Design Studies – Part 1

- 4-way Max Finder
- Binary Input Module
- LCD Display Controller
- Binary Output Module

Jon Turner
4-Way Max Finder Design

- Design a circuit with four serial inputs and a serial output equal to the largest input value (values come in msb first); include reset and make output synchronous.

- Break down into three 2-way maximum circuits:
  - 2-way max circuit compares its two input values and propagates the larger value.
entity max4 is port (  clk, reset : in std_logic;  a, b, c, d : in std_logic;  biggest : out std_logic);
end max4;

architecture arch of max4 is

type stateType is (eq, lt, gt); -- states for 2-way comparator

function nextState(state:stateType; x,y:std_logic) return stateType is
  begin
    if state = eq and x > y then return gt;
    elsif state = eq and x < y then return lt;
    else return state;
  end if;
end function nextState;

function maxBit(state: stateType; x,y: std_logic) return std_logic is
  begin
    if state = gt or (state = eq and x > y) then return x;
    else return y;
  end if;
end function maxBit;
signal s_ab, s_cd, s: stateType;
signal m_ab, m_cd, m: std_logic;
beg
m_ab <= maxBit(s_ab,a,b);
m_cd <= maxBit(s_cd,c,d);
m <= maxBit(s,m_ab,m_cd);
process(clk) begin
  if rising_edge(clk) then
    assert m >= a or s_ab = lt or s = lt;
    assert m >= b or s_ab = gt or s = gt;
    assert m >= c or s_cd = lt or s = gt;
    assert m >= d or s_cd = gt or s = gt;
    if reset = '1' then
      s_ab <= eq; s_cd <= eq; s <= eq;
      biggest <= '0';
    else
      s_ab <= nextState(s_ab,a,b);
      s_cd <= nextState(s_cd,c,d);
      s <= nextState(s,m_ab,m_cd);
      biggest <= m;
    end if;
  end if;
end process;
end arch;
Exercises

1. Draw a state diagram for a 3-way max finder that is not based on a 2-way max finder. How many states are needed? How many would be needed for a 4-way max finder?
2. Write an entity-architecture for a 2-way max finder (Mealy mode). Write an entity-architecture for a 4-way max finder (Moore mode) using the 2-way component as a building block.
3. There is a simple asynchronous state machine for adding two integer values together serially, when they are presented with the least-significant bit first. Draw a state diagram for such a state machine (hint: the state machine must remember the carry that resulted from adding the previous two bits). Explain how you can use 3 copies of this state machine to add four integers.
4. Rewrite the nextState function from the max finder to implement the state transitions for your serial adder.
5. Write a sumBit function, patterned after the maxBit function from the max finder, that produces the output value for your serial adder.
Binary Input Module

- Used with prototype board to facilitate use of knob and buttons
  - buttons debounced and “pulsified”
  - uses knob (and knob interface circuit) used to control a 16 bit register

- To create pulses from debounced buttons
  - “remember” previous button value and make pulse output high if current value is high and previous value was low

- To control register
  - use tick output from knob interface to add to or subtract from register (based on value of clockwise) output
  - knob interface’s delta output gives amount to add/sub
VHDL for BinaryInMod

entity binaryInMod is port(
    clk: in std_logic;
    -- inputs from actual buttons and knob
    btn: in buttons; knob: in knobSigs;
    -- outputs processed with inputs
    resetOut: out std_logic; -- debounced btn(0)
    dBtn: out std_logic_vector(3 downto 1); -- debounced btn(3..1)
    pulse: out std_logic_vector(3 downto 1); -- pulse versions, btn(3..1)
    inBits: out word; -- value controlled by knob
end binaryInMod;

architecture al of binaryInMod is
component debouncer ... end component;
component knobIntf ... end component;

signal dbb, dbb_prev: buttons; -- debounced buttons, previous values
signal reset: std_logic; -- reset from btn(0)
signal tick, clockwise: std_logic; -- from knob interface
signal delta: word; -- from knob interface
signal bits: word; -- internal register

begin
  -- debounce the buttons and knob
  dbl: debouncer generic map(width => 4) port map(clk, btn, dbb);
  dBn <= dbb(3 downto 1);
  reset <= dbb(0); resetOut <= reset;
  ki: knobIntf port map(clk, reset, knob, tick, clockwise, delta);

  -- define pulse and inBits
  process (clk) begin
    if rising_edge(clk) then
      dbb-prev <= dbb; -- previous debounced buttons
      if reset = '1' then bits <= (others => '0');
      elsif tick = '1' then
        if clockwise = '1' then bits <= bits + delta;
        else bits <= bits - delta;
      end if;
    end if;
  end process;
  pulse <= dbb(3 downto 1) and (not dbb-prev(3 downto 1));
  inBits <= bits;
end ai;
Simulation of BinaryInMod

debounced button and pulse signals
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>btn[3:0]</td>
<td>0</td>
</tr>
<tr>
<td>db[3:0]</td>
<td>0</td>
</tr>
<tr>
<td>dbb_prev[3:0]</td>
<td>0</td>
</tr>
<tr>
<td>resetout</td>
<td>0</td>
</tr>
<tr>
<td>dbtn[3:1]</td>
<td>0</td>
</tr>
<tr>
<td>pulse[3:1]</td>
<td>0</td>
</tr>
<tr>
<td>clk</td>
<td>0</td>
</tr>
<tr>
<td>knob (15:0)</td>
<td></td>
</tr>
<tr>
<td>inBits[15:0]</td>
<td>0x02F</td>
</tr>
</tbody>
</table>

The inBits derived from knob interface signals.
Exercises

1. Draw a block diagram for the binary input module circuit. You can show the sub-components as labeled blocks.

2. The prototype boards have only four buttons, which can be a little constraining in some situations. One way to provide some extra control signals is to generate two different pulse outputs per button. One signal is generated if the button is pressed, then immediately released, while the other is generated if the button is pressed and held for at least half a second, before it is released. Write a version of the binary input module that has 6 pulse outputs. The first three are activated if the corresponding button is released after a short delay. The other three are activated if the corresponding button is released after a long delay.
LCD Display Interface

- **Client-side interface to write characters to buffer**
  - 32 byte buffer corresponding to 32 characters on display
  - `selekt` – specifies byte in buffer
  - `nuChar` – ASCII code of character to be written
  - `update` – asserted to write one character

- **Controller-side interface**
  - `data` (4 bits) – used to send data and control bytes to display, four bits at a time
  - `control signals` used to implement protocol with display device
    - details can be found in manual for prototype board
VHDL Source

entity lcdDisplay is port(
  clk, reset : in std_logic;
  update: in std_logic;
  selekt: in std_logic_vector(4 downto 0);
  nuChar: in byte;
  lcd: out lcdSig);
end lcdDisplay;

architecture a1 of lcdDisplay is
  type char_buf is array(0 to 31) of byte;
  signal cb: char_buf := (others => x"20"); -- initialize with spaces
begin
  process(clk) begin -- update character buffer when update is asserted
    if rising_edge(clk) then
      if reset = '0' and update = '1' then
        cb(int(selekt)) <= nuChar;
      end if;
    end if;
  end process;
end a1;
Simulation – client-side interface

writing values to character buffer
values written to buffer
Binary Output Module

- Displays binary values on proto-board LCD display
  - input is simply two 16 bit values
  - shown as 0s and 1s on two rows of display
- Implemented using *lcdDisplay* interface circuit
  - write input bits to *LcdDisplay*’s internal buffer
  - update *LcdDisplay* periodically
    - often enough to appear instantaneous to human user
    - not so often as to cause display flicker
  - use counter to control timing of update operations on *LcdDisplay*
Design of Output Module

- Counter used to control when bytes are written to the `lcdDisplay`'s internal buffer.
- BitSelect block selects the input bit to display:
  - `topRow(15)` when `selekt=0`, `topRow(14)` when `selekt=1`,...
entity binaryOutMod is
  clk, reset: in std_logic;
  topRow, botRow: in word; -- binary values to display
  lcd: out lcdSigs); -- signals for controlling LCD display
end entity binaryOutMod;

architecture al of binaryOutMod is

component lcdDisplay ... end component;

-- counter for controlling when to updatelcdDisplay
constant CNTR_LENGTH: integer := 6+14*operationMode;
signal counter: std_logic_vector(CNTR_LENGTH-1 downto 0);
signal lowBits: std_logic_vector(CNTR_LENGTH-6 downto 0);

-- signals for controlling lcdDisplay
signal update: std_logic;
signal selekt: std_logic_vector(4 downto 0);
signal nuChar: std_logic_vector(7 downto 0);
begin
  disp: lcdDisplay port map(clk, reset, update, selekt, nuchar, lcd);

  lowBits <= counter(CNTR_LENGTH-6 downto 0);
  update <= '1' when lowBits = (lowBits'range => '0') else '0';
  selekt <= counter(CNTR_LENGTH-1 downto CNTR_LENGTH-5);

  nuchar <= x"30" when (selekt(4) = '0' and
    topRow((wordSize-1)-int(selekt(3 downto 0))) = '0') or
    (selekt(4) = '1' and
    botRow((wordSize-1)-int(selekt(3 downto 0))) = '0') else
    x"31";

  process(clk) begin
    if rising_edge(clk) then
      counter <= counter + 1;
      if reset = '1' then
        counter <= (others => '0');
      end if;
    end if;
  end process;
end;
Exercises

1. Modify the binary output module to add a *blink* input. When this input is high, the display should blink on and off, with a period of about two seconds (on for one second, off for one second). When the *blink* input is low, the input data should be displayed in the normal way. Assume a 50 MHz clock.

2. Write a VHDL module that uses the LcdDisplay module to display a 16 bit input value on the display in hexadecimal rather than binary. Write the value in the first four positions of the first row. Your circuit should update the displayed value periodically (say once every 20 milliseconds).