Sequential Circuits – Part 2

- State machines with registers
- Debouncing input signals
- Knob interface for S3 board
- Garage door opener revisited

Jon Turner
State Machines with Registers

- State machines often used as controllers for more complex circuits containing registers
  - controls register values
  - state changes may depend on register values
- Useful to specify using high level state diagrams
  - general conditions on transitions
  - actions performed on registers
- Example – pulse counter
  - input $N$ loaded on first tick
  - event output goes high after $N$ pulses
High Level State Diagram

- End of pulse detected by down-transition on dIn
  - but if dIn starts high, ignore first down-transition

- States
  - reset – enter on reset high and stay until drops
  - start1 – enter if dIn=1 after reset
  - prev0, prev1 – track previous dIn value
entity pulseCount is ...
architecture arch of pulseCount is

signal state: stateType;
signal cnt: std_logic_vector(3 downto 0);

begin
process (clk) begin
  if rising_edge(clk) then
    if reset = '1' then
      state <= resetState; cnt <= x"F";
    else
      case state is
        when resetState =>
          if dfIn = '0' then state <= prev0;
          else state <= start1;
        end if;

        when start1 =>
          if dfIn = '0' then state <= prev0; end if;

        when prev0 =>
          if dfIn = '1' then state <= prev1; end if;

        when prev1 =>
          if dfIn = '0' then
            if cnt /= 0 then cnt <= cnt - 1; end if;
          end if;

        when others =>
          end case;
      end if;
  end if;
end process;

event <= '1' when cnt = x"0" else '0';
end arch;
Simulation Results
Exercises

1. Write a VHDL specification for a separate counter component corresponding to the `cnt` block in the block diagram of the pulse counter. Write a new VHDL specification of a state machine controller that instantiates this component and uses it to implement the pulse-counter.

2. The pulse counter output goes high and stays high when the $N$-th pulse is detected. After this happens, new pulses are not counted. Modify the state diagram so that when the circuit detects the $N$-th pulse, it raises the event output for one clock tock and reloads the counter, then resumes counting pulses, raising the event output again after $N$ more pulses.

3. Show how to modify the VHDL for the pulse counter to implement the version described in the last exercise.

4. Draw a state diagram for a circuit that counts pairs of pulses, rather than individual pulses, where a pulse-pair consists of two pulses that are separated by one clock tick.

5. Show how to modify the VHDL for the pulse counter to implement a pulse pair counter.
## Designing State Machine Controllers

- **Determine what the circuit must “remember”**
  - data stored in registers
  - control state defines steps in process implemented by the circuit

- **Define high level state diagram**
  - state transitions defined among different control states
  - conditions defined on data in registers may determine control state transitions
    - e.g. if count = limit and increment = 1 goto overflow_state
  - state transitions may trigger actions affecting stored data
    - e.g. clear count and set overflow error bit

- **Use state diagram to write VHDL**
  - use state and register values to control transitions
  - make changes to register values on transitions

- **In some cases, no need for explicit states**
  - state is implied by data values
Debouncer

- Mechanical buttons can vibrate as they are pressed and released
  - single button press may appear like several
  - job of debouncer is to filter out “false presses”
  - propagate changes only if they are stable for $\Delta$ ticks
  - typical duration for $\Delta$ is 1 ms
VHDL for Debouncer

entity debouncer is
  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 debouncer;

architecture a1 of debouncer is
signal prevDin: std_logic_vector(width-1 downto 0);
constant debounceBits: integer := 2 + operationMode*14;
signal count: std_logic_vector(debounceBits-1 downto 0);
begin
  process(clk) begin
    if rising_edge(clk) then
      prevDin <= din;
      if prevDin /= din then count <= (others => '1');
      else if count /= (count'range => '0') then count <= count - 1;
      end if;
    else dout <= din;
  end if;
  end process;
end a1;
Simulation Results

- Ignore changes that are not stable.
- Respond to changes after 4 ticks.
Exercises

1. Draw a circuit diagram for the debouncer, showing all registers and the logic involved in updating them as the input changes.

2. Suppose that you are implementing a circuit that is part of a system that uses a 15 MHz clock and you need a debouncer that filters out input signal transitions that are stable for less than 5 milliseconds. How would you modify the debouncer VHDL for use in this system?
Knob Interface Circuit

- Ridges on knob close pair of switches as knob turns
  - direction of rotation determines which switch closes first
  - outputs from interface circuit
    - \textit{tick} high for one clock tick for each "button cycle"
    - clockwise – high when knob is turning clockwise, else low
    - these make it easy to increment or decrement a register as knob turns
    - delta – pressing down on knob changes value from 1 to 16 to 256 to 4096 and back to 1

- Knob signals must be debounced first
Knob Interface VHDL

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

architecture al of knobIntf is

  signal dbKnob: knobSigs;
  signal rot, prevRot: std_logic vector(1 downto 0);
  signal btn, prevBtn: std_logic;
  signal diff : word;

current and previous rotational signals
debounced knob signals
internal register for delta output
current and previous knob press signals
begin
  db: debouncer generic map(width => 3) port map(clk, knob, dbKnob);
  rot <= dbKnob(2 downto 1); btn <= dbKnob(0); delta <= diff;
  process(clk) begin
    if rising_edge(clk) then
      prevRot <= rot; prevBtn <= btn; tick <= '0';
      if reset = '1' then
        diff <= (0 => '1', others => '0');
        clockwise <= '1';
      else
        if prevRot = "00" and rot = "01" then
          tick <= '1'; clockwise <= '0';
        end if;
        if prevRot = "10" and rot = "11" then
          tick <= '1'; clockwise <= '1';
        end if;
        if btn > prevBtn then
          if diff(wordSize-1 downto wordSize-4) /= "0000" then
            diff <= (0 => '1', others => '0');
          else
            diff <= diff(wordSize-5 downto 0) & "0000";
          end if;
        end if;
      end if;
    end if;
  end process;
end al;
Simulation Results

knob turning counter-clockwise

tick per rotation

button presses change delta

changing direction
Exercises

1. Write a VHDL module that uses the knob interface module to produce a 16 bit output signal called knobValue. KnobValue should be incremented or decremented on every “tick” reported by the knob interface (depending on the value of clockwise).
Two Speed Garage Door Opener

- From stop, start slow, speed up after 2 seconds
- When door almost closed, slow down
- Things to remember
  - how long door has been in motion
  - position of the door
    - track position based on door speed and time passed
- Complications
  - variety of different door heights, variation in motor speed
    - adapt to height of specific door, by setting height to zero when door reaches bottom
  - if door starts in middle, position unknown
    - go slow as door closes
Partial State Machine

- **opened**: (inputs: openClose, obstruction, atTop, atBot)
  - inputs=10xx/
    - goDown <= 1; speed <= 0;
    - timeInMotion <= 0;
    - timeSinceUpdate <= 0

- **closing**: (inputs=00x0 and close to bottom/
  - speed <= 0)
  - inputs=00x1/
    - goDown <= 0

- **closed**: (inputs=00x0 and not close to bottom and
time to speedup/
  - speed <= 1)

- **opening**: (inputs=x1xx/
  - goUp <= 1; speed <= 0;
  - timeInMotion <= 0;
  - timeSinceUpdate <= 0)

- **pauseDown**: (inputs=10xx/
  - goDown <= 0)
entity opener is port (  
    clk, reset: in std_logic;
  ...  
    doorSpeed: out std_logic); -- 0 for slow, 1 for fast
end opener;

architecture al of opener is
  type stateType is (opened, closed, opening, closing,  
    pauseUp, pauseDown, resetState);
signal state: stateType;

subtype timeVal is unsigned(15 downto 0); -- up to 65 seconds
subtype positionType is signed(9 downto 0); -- up to 511 cm

constant slowPeriod: timeVal := to_unsigned(100,16); -- ms per cm
constant fastPeriod: timeVal := to_unsigned(40,16); -- ms per cm
constant ticksPerMs: unsigned(15 downto 0) := to_unsigned(50000,16);
custom speedUpTime: timeVal := to_unsigned(2000,16);
custom slowDownPos: positionType := to_signed(25,10);

signal speed: std_logic; -- 0 for slow, 1 for fast
signal ticks: unsigned(15 downto 0); -- count clock ticks to keep time
signal timeInMotion: timeVal; -- time door has been moving
signal timeSinceUpdate: timeVal; -- time since position updated
signal position: positionType; -- position of door (cm from bot)
begin  
  process (clk) begin  
    if rising_edge(clk) then  
      if reset = '1' then  
        state <= resetState; speed <= '0';  
        ticks <= (others => '0');  
        timeInMotion <= (others => '0');  
        timeSinceUpdate <= (others => '0');  
        position <= (others => '0');  
      elsif state = resetState then  
        if atTop = '1' then state <= opened;  
        elsif atBot = '1' then state <= closed;  
        else state <= pauseDown;  
        end if;  
      else  
        -- update time signals  
        if ticks = ticksPerMs then  
          timeInMotion <= timeInMotion + 1;  
          timeSinceUpdate <= timeSinceUpdate + 1;  
          ticks <= (others => '0');  
        else  
          ticks <= ticks + 1;  
        end if;  
      end if;  
  end process;
-- state transitions

case state is
when opened =>
  speed <= '0';
  if openClose = '1' and obstruction = '0' then
    state <= closing;
    timeInMotion <= (others => '0');
    timeSinceUpdate <= (others => '0');
  end if;
when closing =>
  if openClose = '0' and obstruction = '0' and stBot = '1' then
    state <= closed; position <= (others => '0');
  elsif obstruction = '1' then
    state <= opening; speed <= '0';
    timeInMotion <= (others => '0');
    timeSinceUpdate <= (others => '0');
  elsif openClose = '1' then state <= pauseDown;
  elsif position <= slowDownPos then speed <= '0';
  elsif timeInMotion = speedUpTime then speed <= '1';
  end if;
-- track position of door
if (speed = '0' and timeSinceUpdate = slowPeriod) or
   (speed = '1' and timeSinceUpdate = fastPeriod) then
  position<=position-1; timeSinceUpdate<=others=>'0';
end if;
Simulation

- Initial closing stays slow
- Opening door speeds up
- Slows down near bottom
pausing and resuming and again reversing for obstruction