Part A. (40 points) In this part, you are required to implement three new instructions \((l\text{Shift}, r\text{Shift} \text{ and } or)\) for the WashU-2 processor, as described in the lab writeup. In addition, you are required to write a test program to verify that your instructions work correctly and then run a simulation of the processor executing your test program. You should write your test program in the WashU-2 assembly language (your program should execute each of the instructions at least three times) and then use the provided assembler to convert the assembly language version to machine language. On a Mac or Linux computer, this is most easily done by typing the following lines in a shell window.

```
javac Assembler.java
java Assembler myProgramFile
```

where \textit{myProgramFile} is the name of the file that contains your test program. This will also work on \texttt{oasis.cec.wustl.edu} using a command window. If you’re more comfortable running \texttt{java} programs using Eclipse, you can do that too. The output from the assembler is a list of initializers for the WashU-2’s memory. You should paste the output into the initialization section of the memory array within the \texttt{ram} component. You can use the provided \texttt{testTop1} testbench to start the processor running.

Paste a copy of the VHDL for the \textit{cpu} component of the WashU-2 below. \textbf{Highlight your changes to the code by making them bold.}

```
-- CPU module
-- Jon Turner - 5/2010
--
-- Repeatedly fetches and executes instructions from memory.
--
-- The regSelect input selects one of the four registers and
-- puts its value on the dispReg output.
--
-- if regSelect=0, dispReg = ireg
-- if regSelect=1, dispReg = pc
-- if regSelect=2, dispReg = acc
-- if regSelect=3, dispReg = iar
--
-- The pause signal is used to pause the processor for single step
-- operation. If pause is high at the end of an instruction execution,
-- the processor waits for it to go low before proceeding to the
-- next instruction fetch.
--
-- This version adds left and right shift instructions, plus a logical-or
-- instruction.
```

---
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use work.commonDefs.all;
use work.cpuArch.all;

entity cpu is port (  
  clk, reset: in  std_logic;  
  -- memory signals  
  en, rw: out std_logic;  
  aBus: out address; dBus: inout word;  
  -- console interface signals  
  pause, interrupt: in std_logic;  
  regSelect: in std_logic_vector(1 downto 0);  
  dispReg: out word);  
end cpu;

architecture cpuArch of cpu is

type state_type is (  
  resetState, pauseState, fetch,  
  halt, negate, lshift, rshift,  
  branch, brZero, brPos, brNeg, brInd,  
  cLoad, dLoad, iLoad,  
  dStore, iStore,  
  add, andd, orr  
);  

signal state: state_type;  
signal tick: std_logic_vector(3 downto 0);  
signal pc: address; -- program counter  
signal iReg: word; -- instruction register  
signal iar: address; -- indirect address register  
signal acc: word; -- accumulator  
signal alu: word; -- alu output  

-- address of the instruction being executed  
signal this: address;  
-- address for direct load, store, add, andd, ...  
signal opAdr: address;  
-- target for branch instruction  
signal target: word;

begin  
opAdr <= this(15 downto 12) & iReg(11 downto 0);  
target <= this + ((15 downto 8 => iReg(7)) & iReg( 7 downto 0));

-- connect selected register to console  
with regSelect select  
  dispReg <= iReg when "00",  
             this when "01",  
             acc when "10",  
             iar when others;

-- select alu operation based on state  
alu <= (not acc) + x"0001"  
when state = negate else
acc + dbus when state = add else
acc and dbus when state = add else
acc or dbus when state = orr else
(alu'range => '0');

-- synchronous process controlling state, tick and registers
process (clk)

function decode(instr: word) return state_type is begin
    -- Instruction decoding.
case instr(15 downto 12) is
    when x"0" =>
        case instr(11 downto 8) is
        when x"0" =>
            if instr(11 downto 0) = x"00" then
                return halt;
            elsif instr(11 downto 0) = x"01" then
                return negate;
            elsif instr(11 downto 0) = x"02" then
                return lshift;
            elsif instr(11 downto 0) = x"03" then
                return rshift;
            else
                return halt;
            end if;
        when x"1" =>
            return branch;
        when x"2" =>
            return brZero;
        when x"3" =>
            return brPos;
        when x"4" =>
            return brNeg;
        when x"5" =>
            return brInd;
        when others =>
            return halt;
        end case;
    when x"1" =>
        return cLoad;
    when x"2" =>
        return dLoad;
    when x"3" =>
        return iLoad;
    when x"5" =>
        return dStore;
    when x"6" =>
        return iStore;
    when x"8" =>
        return add;
    when x"c" =>
        return andd;
    when x"d" =>
        return orr;
    when others =>
        return halt;
    end case;
end case;
end function decode;

procedure wrapup is begin
    -- Do this at end of every instruction
    if pause = '1' then
        state <= pauseState;
    else
        state <= fetch; tick <= x"0";
    end if;
end procedure wrapup;

begin
    if rising_edge(clk) then
        if reset = '1' then
            state <= resetState; tick <= x"0";
            pc <= (others => '0');
            this <= (others => '0');
        end if;
    end if;
end
iReg <= (others => '0'); acc <= (others => '0'); iar <= (others => '0');
else
  tick <= tick + 1; -- advance time by default
  if interrupt = '1' and intEn = '1' then
    intPend <= '1';
  end if;
  if state = resetState then
    state <= fetch; tick <= x"0";
  elsif state = pauseState then
    if pause = '0' then
      state <= fetch; tick <= x"0";
    end if;
  elsif state = fetch then
    if tick = x"1" then
      iReg <= dBus; this <= pc;
    elsif tick = x"2" then
      state <= decode(iReg);
      pc <= pc + 1; tick <= x"0";
    end if;
  else
    case state is
      -- branch instructions
      when branch =>
        pc <= target; wrapup;
      when brZero =>
        if acc = x"0000" then
          pc <= target;
        end if;
        wrapup;
      when brPos =>
        if acc(15) = '0' and acc /= x"0000" then
          pc <= target;
        end if;
        wrapup;
      when brNeg =>
        if acc(15) = '1' then
          pc <= target;
        end if;
        wrapup;
      when brInd =>
        if tick = x"1" then pc <= dBus; wrapup; end if;
    end case;
    -- load instructions
    when cload =>
      acc <= (15 downto 12 => ireg(11)) & ireg(11 downto 0);
      wrapup;
    when dLoad =>
      if tick = x"1" then acc <= dBus; wrapup; end if;
    when iLoad =>
      if tick = x"1" then iar <= dBus;
      elsif tick = x"3" then acc <= dBus; wrapup;
      end if;
    -- store instructions
    when dStore => wrapup;
    when iStore =>
      if tick = x"1" then iar <= dBus;
elsif tick = x"2" then wrapup;
end if;

-- arithmetic and logic instructions
when negate => acc <= alu; wrapup;
when lShift => acc <= acc(14 downto 0) & "0"; wrapup;
when rShift => acc <= "0" & acc(15 downto 1); wrapup;
when add | add | orr =>
  if tick = x"1" then acc <= alu; wrapup; end if;
when others => state <= halt;
end case;
end if;
end if;
end process;

process (ireg,pc,iar,acc,this,opAdr,state,tick) begin

-- default values for memory control signals
en <= '0'; rw <= '1';
aBus <= (others => 'Z'); dBus <= (others => 'Z');
case state is
when fetch =>
  if tick = x"0" then
    en <= '1'; aBus <= pc;
  end if;
when brInd =>
  if tick = x"0" then
    en <= '1'; aBus <= target;
  end if;
when dLoad | add | andd | orr =>
  if tick = x"0" then
    en <= '1'; aBus <= opAdr;
  end if;
when iLoad =>
  if tick = x"0" then
    en <= '1'; aBus <= opAdr;
  elsif tick = x"2" then
    en <= '1'; aBus <= iar;
  end if;
when dStore =>
  if tick = x"0" then
    en <= '1'; rw <= '0';
    aBus <= opAdr; dBus <= acc;
  end if;
when iStore =>
  if tick = x"0" then
    en <= '1'; aBus <= opAdr;
  elsif tick = x"2" then
    en <= '1'; rw <= '0';
    aBus <= iar; dBus <= acc;
  end if;
when others =>
  state <= halt;
end case;
end process;
end cpuArch;
Paste a copy of your test program below.

-- Basic test of shift and or instructions.
--
    dLoad pat1
     lShift
     lShift
     lShift
     dLoad pat1
     rShift
     rShift
     rShift
     dLoad pat2
     or pat3
     or pat4
     or pat5
     halt

pat1:     07615
pat2:     01248
pat3:     08421
pat4:     06933
pat5:     0
Simulate the processor using the provided `testTop1` testbench. Paste a copy of your simulation output below. Your waveform window should include all the memory signals, the processor state, the tick register, plus IAR, PC and ACC. Your simulation output should demonstrate all of the instructions working correctly. You will probably need more than one screenshot to show this adequately. Make sure your signals are well-organized, with labeled dividers for the memory signals and the processor signals. Explain how the simulation demonstrates the correct operation of all the instructions.

This section shows three lShift instructions. The value x7615 is loaded into the ACC initially and then the three shifts produce the values xec2a, xd854 and xb0a8. Each value can be obtained from the previous one by shifting to the left one position, with a zero entering the low-order bit, or alternatively, by multiplying the previous value by 2. For example xec2a = x7615 << 1 = 2 * x7615.
This next section shows the right shift instructions. We start by loading x7615 in the ACC, then shifting to the right three times, giving x3b0a, x1d85, x0ec2. Each shift is equivalent to dividing by 2, so we can easily verify each result by dividing, using the standard long-division algorithm on the hex values.

This final section shows the or instructions. We first load x1248, then or the values x8421 (giving x9669), x6933 (giving xff7b) and x0000 (giving xff7b).
Part B. (40 points) In your repository, you will find a file, `divide.asm` that contains an incomplete version of a division subprogram. Complete the program and paste a copy of the completed program below (after you have gotten it working correctly). Read the provided pseudo-code carefully, and make sure you understand how the program is supposed to work, before writing any code. Be sure to include all of the comments from the original file, so that there is an obvious correspondence between the assembly language and the provided pseudo-code. **Highlight your changes by making them bold.**

```assembly
-- Division subroutine for washu-2
-- for non-negative numbers only
--
div(x,y) {
  -- if (x < 0 || y <= 0) halt;
  -- if (x == 0) return (0,y)
  -- i = 0;
  -- while (y < x && y&x4000 == 0)
  -- y = y << 1; i++;
  -- r = x; q = 0;
  -- while (true) {
  -- if (r >= y) {
  -- q = q + 1;
  -- r = r - y;
  -- }
  -- if (i == 0) break;
  -- y = y >> 1;
  -- q = q << 1;
  -- i--;
  -- }
  return(q,r);
-- }

location 0100

div_x: 0 -- first argument
div_y: 0 -- second argument
div_q: 0 -- quotient (and returned value)
div_r: 0 -- quotient (and returned value)
div_ret: 0 -- return address
div: dLoad div_x -- if (x < 0 || y <= 0) halt;
  brNeg 3
dLoad div_y
  brPos 2
  halt
dLoad div_x -- if (x == 0) return (0,y)
  brZero 2
  branch div_skip1
cLoad 0
dStore div_q
dLoad div_y
dStore div_r
  iBranch div_ret

div_skip1: cLoad 0 -- i = 0;
dStore div_i

div_loop1: dLoad div_x -- while (y < x && y&x4000 == 0) {
negate
  add div_y
  brNeg 2
  branch div_xit1
```

- 9 -
In your repo, you will also find a file called main.asm that contains an incomplete version of a test program for the division subprogram, plus a file mult.asm that contains a multiplication subprogram that is needed for the test program. Read the provided pseudo-code carefully, and make sure you understand how the program is supposed to work, before writing any code. Complete the test program and paste a copy of the completed program below. Highlight your changes by making them bold.

-- Exhaustive test program for divide subprogram.
-- Results are checked as the computation progresses.
-- If error is found, halts after loading xffff in ACC.
-- Otherwise, just continues to run
--
-- i = 10;
-- while (true) {
--  for (j = 1; j <= i+1; j++) {
--   (q,r) = div(i,j);
--   if ((i == 0 && q != 0) ||
--        (i != mult(q,j) + r)) {
--     }
--  }
--  iBranch div_ret -- return (q,r);
-- location 01f0 -- }

0
04000
acc = 0xffff; halt;
}

i++;
if (i < 0) i = 0;
}

i = 10;
while (true) {
for (j = 1; j <= i+1; j++) {
(q,r) = div(i,j)

if ((i == 0 & & q != 0) ||
(i != mult(q,j) + r)) {

acc = 0xffff; halt;
}

i++;
if (i < 0) i = 0;
branch 3
cLoad 0
dStore i
branch loop1  -- }
location 0f0

i: 0
j: 0
q: 0
r: 0

Combine the three files into a single file with the main program first, followed by the division and multiplication program. (On a Mac or Linux computer, this can be done by typing “cat main.asm divide.asm mult.asm >testDiv.asm”, for example.) Next, use the assembler to produce the machine language version and paste the output of the assembler into the ram module (replacing the code from the previous part). Now, simulate the processor running this program. Increase the length of the final wait in the testbench to at least 10 ms. For this simulation, your waveform window should include the signals i, j, q, r, div_x, div_y, div_q, div_r, mult_a, mult_b and mult_prod from the ram component. These correspond to program variables in the main program, the division subprogram and the multiply subprogram. You will find these useful when debugging. Use the unsigned radix for all these signals.

Paste the simulation output showing the entire time period when the program variable i in the main program is equal to 23. Make sure that the values of j, q and r are clearly visible throughout this period. Verify that the results computed by the division subroutine and saved in q and r are correct.

In this section, we’re dividing 23, by all values from 1 to 24. All the results are correct. For example, when j=7, the computed quotient is 3 and the remainder is 2, as expected. The program is checking its results, and since it never halts, we can be reasonably confident in the program’s correctness.
Part C. (30 points). In this part, you are to implement the interrupt instructions described in the lab writeup. You are also to extend the console, so that it implements the “continuous input” mode. Paste a copy of your VHDL for the modified cpu below. Highlight your changes by making them bold.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use work.commonDefs.all;

entity cpu is port(
    clk, reset: in  std_logic;
    -- memory signals
    en, rw: out std_logic;
    aBus: out address; dBus: inout word;
    -- console interface signals
    pause, interrupt: in std_logic;
    regSelect: in std_logic_vector(1 downto 0);
    dispReg: out word);
end cpu;

architecture cpuArch of cpu is

```

```vhdl
type state_type is (
    resetState, pauseState, fetch,
    halt, negate, lshift, rshift,
    enInt, disInt, setVec, retInt,
    branch, brZero, brPos, brNeg, brInd,
    cLoad, dLoad, iLoad,
    dStore, iStore,
    add, andd, orr
);
```

```vhdl
-- CPU module
-- Jon Turner - 5/2010
--
-- Repeatedly fetches and executes instructions from memory.
--
-- The regSelect input selects one of the four registers and
-- puts its value on the dispReg output.
--
-- if regSelect=0, dispReg = ireg
-- if regSelect=1, dispReg = pc
-- if regSelect=2, dispReg = acc
-- if regSelect=3, dispReg = iar
--
-- The pause signal is used to pause the processor for single step
-- operation. If pause is high at the end of an instruction execution,
-- the processor waits for it to go low before proceeding to the
-- next instruction fetch.
--
-- This version supports left and right shift instructions, plus a
-- logical-or instruction. It also supports four interrupt instructions
-- and an interrupt input.
--
------------------------------------------------------------------------
```
signal state: state_type;
signal tick: std_logic_vector(3 downto 0);

signal pc: address; -- program counter

signal iReg: word; -- instruction register
signal iar: address; -- indirect address register
signal acc: word; -- accumulator
signal alu: word; -- alu output

signal accShadow: word;
signal pcShadow, intVec: address;
signal intEn, intPend: std_logic;
-- address of the instruction being executed
signal this: address;
-- address for direct load, store, add, andd, ...
signal opAdr: address;
-- target for branch instruction
signal target: word;

begin
  opAdr <= this(15 downto 12) & iReg(11 downto 0);
target <= this + ((15 downto 8 => iReg(7)) & iReg(7 downto 0));

  -- connect selected register to console
  with regSelect select
    dispReg <= iReg when "00",
                this when "01",
                acc when "10",
                iar when others;

  -- select alu operation based on state
  alu <=
    (not acc) + x"0001" when state = negate else
    acc + dbus when state = add else
    acc and dbus when state = andd else
    acc or dbus when state = orr else
    (alu'range => '0');

  -- synchronous process controlling state, tick and registers
  process (clk)
  function decode(instr: word) return state_type is begin
    -- Instruction decoding.
    case instr(15 downto 12) is
    when x"0" =>
      case instr(11 downto 8) is
      when x"0" =>
        if instr(11 downto 0) = x"00" then
          return halt;
        elsif instr(11 downto 0) = x"01" then
          return negate;
        elsif instr(11 downto 0) = x"02" then
          return lshift;
        elsif instr(11 downto 0) = x"03" then
          return rshift;
        else
          return halt;
      end case;
    when others =>
      return halt;
    end case;
  end case;
function decode (instr : in OUT) return natural is
  when x"1" => return branch;
  when x"2" => return brZero;
  when x"3" => return brPos;
  when x"4" => return brNeg;
  when x"5" => return brInd;
  when x"f" =>
    if instr(7 downto 0) = x"f0" then
      return enInt;
    elsif instr(7 downto 0) = x"f1" then
      return disInt;
    elsif instr(7 downto 0) = x"f2" then
      return setVec;
    elsif instr(7 downto 0) = x"f3" then
      return retInt;
    else
      return halt;
    end if;
  when others => return halt;
end case;

end function decode;

procedure wrapup is begin
  -- Do this at end of every instruction
  if pause = '1' then
    state <= pauseState;
  else
    state <= fetch; tick <= x"0";
  end if;
end procedure wrapup;

begin
  if rising_edge(clk) then
    if reset = '1' then
      state <= resetState; tick <= x"0";
      pc <= (others => '0'); this <= (others => '0');
      iReg <= (others => '0'); acc <= (others => '0');
      iar <= (others => '0');
      pcShadow <= (others => '0'); accShadow <= (others => '0');
      intEn <= '0'; intPend <= '0';
    else
      tick <= tick + 1; -- advance time by default
      if interrupt = '1' and intEn = '1' then
        intPend <= '1';
      end if;
      if state = resetState then
        state <= fetch; tick <= x"0";
      elsif state = pauseState then

if pause = '0' then
    state <= fetch; tick <= x"0";
end if;
elsif state = fetch then
    if tick = x"0" and intPend = '1' then
        -- abort fetch and handle interrupt
        pcShadow <= pc; accShadow <= acc;
        pc <= intVec; intEn <= '0'; intPend <= '0';
        tick <= x"0";
    elsif tick = x"1" then
        iReg <= dBus; this <= pc;
    elsif tick = x"2" then
        state <= decode(iReg);
        pc <= pc + 1; tick <= x"0";
    end if;
else
    case state is
    -- branch instructions
    when branch =>
        pc <= target; wrapup;
    when brZero =>
        if acc = x"0000" then
            pc <= target;
        end if;
        wrapup;
    when brPos =>
        if acc(15) = '0' and acc /= x"0000" then
            pc <= target;
        end if;
        wrapup;
    when brNeg =>
        if acc(15) = '1' then
            pc <= target;
        end if;
        wrapup;
    when brInd =>
        if tick = x"1" then pc <= dBus; wrapup; end if;
    -- load instructions
    when cload =>
        acc <= (15 downto 12 => ireg(11)) & ireg(11 downto 0);
        wrapup;
    when dload =>
        if tick = x"1" then acc <= dBus; wrapup; end if;
    when iload =>
        if tick = x"1" then iar <= dBus;
        elsif tick = x"3" then acc <= dBus; wrapup;
        end if;
    -- store instructions
    when dstore => wrapup;
    when istore =>
        if tick = x"1" then iar <= dBus;
        elsif tick = x"2" then wrapup;
        end if;
    -- arithmetic and logic instructions
    when negate => acc <= alu; wrapup;

when lshift => acc <= acc(14 downto 0) & "0"; wrapup;
when rshift => acc <= "0" & acc(15 downto 1); wrapup;
when add | andd | orr =>
  if tick = x'1" then acc <= alu; wrapup; end if;

-- interrupt handling
when enInt => intEn <= '1'; intPend <= '0'; wrapup;
when disInt => intEn <= '0'; wrapup;
when setVec => intVec <= acc; wrapup;
when retInt => acc <= accShadow; pc <= pcShadow;
  intEn <= '1'; intPend <= '0'; wrapup;
when others => state <= halt;
end case;
end if;
end if;
end if;
end if;
end process;

process (ireg,pc,iar,acc,this,opAdr,state,tick) begin

-- Memory control section (combinational)
-- default values for memory control signals
en <= '0'; rw <= '1';
aBus <= (others => 'Z'); dBus <= (others => 'Z');
case state is
  when fetch =>
    if tick = x"0" then
      en <= '1'; aBus <= pc;
    end if;
  when brInd =>
    if tick = x"0" then
      en <= '1'; aBus <= target;
    end if;
  when dLoad | add | andd | orr =>
    if tick = x"0" then
      en <= '1'; aBus <= opAdr;
    end if;
  when iLoad =>
    if tick = x"0" then
      en <= '1'; aBus <= opAdr;
    elsif tick = x"2" then
      en <= '1'; aBus <= iar;
    end if;
  when dStore =>
    if tick = x"0" then
      en <= '1'; rw <= '0';
      aBus <= opAdr; dBus <= acc;
    end if;
  when iStore =>
    if tick = x"0" then
      en <= '1'; aBus <= opAdr;
    elsif tick = x"2" then
      en <= '1'; rw <= '0';
      aBus <= iar; dBus <= acc;
    end if;
  when others =>
    end case;
end process;
end cpuArch;

Paste a copy of your VHDL for the modified console below. **Highlight your changes by making them bold.**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
use work.commonDefs.all;

entity console is port(
  clk: in std_logic;

  -- inputs and outputs
  btn: in buttons;
  knob: in knobSigs;
  swt: in switches;

  resetOut: out std_logic;
  pause, interrupt: out std_logic; -- pause/interrupt CPU

  -- memory signals
  memEnIn, memRwIn: in std_logic;
  memEnOut, memRwOut: out std_logic;
  aBus: out word;
  dBus: inout word;
```
-- signals for observing on CPU registers
regSelect: out std_logic_vector(1 downto 0);
cpuReg: in word;

-- signals for controlling LCD display
lcd: out lcdSigs);
end console;

architecture a1 of console is

component debouncer
generic(width: integer := 8);
port( clk: in std_logic;
din: in std_logic_vector(width-1 downto 0);
dout: out std_logic_vector(width-1 downto 0));
end component;

component knobIntf port(
clk, reset: in std_logic;
knob: in knobSigs; -- knob signals
tick: out std_logic; -- high for each knob transition
clockwise: out std_logic; -- high for clockwise rotation
delta: out word; -- add/subtract amount
end component;

component lcdDisplay port(
clk, reset : in std_logic;
-- internal interface for controlling display
update: in std_logic; -- update a stored value
selekt: in std_logic_vector(4 downto 0); -- character to replace
nuChar: in std_logic_vector(7 downto 0); -- new character value
-- connections to external pins
lcd: out lcdSigs);
end component;

signal dBtn, prevDBtn: buttons;
signal reset: std_logic;
signal tick, clockwise : std_logic;
signal delta: word;

-- single step control signal
signal singleStep: std_logic;

-- local signals for controlling memory
signal memEn, memRw: std_logic;

-- signals for controlling snooping
signal snoopAdr, snoopData: word;
signal snoopCnt: std_logic_vector(6*operationMode + 9 downto 0);
signal snoopTime, writeReq: std_logic;

-- 0 means out, 2 means in, 3 means continuous in
signal snoopMode: std_logic_vector(1 downto 0);

-- signals for controlling lcdDisplay
constant CNTR_LENGTH: integer := 8 + operationMode*12;
signal lcdCounter: std_logic_vector(CNTR_LENGTH-1 downto 0);
begin
  snoopMode <= swt(3) & swt(2);
  reset <= dBtn(0); resetOut <= reset;

  -- connect all the sub-components
  db: debouncer generic map(width => 4) port map(clk, btn, dBtn);
  kint: knobIntf port map(clk, reset, knob, tick, clockwise, delta);
  disp: lcdDisplay port map(clk, reset, update, selekt, nuchar, lcd);

  pause <= singleStep or snoopTime;

  -- process for controlling single step operation
  process(clk) begin
    if rising_edge(clk) then
      prevDBtn <= dBtn;
      if reset = '1' then
        singleStep <= '0';
      else
        if dBtn(3) > prevDBtn(3) then
          singleStep <= not singleStep;
        elsif dBtn(3) = '1' then
          singleStep <= '1';
        elsif dBtn(2) > prevDBtn(2) then
          singleStep <= '0';
        end if;
      end if;
    end if;
  end process;

  memEnOut <= memEnIn or memEn;
  memRwOut <= memRwIn and memRw;

  snoopTime <= '1' when snoopCnt(snoopCnt'high downto 4) =
    (snoopCnt'high downto 4 => '1')
    else '0';

  -- process controlling memory signals for snooping
  process (snoopTime, snoopCnt, snoopData, snoopAdr) begin
    memEn <= '0'; memRw <= '1';
    aBus <= (others => 'Z'); dBus <= (others => 'Z');
    interrupt <= '0';
    if snoopTime = '1' then
      -- allow time for in-progress instruction to complete
      if snoopCnt(3 downto 0) = x"c" then
        memEn <= '1'; aBus <= snoopAdr;
      elsif writeReq = '1' and snoopCnt(3 downto 0) = x"f" then
        memEn <= '1'; memRw <= '0';
    end if;
  end process;
aBus <= snoopAdr; dBus <= snoopData;
interrupt <= '1';
end if;
end if;
end process;

-- process that controls snoop registers and writeReq
process(clk) begin
if rising_edge(clk) then
if reset = '1' then
snoopAdr <= (others => '0'); snoopData <= (others => '0');
writeReq <= '0';
snoopCnt <= (others => '0');
else
snoopCnt <= snoopCnt + 1;
end if;
-- generate writeReq signal in response to button push
if dBtn(1) > prevDBtn(1) and snoopMode = "10" then
writeReq <= '1';
end if;
if writeReq = '1' and snoopTime = '1'
and snoopCnt(3 downto 0) = x"f" then
writeReq <= '0';
end if;
-- load snoopData at end of snoopTime period
if snoopTime = '1' and snoopMode = "00" then
if snoopCnt(3 downto 0) = x"d" then
snoopData <= dBus;
end if;
end if;
-- update snoopAdr or snoopData in response to knob signals
if tick = '1' then
if snoopMode = "00" then
if clockwise = '1' then snoopAdr <= snoopAdr + delta;
else snoopAdr <= snoopAdr - delta;
end if;
elself
if clockwise = '1' then snoopData <= snoopData + delta;
else snoopData <= snoopData - delta;
end if;
if snoopMode = "11" then writeReq <= '1'; end if;
end if;
end if;
end if;
end process;

-- update LCD display to show cpu registers and snoop registers
-- first row: ireg acc snoopAdr
-- second row: pc iar snoopData
lowBits <= lcdCounter(CNTR_LENGTH-6 downto 0);
update <= '1' when lowBits = (lowBits'range => '0') else '0';
selekt <= lcdCounter(CNTR_LENGTH-1 downto CNTR_LENGTH-5);
regSelect <=
"00" when selekt <= slv(4,5) else
"10" when selekt <= slv(10,5) else
"01" when selekt <= slv(20,5) else
"11";
process (cpuReg, snoopAdr, snoopData, selekt) begin
  case selekt is
    when "00000" | "00110" | "10000" | "10110" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(15 downto 12))));
    when "00001" | "00111" | "10001" | "10111" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(11 downto 8))));
    when "00010" | "01000" | "10010" | "11000" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(7 downto 4))));
    when "00011" | "01001" | "10011" | "11001" =>
      nuChar <= c2b(hex2Ascii(int(cpuReg(3 downto 0))));
    when "01100" => nuChar <= c2b(hex2Ascii(int(snoopAdr(15 downto 12))));
    when "01101" => nuChar <= c2b(hex2Ascii(int(snoopAdr(11 downto 8))));
    when "01110" => nuChar <= c2b(hex2Ascii(int(snoopAdr(7 downto 4))));
    when "01111" => nuChar <= c2b(hex2Ascii(int(snoopAdr(3 downto 0))));
    when others => nuChar <= c2b(' ');
  end case;
end process;

process(clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      lcdCounter <= (others => '0');
    else
      lcdCounter <= lcdCounter + 1;
    end if;
  end if;
end process;
end a1;
Part D. (50 points) In this part, you will write an Etch-a-Sketch program that runs whenever new data is written to memory, causing an interrupt to be triggered. Before your program can run, you need to setup the interrupt subsystem so that the interrupt vector points to the start of your interrupt program, and interrupts are enabled. Write a short code fragment that does this and insert it at the start of the main program you wrote earlier to test the division program. Paste a copy of this code fragment below.

-- Setup Interrupts
--
-- set interrupt vector = x0300
-- enable interrupts
  cLoad 0300
  setVec
  enInt

You will find a file with an incomplete version of the Etch-a-Sketch program in your repository. Complete this program. Note that the program needs to use the division subroutine we wrote earlier, but since the main program is also going to be performing divisions, we will need a separate copy of the division program that can be used here. Call it div2 and adjust all the labels in this copy so that they start with div2 instead of div. Paste a copy of the Etch-a-Sketch program below. **Highlight your changes by making them bold.**

-- Etch-a-sketch application
--
-- Entered via an interrupt that is triggered when a new data
-- value has been written to memory.
--
-- Examine M[03f0] which specifies a pixel in the display buffer.
-- The top byte specifies the x-coordinate of the pixel,
-- the bottom byte specifies the y-coordinate.
-- Make the specified pixel white.
--
-- y = pixel & 0ff;  -- pixel = M[03f0]
-- x = pixel >> 8;
-- q = x/5; r = x mod 5;
-- p = 0f000 + 32*y + q;
-- if (r == 0) *p = *p | x7000;
-- if (r == 1) *p = *p | x0e00;
-- if (r == 2) *p = *p | x01c0;
-- if (r == 3) *p = *p | x0038;
-- if (r == 4) *p = *p | x0007;
--
-- Then return from interrupt
  location 0300
etch:       cLoad 0ff
      and etch_pix
dStore etch_y
dLoad etch_pix
  rShift
  rShift
  rShift
  rShift
  rShift
  rShift
  rShift
  rShift
dStore etch_x

dLoad etch_x -- q = x/5; r = x mod 5;

dStore div2_x
cLoad 5
dStore div2_y
cLoad etch_ret
dStore div2_ret

iBranch 1
div2

etch_ret:

dLoad div2_q
dStore etch_q
dLoad div2_r
dStore etch_r
dLoad etch_f000

-- p = 0f000 + 32*y + q;
dStore etch_y

1Shift
1Shift
1Shift
1Shift
add etch_p
add etch_q
dStore etch_p
dLoad etch_r

-- if (r == 0) *p = *p | x7000;
brZero 2
branch 4
iLoad etch_p
or etch_m0
iStore etch_p
cLoad 1

-- if (r == 1) *p = *p | x0e00;
negate
add etch_r
brZero 2
branch 4
iLoad etch_p
or etch_m1
iStore etch_p
cLoad 2

-- if (r == 2) *p = *p | x01c0;
negate
add etch_r
brZero 2
branch 4
iLoad etch_p
or etch_m2
iStore etch_p
cLoad 3

-- if (r == 3) *p = *p | x0038;
negate
add etch_r
brZero 2
branch 4
iLoad etch_p
or etch_m3
iStore etch_p
cLoad 4

-- if (r == 4) *p = *p | x0007;
negate
add etch_r
brZero 2
branch 4
iLoad etch_p
or etch_m4
iStore etch_p
retInt

-- return from interrupt

location 03f0

etch_pix: 0 -- location changed by user
etch_x: 0 -- x-coordinate
etch_y: 0 -- y coordinate
etch_p: 0 -- pointer to x,y pixel
etch_q: 0 -- q = x/5
etch_r: 0 -- q = x mod 5
etch_m0: 07000 -- mask for first pixel in word
etch_m1: 00e00 -- mask for second pixel in word
etch_m2: 001c0 -- mask for third pixel in word
etch_m3: 00038 -- mask for fourth pixel in word
etch_m4: 00007 -- mask for fifth pixel in word
etch_f000: 0f000 -- first address in display buffer
You will need to combine your *Etch-a-Sketch* program with the other components. Combine the modified version of `main`, the `division` subprogram, the `multiply` subprogram, the *Etch-Sketch* subprogram and the `div2` subprogram, into a single file, in that order. Then, use the assembler to generate the machine language version and paste the output into the `ram`. Simulate your program with the provided testTop2 testbench. This includes input signals from the knob that should cause your *Etch-a-Sketch* program to draw a square on the screen. Paste a copy of the waveform window covering the time period from 3.4 ms to 4 ms. Your waveform window should include the signals `i`, `j`, `p` and `q` from the `ram` module, as before, and the signals `etch_pix`, `etch_x`, `etch_y`, `etch_p`, `etch_q`, `etch_r` from the `ram` module. It should also include the interrupt signal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[i[15:0]]</td>
<td>17</td>
</tr>
<tr>
<td>[j[15:0]]</td>
<td>5</td>
</tr>
<tr>
<td>[q[15:0]]</td>
<td>3</td>
</tr>
<tr>
<td>[r[15:0]]</td>
<td>2</td>
</tr>
<tr>
<td>etch_a-sketch</td>
<td></td>
</tr>
<tr>
<td>etch_pix[15:0]</td>
<td>0</td>
</tr>
<tr>
<td>etch_x[15:0]</td>
<td>82</td>
</tr>
<tr>
<td>etch_y[15:0]</td>
<td>70</td>
</tr>
<tr>
<td>etch_p[15:0]</td>
<td>63503</td>
</tr>
<tr>
<td>etch_q[15:0]</td>
<td>16</td>
</tr>
<tr>
<td>etch_r[15:0]</td>
<td>2</td>
</tr>
</tbody>
</table>

Verify that the several of the values computed by your program for `etch_x` and `etch_y` are correct, given the `etch_pix` values. Explain how the values match.

At the left end of the window, we see that `x=76` and `y=64`. The corresponding value of `etch_pix` is `76*256+64=19520`, which does match the value shown in the simulation. As `x` increases from here, we expect `etch_pix` to increase by `256`, and that is what we observe. About halfway through the waveform, `x` stops changing and `y` increments. Here, we also expect to see `etch_pix` incrementing, and we do. At `x=82`, `y=64`, `etch_pix=82*256+64=21056`, which matches the value in the waveform window. From here, each new value is one larger than the previous one.

Verify that several of the values of `etch_p` are correct, given the values of `etch_x` and `etch_y`. Explain how the values match.

When `x=75` and `y=64`, `etch_p` should be `15*4096 + 64*160/5 + 75/5 = 63503`, which matches the value shown. As `x` increases up to `79`, this value remains the same. When `x` becomes `80`, it increases by `1`. When `y` starts changing, we expect to see `etch_p` increasing by `32`, and this is what we observe.

Verify that the test program for the division subprogram continues to work correctly, even when it is being interrupted periodically by the *Etch-a-Sketch* program.

The test program is dividing `16` and `17` by several values in this part of the simulation. We observe that the computed value `17/3` is `5` with a remainder of `2`, as we expect. The other division operations also appear to be producing the correct results.
Next, find a place in the simulation where the interrupt signal goes high and zoom in on this part of the simulation. Here, your waveform window should show the memory signals, the processor registers and all the interrupt signals including the two shadow signals.

Verify that all signal changes that happen here are correct. Explain.

When interrupt does high, we see that the processor aborts the next fetch and the PC changes from 0115 to 0300, which is the start of the interrupt subprogram. The interrupt enable and pending signals also go low at this point and the accShadow becomes 0018 (which is the value in the acc) while the pcShadow becomes 0115.
From this point in the simulation scan ahead to the point where the interrupt subprogram returns (you should see $\text{intEn}$ go high at this point). Paste a copy of the waveform below.

Verify that the processor is correctly handling the interrupt return. Explain.

The interrupt return occurs just past the middle of the waveform window. We can see the instruction code 0ff3 for the return-from-interrupt instruction. When this instruction executes, the PC value is changed to 0115 and the ACC to 0018. These are the values that were saved earlier.
Part E. (10 points) Proceed to this part only after you have completed the simulation in Part F and have convinced yourself that the complete circuit will work correctly when transferred to the prototype board. Prototype your circuit using one of the prototype boards available in Bryan 316. Once you have your circuit loaded onto the board and you have convinced yourself that it works correctly, fill in your name below on the printed copy and have one of the TAs check it and sign their name below, after assigning the appropriate number of demo points.

Student name: _________________________________ has successfully demonstrated the Etch-a-Sketch program running on the modified WashU-2.

TA name: ______________________________________

TA signature: _________________________________

Demo points (out of 10):_________________

Comments (if the program does not work 100% correctly, make a note of all issues below):