Very similarly to the multiplication operation, the step state is the main working state for division. On each iteration, a portion of the operation is completed. This state remains the active state until the division is completed. At this point, any required rounding is performed and the system transitions back to the waiting state.

Conclusion

The project was an overall success, as seen in the Figures 5-8.

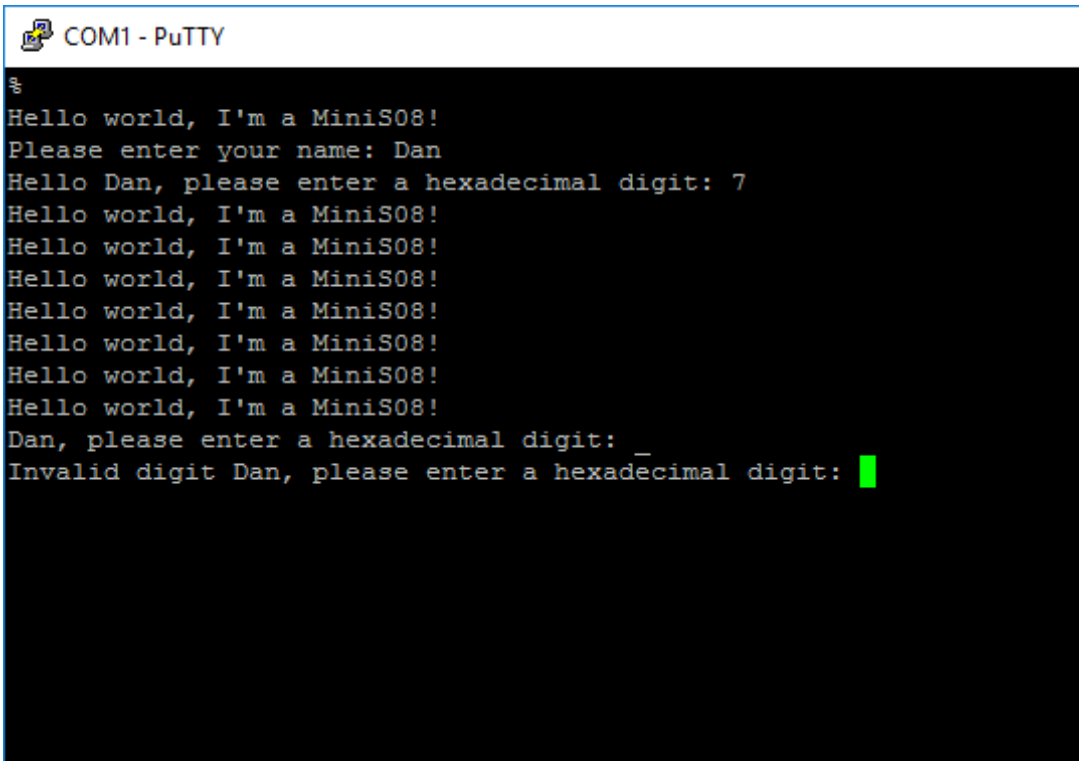

Figure 5: Proof of Mini S08 Functionality.

Initially, the project was daunting and required more work than thought. Writing the register transfer notation (RTN) statements was simple from the data flow diagram provided. Programming with Verilog and translating those statements proved another challenge, as the Altera Quartus II software enjoyed automatically optimizing code fragments away. These optimizations resulted in bugs when testing and were usually resolved by fixing a variable declaration or logic error.

The project is an excellent example of the overall design process of a computer. First, we started with the data flow and configuring how to accomplish and coordinate the data lines. From there, we developed the RTN. Then, we translated the statements into Verilog and debugged the result. Finally, we had our working computer.

The only major change I would recommend is moving away from the Quartus II IDE. It is not very useful with a majority of the bugs that we were facing (auto-optimization) and was not helpful when debugging.

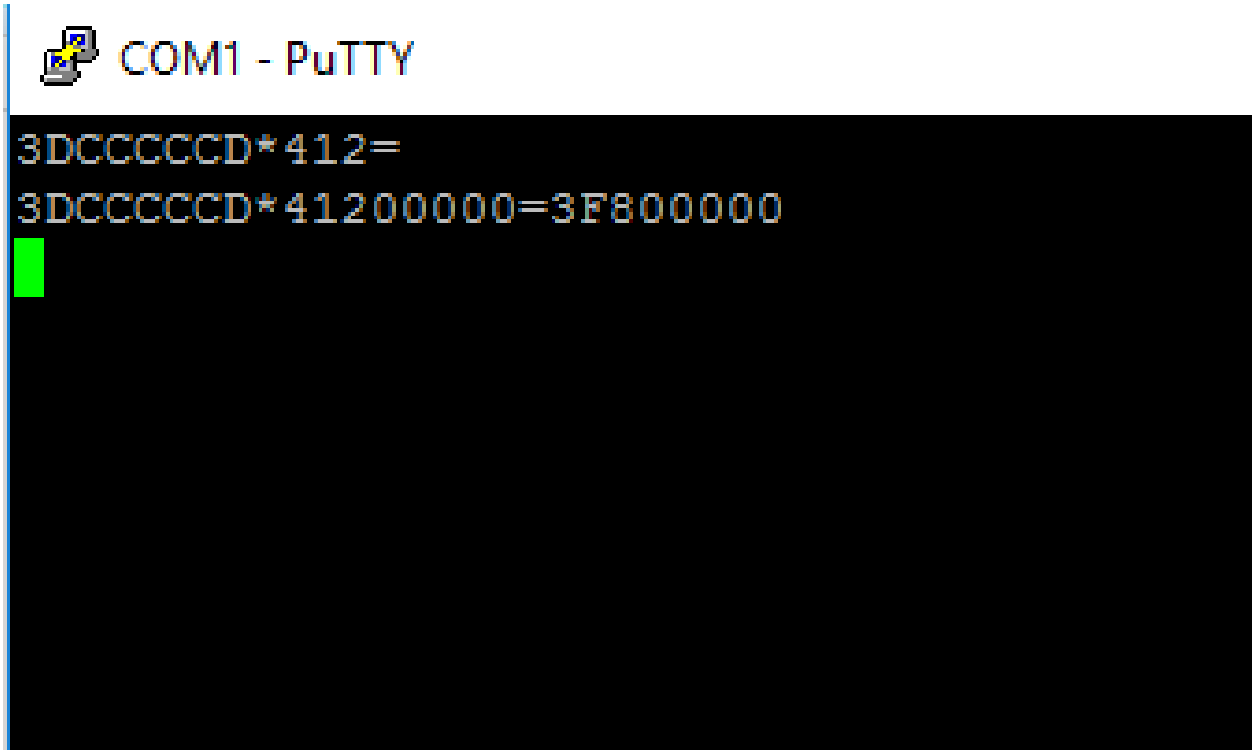

Figure 6: Proof of Mini S08 Multiplication.

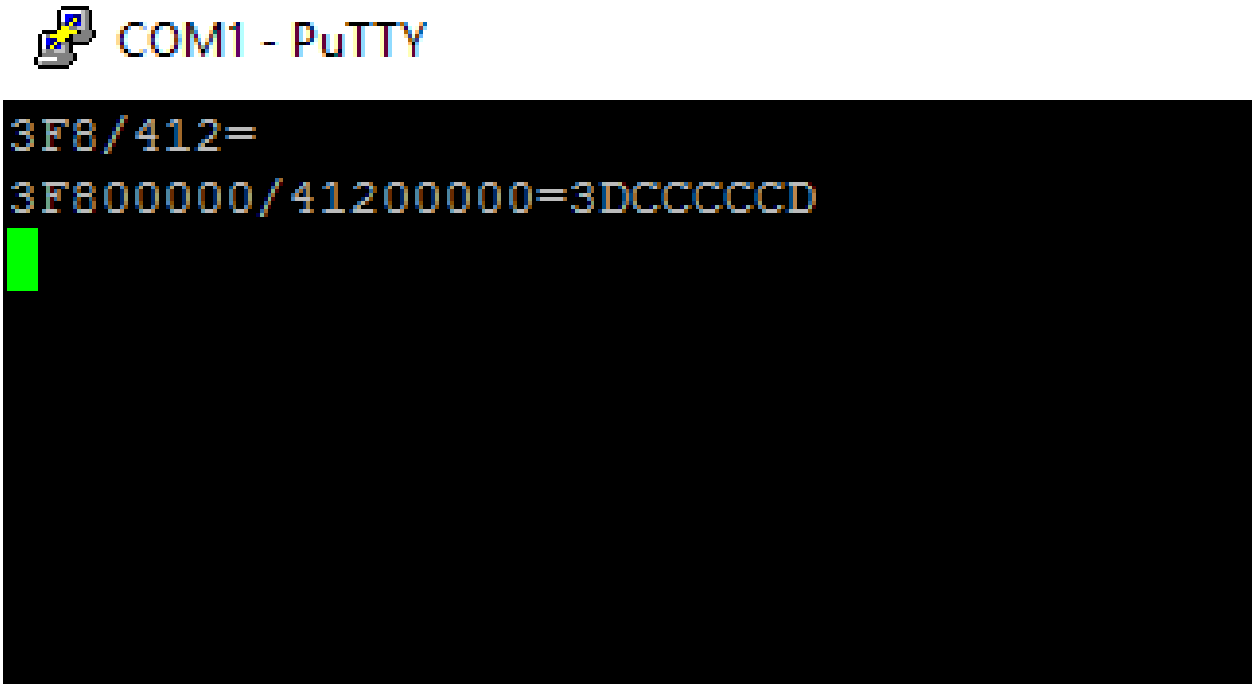

Figure 7: Proof of Mini S08 Division.

References

1. IEEE Standard for Binary Floating-Point Arithmetic IEEE Std. 754-2008, 2008.

Source Code: S08

```
 // MiniS08 starter file
module minis08(clk50,rxd,resetin,pbin,clksel,clkdisp,txd,addr,data,stateout,IRout);
input clk50, rxd, resetin, pbin;
input [2:0] clksel;
output txd;
output [20:0] addr;  // three 7-segment displays
output [13:0] data;  // two 7-segment displays
output [13:0] IRout;
output [6:0] stateout;
wire ldIR, ldA, ldHX, ldMARH, ldMARL, clrMARH;
wire ldPC, oeIncPC, oePCH, oePCL;
wire br, jmp, jsr, rts;
wire Imm, Ix, Inh, Stk, Dir, Ext, Rel;
wire st0, st1, st2, st3, st4;
wire N, Z;
wire oeA, oeH, oeX, oeHX, oeMAR;
wire incSP, decSP, oeSP;
wire modA, stoA, modHX;
wire pula, psha, pulx, pshx, pulh, pshh, aix, add, sub, And, ora, eor;
wire lsra, asra, lsla, lda, ldx, sta, stx;
wire bra, bcc, bcs, bpl, bmi, bne, beq, BCT;
wire Read, Write;
wire IOaddr, RAMaddr, ROMaddr, FPUaddr;
wire [7:0] sciout;
wire [7:0] FPUout;
wire [7:0] ROMout, RAMout;
wire [9:0] abus;
wire [7:0] dbus;
wire Reset, Co;
wire [9:0] PCLU, HXLU;
wire [8:0] ALU;
reg [2:0] CPUstate;
reg [27:0] clkdiv;
reg resetout, gotoreset, pbout, pbreg, S08clk, C;
output clkdisp;
reg [7:0] A, SP, IR;
reg [9:0] HX, PC, MAR;

sevenseg A2({2'd0,abus[9:8]},addr[20:14]);
sevenseg A1(abus[7:4],addr[13:7]);
sevenseg A0(abus[3:0],addr[6:0]);
sevenseg D1(dbus[7:4],data[13:7]);
sevenseg D0(dbus[3:0],data[6:0]);
```

```
//sevenseg D1(A[7:4],data[13:7]);     //***** debugging
//sevenseg D0(A[3:0],data[6:0]);      //***** debugging

sevenseg IR1(IR[7:4], IRout[13:7]);
sevenseg IR0(IR[3:0], IRout[6:0]);
//sevenseg IR1(SP[7:0], IRout[13:7]);     //***** debugging
//sevenseg IR0(SP[3:0], IRout[6:0]);      //***** debugging
//sevenseg IR1(0, IRout[13:7]);            //***** debugging
//sevenseg IR1({2'b0,HX[9:8]}, IRout[13:7]);//***** debugging
sevenseg ST({1'b0,CPUstate}-1,stateout);

sci S08sci(clk50, dbus, sciout, IOaddr, abus[2:0], Read, Write, rxd, txd);
S08ram S08ram(abus[7:0],clk50,dbus,Write&RAMaddr,RAMout);
S08rom S08rom(abus,clk50,ROMout);
fpu S08fpu(clk50, dbus, FPUout, FPUaddr, abus[1:0], Read, Write);

assign clkdisp = clksel==0 ? pbreg : clksel==1 ? clkdiv[27]
   : clksel==2 ? clkdiv[24] : clksel==3 ? clkdiv[21]
           : clksel==4 ? clkdiv[17] : clksel==5 ? clkdiv[15]
   : clksel==6 ? clkdiv[13] : clkdiv[2];

always @(posedge clk50)
   begin
   clkdiv <= clkdiv+1;
   gotoreset <= resetout;
   resetout <= ~resetin;
   pbreg <= pbout;
   pbout <= ~pbin;
   S08clk <= clkdisp;
   end

always @(posedge S08clk)
   begin
   A <= ldA ? ALU[7:0] : A;
   HX <= ldHX ? HXLU : HX;
   SP <= Reset ? 'hFF : incSP  ? (SP + 1) : decSP ? (SP - 1) : SP;
   PC <= Reset ? 'h100 : ldPC ? PCLU : oeIncPC ? PC+1 : PC;
   IR <= ldIR ? dbus : IR;
   C <= ldA&(add|sub|lsla|lsra|asra) ? Co : C;
   // 2'X0, where X is any base (doesn't matter)
   MAR <= ldMARL&clrMARH ? {2'h0, dbus[7:0]} : ldMARH ? {dbus[1:0], MAR[7:0]}
: ldMARL ? {MAR[9:8], dbus[7:0]} : clrMARH ? {2'h0, MAR[7:0]} : MAR;

   CPUstate <= gotoreset ? 0 : Reset ? 1 : st0 ? 2
             : st1 ? ((Inh|Imm|Ix|Rel|psha|pshh|pshx) ? 1: 3)
```

```
            : st2 ? (((Ext&!jmp)|rts) ? 4 : 1)
            : st3 ? (jsr ? 5 : 1)
 : st4 ? 1 : st0;
   end

assign Reset = CPUstate==0;
assign st0 = CPUstate==1;
assign st1 = CPUstate==2;
assign st2 = CPUstate==3;
assign st3 = CPUstate==4;
assign st4 = CPUstate==5;

assign N = A[7];
assign Z = A==0;
assign ALU = lda ? dbus : pula ? dbus : add ? ({1'b0, A} + {1'b0, dbus})
: sub ? ({1'b0, A} - {1'b0, dbus}) : And ? (A & dbus) : ora ? A | dbus
: eor ? A ^ dbus : lsra ? ({A[0], A >> 1}) : asra ? ({A[0], A >>> 1})
: lsla ? ({A[7], A << 1}) : 0;

assign Co = ALU[8];

assign HXLU = ldx ? {HX[9:8], dbus} : pulx ? {HX[9:8], dbus}
    : pulh ? {dbus[1:0], HX[7:0]} : aix ? (HX + {dbus[7], dbus[7], dbus}) : 0;

assign PCLU =  BCT ? (PC+{dbus[7], dbus[7], dbus}+1)
: jsr ? MAR
: {MAR[9:8], dbus};

assign abus = oeHX ? HX : oeSP ? {2'h0, SP[7:0]} : oeMAR ? MAR : oeIncPC ? PC : 0;
assign dbus = oeA ? A : oeX ? HX[7:0] : oeH ? HX[9:8] : oePCL ? PC[7:0]
: oePCH ? PC[9:8] : Read&IOaddr ? sciout
: Read&ROMaddr ? ROMout : Read&RAMaddr ? RAMout
: Read&FPUaddr ? FPUout : 0;

assign ldIR = st0;
assign ldA = modA&Inh&st1 | modA&Stk&st2 | modA&Imm&st1
| modA&Dir&st2 | modA&Ext&st3 | modA&Ix&st1;
assign oeA = stoA&Stk&st1 | stoA&Dir&st2 | stoA&Ext&st3 | stoA&Ix&st1;
assign ldHX = modHX&Stk&st2 | modHX&Imm&st1 | modHX&Dir&st2
| modHX&Ext&st3 | modHX&Stk&Ix;
assign oeH = pshh&st1;
assign oeX = pshx&st1 | stx&Dir&st2 | stx&Ext&st3 | stx&Ix&st1;
assign oeHX = modA&Ix&st1 | stoA&Ix&st1 | modHX&Ix&st1 | stx&Ix&st1;
assign incSP = modA&Stk&st1 | modHX&Stk&st1 | rts&st1 | rts&st2;
assign decSP = stoA&Stk&st1 | stx&Stk&st1 | pshx&st1 | pshh&st1 |  jsr&st3 | jsr&st4;
```

```
assign oeSP = decSP | modA&Stk&st2 | modHX&Stk&st2 | rts&st2 | rts&st3;
assign ldMARH = modA&Ext&st1 | stoA&Ext&st1 | modHX&Ext&st1
| stx&Ext&st1 | jmp&st1 | jsr&st1 | rts&st2;
assign ldMARL = modA&Dir&st1 | modA&Ext&st2 | stoA&Dir&st1
| stoA&Ext&st2 | modHX&Dir&st1 | modHX&Ext&st2
| stx&Dir&st1 | stx&Ext&st2 | jsr&st2;
assign clrMARH = modA&Dir&st1 | stoA&Dir&st1 | modHX&Dir&st1 | stx&Dir&st1;

assign oeMAR = modA&Dir&st2 | modA&Ext&st3 | | stoA&Dir&st2
| stoA&Ext&st3 | modHX&Dir&st2 | modHX&Ext&st3 | stx&Dir&st2 | stx&Ext&st3;
assign ldPC = BCT&br&st1 | jmp&st2 | jsr&st4 | rts&st3;
assign oeIncPC = st0 | modA&Imm&st1 | modA&Ext&st1 | modA&Ext&st2 | modA&Dir&st1
 | stoA&Dir&st1 | stoA&Ext&st1 | stoA&Ext&st2 | modHX&Imm&st1
 | modHX&Dir&st1 | modHX&Ext&st1 | modHX&Ext&st2 | stx&Dir&st1
 | stx&Ext&st1 | stx&Ext&st2 | br&st1 | jmp&st1
 | jmp&st2 | jsr&st1 | jsr&st2;
assign oePCH = jsr&st4;
assign oePCL = jsr&st3;
assign Write = stoA&Stk&st1 | stoA&Dir&st2 | stoA&Ext&st3 | stoA&Ix&st1
     | stx&Stk&st1 | pshx&Stk&st1 | stx&Dir&st2 | stx&Ext&st3 | stx&Ix&st1
     | pshh&Stk&st1| jsr&st3 | jsr&st4;
assign Read = st0 | modA&Stk&st2 | modA&Imm&st1 | modA&Dir&st1 | modA&Dir&st2
     | modA&Ext&st1 | modA&Ext&st2 | modA&Ext&st3 | modA&Ix&st1
     | stoA&Dir&st1 | stoA&Ext&st1 | stoA&Ext&st2 | modHX&Stk&st2
     | modHX&Imm&st1 | modHX&Dir&st1 | modHX&Dir&st2 | modHX&Ext&st1
 | modHX&Ext&st2 | modHX&Ext&st3 | modHX&Ix&st1  | stx&Dir&st1
     | stx&Ext&st1 | stx&Ext&st2 | br&st1 | jmp&st1 | jmp&st2
 | jsr&st1 | jsr&st2 | rts&st2 | rts&st3;
assign IOaddr = (abus < 'h8)&(!FPUaddr);
assign RAMaddr = !IOaddr & !ROMaddr & !FPUaddr;
assign ROMaddr = (abus > 'hFF);
assign FPUaddr = (abus < 'h4);
assign modA = lda | pula | add | sub | And | ora | eor | lsra | asra | lsla;
assign stoA = sta | psha;
assign modHX = ldx | pulx | pulh | aix;
assign br = IR[7:4] == 'h2;
assign jmp = IR=='hCC;
assign jsr = IR=='hCD;
assign rts = IR=='h81;
assign add = (IR == 'hAB) | (IR == 'hBB) | (IR == 'hCB) | (IR == 'hFB);
assign sub = (IR == 'hA0) | (IR == 'hB0) | (IR == 'hC0) | (IR == 'hF0);
assign And = (IR == 'hA4) | (IR == 'hB4) | (IR == 'hC4) | (IR == 'hF4);
assign ora = (IR == 'hAA) | (IR == 'hBA) | (IR == 'hCA) | (IR == 'hFA);
assign eor = (IR == 'hA8) | (IR == 'hB8) | (IR == 'hC8) | (IR == 'hF8);
assign lsra = (IR == 'h44);
```

```
assign asra = (IR == 'h47);
assign lsla = (IR == 'h48);
assign Imm = IR[7:4] == 'hA;
assign Ix = IR[7:4] == 'hF;
assign Inh = IR[7:4] == 'h4;
assign Rel = bra | bcc | bcs | bpl | bmi | bne | beq;
assign Dir = IR[7:4] == 'hB;
assign Ext = IR[7:4] == 'hC;
assign Stk = IR[7:4] == 'h8;
assign psha = IR=='h87;
assign pshh = IR=='h8B;
assign pshx = IR=='h89;
assign lda = (IR == 'hA6) | (IR == 'hB6) | (IR == 'hC6) | (IR == 'hF6);
assign ldx = (IR == 'hAE) | (IR == 'hBE) | (IR == 'hCE) | (IR == 'hFE);
assign pula = IR=='h86;
assign pulx = IR=='h88;
assign pulh = IR=='h8A;
assign aix = IR=='hAF;
assign BCT = bra | bcc&!C | bcs&C | bpl&!N | bmi&N | bne&!Z | beq&Z;
assign bra = IR=='h20;
assign bcc = IR=='h24;
assign bcs = IR=='h25;
assign bpl = IR=='h2A;
assign bmi = IR=='h2B;
assign bne = IR=='h26;
assign beq = IR=='h27;
assign sta = (IR == 'hB7) | (IR == 'hC7) | (IR == 'hF7);
assign stx = (IR == 'hBF) | (IR == 'hCF) | (IR == 'hFF);

endmodule

module sevenseg(hex, segs);
input [3:0] hex;
output [6:0] segs;
assign segs = hex==0  ? 'b0000001   // active-low segments
            : hex==1  ? 'b1001111
            : hex==2  ? 'b0010010
            : hex==3  ? 'b0000110
            : hex==4  ? 'b1001100
            : hex==5  ? 'b0100100
            : hex==6  ? 'b0100000
            : hex==7  ? 'b0001111
            : hex==8  ? 'b0000000
            : hex==9  ? 'b0001100
            : hex==10 ? 'b0001000
```

```
            : hex==11 ? 'b1100000
            : hex==12 ? 'b0110001
            : hex==13 ? 'b1000010
            : hex==14 ? 'b0110000
            :           'b0111000;
endmodule
```

Source Code: FPU

```
// starter file for the FPU.  It handles all the reading of and writing to registers
module fpu(clk, datain, dataout, FPUsel, addr, read, write);
input clk, FPUsel, read, write;
input [7:0] datain; // command or value being sent to the FPU
                    // commands are 1 Set Y as next value
                    //              2 Set X as next value
                    //              3 Perform division
                    //              4 Perform multiplication
input [1:0] addr;   // status read 00, result read 01, command write 10, value write 11
output [7:0] dataout;  // result being read from the FPU
reg [2:0] inloc;    // index in the YX set for next incoming value
reg [1:0] outloc;   // index in the Res reg for next result out (on read)
reg [31:0] Y,X,Res;
reg prevreadval, prevwritecmd, prevwriteval;
wire readval, readstatus, writecmd, writeval;
wire readvalnegedge, writecmdposedge, writevalnegedge;
assign readstatus = read & FPUsel & (addr==0);
assign readval = read & FPUsel & (addr==1);
assign writecmd = write & FPUsel & (addr==2);
assign writeval = write & FPUsel & (addr==3);

// Division wires, registers, vectors
wire Done, Rneed, Wait, Step, Start, Neg;
reg S, DivDone, DivState;
wire [24:0] AmB;
reg [9:0] E;
reg [24:0] A, B;
reg [23:0] Q;

// Multiplication wires, registers, vectors
wire mDone, mRneed, mWait, mStep, mShift, mStart, Sneed;
reg mS, MulDone, MulState, Round, SBit;
reg [9:0] mE;
reg [23:0] mA, mB;
reg [24:0] P;

always @(posedge clk)
   begin
      prevreadval <= readval;
      prevwritecmd <= writecmd;
      prevwriteval <= writeval;
      inloc <= writevalnegedge ? inloc+1 : writecmdposedge&datain==1 ? 0
: writecmdposedge&datain==2 ? 4 : inloc;
```

```
      outloc <= DivDone | MulDone ? 0 : readvalnegedge ? outloc+1 : outloc;
      Y[31:24] <= writeval&inloc==0 ? datain : Y[31:24];
      Y[23:16] <= writeval&inloc==0 ? 0 : writeval&inloc==1 ? datain : Y[23:16];
      Y[15:8] <= writeval&inloc==0 ? 0 : writeval&inloc==2 ? datain : Y[15:8];
      Y[7:0] <= writeval&inloc==0 ? 0 : writeval&inloc==3 ? datain : Y[7:0];
      X[31:24] <= writeval&inloc==4 ? datain : X[31:24];
      X[23:16] <= writeval&inloc==4 ? 0 : writeval&inloc==5 ? datain : X[23:16];
      X[15:8] <= writeval&inloc==4 ? 0 : writeval&inloc==6 ? datain : X[15:8];
      X[7:0] <= writeval&inloc==4 ? 0 : writeval&inloc==7 ? datain : X[7:0];
      Res <= DivDone ? {S, E[7:0], Q[22:0]} : MulDone ? {mS, mE[7:0], P[22:0]} : Res;
   end

assign dataout = readval&outloc==0 ? Res[31:24]
               : readval&outloc==1 ? Res[23:16]
               : readval&outloc==2 ? Res[15:8]
               : readval&outloc==3 ? Res[7:0]
               : readstatus ? {~Wait|~mWait,7'd0}  // msb is a FPU busy bit
               : 0;
// we want to react to each cmd immediately, but only once so need a pos edge
assign writecmdposedge = writecmd & ~prevwritecmd; // pos edge
// we need to advance the result index at end of a reading a value so need neg edge
assign readvalnegedge = ~readval & prevreadval;    // neg edge
// we need to advance the XY index at end of a write value so need neg edge
assign writevalnegedge = ~writeval & prevwriteval; // neg edge

// FP Divide section
// student needs to declare Done, Rneed, Wait, Step, Start, Neg
//                          A, B, AmB, E, Q, S, DivDone, DivState
// some are wires others regs, some scalars others vectors
assign Wait = ~DivState;
assign Step = DivState;
assign Start = writecmdposedge&datain==3;
always @(posedge clk)
   begin
      DivState <= writecmdposedge&datain==1 ? 0 : Wait&!Start ? 0
: Step&Done ? 0 : Wait&Start ? 1 : DivState;
      S <= Wait&Start ? Y[31] ^ X[31] : S;
      A <= Wait&Start ? {1'b0, 1'b1, Y[22:0]} : Step&!Neg ? {AmB[23:0], 1'b0}
: Step&Neg ? {A[23:0], 1'b0} : A; // CHECK THIS
      B <= Wait&Start ? {1'b0, 1'b1, X[22:0]} : B;
      E <= Wait&Start ? Y[30:23] - X[30:23] + 127 + 24 : Step&!Done ? E - 1 : E;
      Q <= Wait&Start ? 0 : Step&!Done ? {Q[22:0], !Neg}
: Step&Done&Rneed ? Q+Rneed : Q;
      DivDone <= Step&Done;
    end
```

```
assign Done = Q[23];
assign AmB = A - B;
assign Neg = AmB[24];
assign Rneed = (A>B) | (A==B)&Q[0];


// FP Multiply section
// Even though several Vectors are 1-bit different in size than the Divide, it would
//    be possible to share them.  However, I think the benefit of keeping them separate
//    far outweighs any savings.  Therefore, below I have placed an m in front of any
//    variable named the same in my multiply and divide write-ups.
// student needs to declare mDone, mRneed, mWait, mStep, mShift, mStart, Sneed,
//                          mA, mB, mE, P, mS, MulDone, MulState, Round, SBit
// some are wires others regs, some scalors others vectors

assign mWait = MulState == 0;
assign mStep =  MulState == 1;
assign mShift = MulState == 2;
assign mStart = writecmdposedge&datain==4;
always @(posedge clk)
   begin
      MulState <= writecmdposedge&datain==1 ? 0 : mWait&mStart ? 1
: mStep&mDone ? mRneed ? 2 : 0 : mShift ? 0 : MulState;
      mS <= mWait&mStart ? Y[31] ^ X[31] : mS;
      mA <= Wait&mStart ? {1'b1, X[22:0]} : mStep ? {1'b0, mA[23:1]} : mA;
      mB <= Wait&mStart ? {1'b1, Y[22:0]} : mB;
      mE <= mWait&mStart ? X[30:23] + Y[30:23] - 127 - 24 : mStep&!mDone ? mE + 1
: mShift&Sneed ? mE + 1 : mE;  // 24 is the # of bits of mA
      P <= mWait&mStart ? 0 : mStep&!mDone ? {1'b0, P[24:1]} + {1'b0, {24{mA[0]}}&mB}
: mStep&mDone&mRneed ? P + 1 : mShift&Sneed ? {1'b0, P[24:1]} : P;
      MulDone <= mStep&mDone&!mRneed ? 1 : mShift ? 1 : 0;
      Round <= mStep ? P[0] : Round;
      SBit <= mWait&mStart ? 0 : mStep ? (SBit | Round) : SBit;
   end
assign mDone = (mA==0)&~P[24];
assign mRneed = Round&SBit | P[0]&Round&~SBit;
assign Sneed = P[24];
endmodule
```