Sequential Circuit Design

- Registers and Counters
- Complex Sequential Circuits
Registers in the Basic Computer

- Registers are basic building blocks in digital systems.
  - store information
  - auxiliary circuits may modify stored information or “steer it” to and from register

Diagram showing:
- Program Counter
- Instruction Register & Decoder
- Indirect Address Register
- Accumulator
- Controller
- Memory
- Data Bus
- Address Bus

Components labeled:
- PC
- IR&D
- IAR
- ACC
- ALU
Program Counter Schematic (4 bit)

input mux

flip flop

increment logic

reset logic

tri-state buffer
Registers and Counters

- A register is a set of flip flops, often supplemented by additional circuits to control input and output.
  - can have parallel I/O or serial I/O or combination

- Usually, registers are used to store a set of related bits.
  - bits that collectively represent an integer value
  - bits of an ASCII character code
  - status bits for a device in a computer system (disk controller)

- Counters are registers that store numeric values along with circuits to increment/ decrement the stored value.
  - up-counters, down-counters, up-down counters
  - generalized counters
    - BCD counters, gray-code counters, ...
Simple Parallel Load Register

- Four bit register.
  - if LD is high when clock rises, new values are stored
  - LD should change only while CLK is high
- Registers using gated clocks can lead to timing problems.
  - increases clock skew
  - may lead to violations of flip flop setup, hold time specs
  - extra care needed to ensure correct operation
  - safer to avoid clock gating whenever possible
Preferred Parallel Load Register

- Multiplexor for each register bit.
  - new value loaded when LD is asserted
  - otherwise, old value stored
- No gated clock, minimizing clock skew.
  - simplifies checking of setup and hold time specs.
  - can focus on delays between connected flip flops
- Increases gate count by about 30%.
**VHDL Specification for Register**

- Register stores new input value when \( \text{ld} \) is high, otherwise retains old value.
- Vector notation makes arbitrary register sizes easy.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity reg4u is
    port(
        d: in STD_LOGIC_VECTOR(3 downto 0);
        ld, clk: in STD_LOGIC;
        q: out STD_LOGIC_VECTOR(3 downto 0)
    );
end reg4u;

architecture reg4u_arch of reg4u is
begin
    signal reg: STD_LOGIC_VECTOR(3 downto 0);
    process(clk) begin
        if clk'event and clk = '1' then
            if ld = '1' then
                q <= d;
            end if;
        end if;
    end process;
    q <= reg;
end reg4u_arch;
```
Shift Registers

- Shift registers support serial input and output.
  - useful for communication over serial channels
- With parallel outputs, can be used for serial-to-parallel conversion.
- With parallel inputs can be used for parallel-to-serial conversion.
  - requires 3 input muxes and second control input
library IEEE;
use IEEE.std_logic_1164.all;

entity sreg is
    port ( 
        d: in STD_LOGIC_VECTOR(3 downto 0);
        ld, sl, sr: in STD_LOGIC;
        clk: in STD_LOGIC;
        q: out STD_LOGIC_VECTOR(3 downto 0) 
    );
end sreg;

architecture sreg_arch of sreg is
    signal reg: STD_LOGIC_VECTOR(3 downto 0);
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if ld = '1' then
                if sl = '1' then
                    reg <= reg(2 downto 0) & d(0);
                elsif sr = '1' then
                    reg <= d(3) & reg(3 downto 1);
                else
                    reg <= d;
                end if;
            end if;
        end if;
    end process;
    q <= reg;
end sreg_arch;

- ld enables loading.
  » if sl asserted then left shift
  » else if sr asserted then right
  » else parallel load
Synchronous Ripple Carry Counter

- Change in low order bit can affect carry in all higher bits.
- No problem, so long as carry stable by next rising clock edge.
- Can be too slow for counters with many bits.
Counter with Carry Look-ahead

- Carries sent forward to eliminate carry propagation delay.
- In large counters, carry logic becomes major part of counter complexity.
- Large fanout of carry signals limit performance gains.
  » insert buffers to compensate
More Scalable Carry Look-ahead

- Circuit complexity grows as $n \log n$.
- Small fanout for all gates.

1. First rank ANDs adjacent outputs
2. Second rank ANDs last four outputs
3. Third rank ANDs last eight outputs
4. Max carry delay grows logarithmically
Up-down Counter with Parallel Load

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ud_cntr is
  port (
    d: in STD_LOGIC_VECTOR(3 downto 0);
    ld, cnt, up, clk: in STD_LOGIC;
    q: out STD_LOGIC_VECTOR(3 downto 0)
  );
end ud_cntr;

architecture ud_cntr_arch of ud_cntr is
signal reg: STD_LOGIC_VECTOR(3 downto 0);
begnin
  process(clk) begin
    if clk'event and clk = '1' then
      if ld = '1' then
        reg <= d;
      elsif cnt = '1' then
        if up = '1' then
          reg <= reg + "0000"
        else
          reg <= reg - "0000"
        end if;
      end if;
    end if;
  end process;
  q <= reg;
end ud_cntr_arch;

- IEEE.std_logic_unsigned package enables arithmetic on logic vectors.
- Straightforward extension to larger word sizes.
- Synthesizer generates carry logic and balances size vs. speed objectives.
Non-Standard Counters

- Counters are sometimes defined that count in an order other than standard numerical order.
- The state machine below is for a gray code counter in which one bit changes at a time.
VHDL for Gray Code Counter

- Case statement allows compact expression of alternatives.
  - others clause specifies action when no other alternative applies
  - needed here due to non-binary values of STD_LOGIC
- Note that synthesis tool generates all the next-state logic.

library IEEE;
use IEEE.std_logic_1164.all;

entity gray_cnt is
  port ( 
    clr, cnt, clk: in STD_LOGIC;
    q: out STD_LOGIC_VECTOR(2 downto 0)
  );
end gray_cnt;

architecture gray_cnt_arch of gray_cnt is
  signal reg: STD_LOGIC_VECTOR(2 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if clr = '1' then reg <= "000";
      elsif cnt = '1' then
        case reg is
          when "000" => reg <= "001";
          when "001" => reg <= "011";
          when "011" => reg <= "010";
          when "010" => reg <= "110";
          when "110" => reg <= "111";
          when "111" => reg <= "101";
          when "101" => reg <= "100";
          when "100" => reg <= "000";
          when others => reg <= "000";
        end case;
    end if;
  end if;
end process;
q <= reg;
end gray_cnt_arch;
Larger Sequential Circuit Design

- Traffic light controller
  - T intersection
  - default to green on main road
  - sensor enables green for cross street
  - delay switching for right-turn-on-red from cross street
  - programmable delays
High Level State Machine

- Stay in thruG state until sensor is activated.
- Wait in pause state to see if sensor deactivates.
  (right-turn-on-red)
- Then proceed through sequence, waiting in each state for specified time delay.
- In each state, provide appropriate control signals for lights.
- 5 states, so at least 3 flip flops.
Controller keeps track of state and turns lights on/off.
  - \( t_G \) means thruGreen, etc.
  - State assignment
    - 000 = thruG, 001 = pause
    - 010 = thruY, 011 = thruR
    - 100 = crossY

Counter used to regulate delays.
  - Set to zero when not enabled

Dials at left specify delays.
  - Note how clock frequency affects delay values and counter size
Detailed State Diagram and State Table

- **Output equations**
  \[ t_G = s_2's_1 \quad t_Y = s_1's_0' \quad t_R = (t_G + t_Y)' \]
  \[ x_G = s_1's_0 \quad x_Y = s_2 \quad x_R = (x_G + x_Y)' \]
  \[ TEN = s_1's_0's_1'd_1's_0'd_2' + s_1's_0'd_3' + s_2'd_4' \]

- **Next state equations**
  \[ ns_2 = s_1's_0'd_3 + s_2'd_4' \]
  \[ ns_1 = s_1's_0's_1'd_1 + s_1's_0' + s_1's_0'd_3' \]
  \[ ns_0 = s_2's_1's_0'S + s_1's_0's_1'd_1 + s_1's_0'd_2 + s_1's_0'd_3' \]

<table>
<thead>
<tr>
<th>current state</th>
<th>inputs</th>
<th>outputs</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_2s_1s_0</td>
<td>Sd_1d_2d_3d_4</td>
<td>TEN t_G t_Y t_R x_G x_Y x_R</td>
<td>ns_2ns_1ns_0</td>
</tr>
<tr>
<td>000</td>
<td>0xxxx</td>
<td>010001</td>
<td>000</td>
</tr>
<tr>
<td>000</td>
<td>1xxxx</td>
<td>010001</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td>0xxxx</td>
<td>010001</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>10xxxx</td>
<td>110001</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td>11xxxx</td>
<td>010001</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>x0xxx</td>
<td>101001</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>x1xxx</td>
<td>001001</td>
<td>011</td>
</tr>
<tr>
<td>011</td>
<td>xx0x</td>
<td>1001100</td>
<td>011</td>
</tr>
<tr>
<td>011</td>
<td>xx1x</td>
<td>001100</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>x0x</td>
<td>1001010</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>x1x</td>
<td>001010</td>
<td>000</td>
</tr>
<tr>
<td>100</td>
<td>xxx0</td>
<td>1001010</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>xxx1</td>
<td>001010</td>
<td>000</td>
</tr>
</tbody>
</table>
Schematic for Traffic Light Controller

- comparators
- input delays from switches
- state machine
- timer
Simulation of Traffic Light Controller

thruG

pause

thruY

thruR

crossY

pause
VHDL for Traffic Light Controller

library IEEE; use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity trafficv is
  port (reset, sensor, CLK: in STD_LOGIC;
d1,d2,d3,d4: in STD_LOGIC_VECTOR(3 downto 0);
tG, tY, tR, xG, xY, xR: out STD_LOGIC
  );
end trafficv;

architecture trafficv_arch of trafficv is
  type state_type is (thruG,pause,thruY,thruR,crossY);
signal state: state_type;
signal timer: STD_LOGIC_VECTOR(3 downto 0);
signal TEN: STD_LOGIC_VECTOR(3 downto 0);
begin
  timer_process: process(clk) begin
    if clk'event and clk = '1' then
      if TEN = '0' then timer <= "0000";
        else timer <= timer + "0001";
      end if;
    end if;
  end process;

  timer process
  counts when
  enabled
VHDL for Traffic Light Controller

```vhdl
state_machine_process: process(clk) begin
    if clk'event and clk = '1' then
        if reset = '1' then
            state <= thruG; TEN <= '0';
        else
            if state = thruG and sensor = '0' then
                state <= thruG; TEN <= '0';
            elsif state = thruG and sensor = '1' then
                state <= pause; TEN <= '0';
            elsif state = pause and sensor = '0' then
                state <= thruG; TEN <= '0';
            elsif state = pause and sensor = '1' and timer /= d1 then
                state <= pause; TEN <= '1';
            elsif state = pause and sensor = '1' and timer = d1 then
                state <= thruY; TEN <= '0';
            elsif state = thruY and timer /= d2 then
                state <= thruY; TEN <= '1';
            elsif state = thruY and timer = d2 then
                state <= thruR; TEN <= '0';
            elsif state = thruR and timer /= d3 then
                state <= thruR; TEN <= '1';
            elsif state = thruR and timer = d3 then
                state <= crossY; TEN <= '0';
            elsif state = crossY and timer /= d4 then
                state <= crossY; TEN <= '1';
            elsif state = crossY and timer = d4 then
                state <= thruG; TEN <= '0';
        end if;
    end if;
end process;
```
VHDL for Traffic Light Controller

tG <= '1' when (state = thruG) or (state = pause) else '0';
tY <= '1' when (state = thruY) else '0';
tR <= '1' when (state = thruR) else '0';
xG <= '1' when (state = thruR) else '0';
xY <= '1' when (state = crossY) else '0';
xR <= '1' when (state = thruG) or (state = pause) or (state = thruY) else '0';
end trafficv_arch;

outputs specified by concurrent assignments
Metastability

- Most synchronous systems have asynchronous inputs.
  - keyboard input on a computer,
  - sensor on a traffic light controller,
  - card insertion on an ATM, etc.

- Asynchronous inputs change at unpredictable times.
  - so, can change during clock transition, causing metastability

- Output of a metastable flip flop can oscillate causing unpredictable behavior in other flip flops.
  - oscillations usually end quickly, but no definite time limit
  - so, circuit failures due to metastability are unavoidable
  - however, systems can be designed to make failures rare
Synchronizers

- Synchronizers are used to isolate metastable signals until they are “probably safe.”

- If the clock period is long enough, failure probability is small and expected time between failures is large.

  \[ \text{MTBF} = \text{Mean Time Between Failures} \approx \left( \frac{\alpha T}{T_0} \right) e^{T/\tau} \]

  where \( T \) is the clock period, \( \alpha \) is the average time between asynchronous input changes, \( \tau \) and \( T_0 \) are parameters of the flip flop being used.

- If \( T = 50 \text{ ns}, \alpha = 1 \text{ ms}, \tau = 1 \text{ ns}, T_0 = 1 \mu \text{ s}, \text{MTBF} \approx 8 \text{ billion years} \)