Introduction to Sequential Circuits

- Basic Sequential Circuit Design
- Latches and Flip Flops
- Sequential Circuit Analysis
- General Circuit Design Method
- Designing Sequential Circuits with VHDL
Program Counter Schematic (4 bit)

- input mux
- flip flop
- increment logic
- tri-state buffer
- reset logic
- same inputs, different outputs
entity program_counter is
  port (
    en_A, ld, inc, reset: in STD_LOGIC;
    aBus: out STD_LOGIC_VECTOR(15 downto 0);
    dBus: in STD_LOGIC_VECTOR(15 downto 0)
  );
end program_counter;
architecture pcArch of program_counter is
begin
  signal pcReg: STD_LOGIC_VECTOR(15 downto 0);
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then pcReg <= "0000000000000000";
      elsif ld = '1' then pcReg <= dBus;
      elsif inc = '1' then
        pcReg <= pcReg + "0000000000000001";
      end if;
    end if;
  end process;
  aBus <= pcReg when en_A = '1' else "ZZZZZZZZZZZZZZZZZ";
end pcArch;
VHDL PC Simulation

- Increment
- Load
- Enable output
In **sequential circuits**, output values may depend on both current and past input values.

- consists of combinational circuit and set of storage elements
- each storage element stores one bit of information
- the **state** of a sequential circuit is the set of stored values

In **clocked sequential circuits**, state changes are driven by clock signals.

Information stored using **flip-flops**.
Edge-Triggered D Flip Flop

- D flip flop stores value at D input when clock rises.
- Most widely used storage element for sequential circuits.
- Propagation time is time from rising clock to output change.
- If input changes when clock rises, new value is uncertain.
  » output may oscillate (metastability)
- Timing rules to avoid metastability
  » D input must be stable for setup time before rising clock edge
  » must remain stable for hold time following rising clock edge
Serial Parity Generator

Output is high if number of 1's in input stream is odd.

Cleared when enable is low.

Next state table gives next state and output, as function of current state and input.
To meet setup & hold time requirements of flip flop, inputs to circuit must be stable during certain times.

» let setup time=2 ns, hold time=1 ns and gate delay=1 ns

» then D must be stable from 4 ns before clock edge until 1 ns before clock edge; similarly for EN

» unshaded period in timing diagram shows when inputs must be stable

» if gate delay can vary between .4 and 1.5 ns, then stable period for D is from 5 ns before clock edge to .2 ns after
The SR Latch

- Pair of inverters provides stable storage.
- To enable stored value to be changed, use cross-coupled NOR gates.
  » equivalent to inverter pair when both inputs are low
- SR latch is key building block for flip flops.
  » when \( S = 1 \), \( R = 0 \) flip-flop is **set**
  » when \( S = 0 \), \( R = 1 \) flip-flop is **reset**
  » when \( S = 0 \), \( R = 0 \) flip-flop retains value
  » when \( S = 1 \), \( R = 1 \) flip-flop state is undefined

<table>
<thead>
<tr>
<th>( S )</th>
<th>( R )</th>
<th>( Q(t+1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( Q(t) )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>??</td>
</tr>
</tbody>
</table>
S-R Latch Behavior

- Note that when $S=R=1$, both outputs are low.
  - Outputs are not complements of each other in this case.
- When $S$, $R$ drop together, the latch can oscillate indefinitely.
  - 1 ns gate delay used in simulation.
- Such oscillations can lead to unpredictable circuit behavior.
- For these reasons, the $S=R=1$ condition should be avoided.
More on SR Latches

- SR latch most often implemented with NAND gates.
  - inputs are active low (negative logic inputs)
  - when both inputs are low, both outputs high
  - when inputs rise together, outputs oscillate

- SR latch with control input changes state only when control input is high.
  - inputs are active high
  - forbidden input condition is \( C=S=R=1 \)
  - change \( S, R \) inputs when \( C=0 \)

<table>
<thead>
<tr>
<th>( SR )</th>
<th>( Q(t+1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>??</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>( Q(t) )</td>
</tr>
</tbody>
</table>
The SR master-slave flip flop uses two latches with complementary enables.

First stage follows all changes while clock is high, but second stage only "sees" value just before clock drops.

Forbidden input combination causes oscillations.

Recommended usage: change $S$, $R$ only when $C=0$. 
The D latch stores the value on the D input when the enable input is asserted.
- no forbidden input combinations
- but input should be stable when the control input drops
- if not, oscillations may occur - metastability

Alternative implementation uses transmission gates.
- TGs enable either input or feedback path
- in CMOS, this uses 10 transistors vs. 18
Implementing D Flip Flops

- When clock rises, value in D-latch propagates to SR-latch and outputs.
- Same structure as SR master-slave, but behaves differently.
  - SR master-slave flip flop affected by changes while clock high, not just at transition
- Flip flop setup and hold time conditions designed to prevent metastability in latches.
- Propagation delay determined by SR-latch delay
Types of Latches and Flip Flops

Standard Graphic Symbols

Latches

Master-Slave Flip Flops

Edge-Triggered Flip Flops

Characteristic Tables

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q(t)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Q′(t)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q(t)</td>
</tr>
<tr>
<td>1</td>
<td>Q′(t)</td>
</tr>
</tbody>
</table>
The behavior of a sequential circuit can be defined by a state table, which specifies:

- how inputs cause state transitions and
- the outputs produced by the circuit in different conditions.

The following state table describes a sequential circuit with two flip flops.

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B</td>
<td>X</td>
<td>D_A D_B</td>
<td>Y</td>
</tr>
<tr>
<td>0  0</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0  0</td>
<td>1</td>
<td>1 0</td>
<td>1</td>
</tr>
<tr>
<td>0  1</td>
<td>0</td>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>0  1</td>
<td>1</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1  0</td>
<td>0</td>
<td>1 1</td>
<td>0</td>
</tr>
<tr>
<td>1  0</td>
<td>1</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1  1</td>
<td>0</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1  1</td>
<td>1</td>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>
Analyzing Sequential Circuits

- Analysis involves finding the specification (e.g. state table) for a given sequential circuit.

**Procedure**
1. Name inputs, outputs and flip flops.
2. Format table.
3. Fill in output columns.
4. Fill in next state columns.

**Alternative specification.**
- Output equations
  \( Y = A \cdot B + X \)
- Next state equations
  \( D_A = A \oplus B \oplus X, \quad D_B = A \cdot B \cdot X \)
In some sequential circuits, the outputs are functions of the current state only.

- These are called **Moore model circuits**
- General sequential circuits are called **Mealy model circuits**

State diagrams are equivalent to state tables.

**Mealy Model State Diagram**

**Moore Model State Diagram**
Sequential Circuit Design Procedure

- State machine specifications are often given in English.
  - e.g. design a circuit whose output is 1, if the number of logic “1”s seen in a sequential input stream is odd.
  - designer’s job is to find a circuit that does this

- Construct a state table or state diagram, then.
  1. Determine number of states and number of flip flops needed (at least $2^s$ where $s$ is number of states).
  2. Assign **encoding** for each state.
  3. Determine logic equations for each output signal.
  4. Determine logic equation for each flip flop input.
A sequential comparator has two data inputs \((A, B)\), an enable input \((E)\) and a single output \((A > B)\).

- When enable is low, the output is zero.
- When enable is high, the circuit compares \(A\) and \(B\) numerically (assuming the values are presented with the most-significant bit, first) and outputs 1 if \(A > B\).

**Example:**

```
CLK  E  A  B  A>B
000  0  0  0  1
001  1  0  0  0
010  1  1  0  0
011  0  0  0  1
100  1  1  0  0
101  0  0  1  0
110  0  1  1  1
111  1  0  1  0
```

- \(EAB\)
- \(A > B\)
- \(A < B\)
Sequential Comparator Design

- **Output equation:**
  \[ A > B = s_1s_0' \] (simplify to \( s_1 \))

- **Next state equations:**
  \[ D_{s1} = (s_1 + s_1's_0'A'B')E \]
  \[ = (s_1 + s_0'AB')E \]
  \[ D_{s0} = (s_0 + s_1's_0'A'B')E \]
  \[ = (s_0 + s_1'A'B')E \]

Three states implies at least 2 flip flops. One encoding is:
- 00 for ??
- 10 for \( A > B \)
- 01 for \( A < B \)

<table>
<thead>
<tr>
<th>Present State ( s_1s_0 )</th>
<th>Inputs EAB</th>
<th>Next State ( s_1s_0 )</th>
<th>Output ( A &gt; B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0xx</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>111</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>101</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1xx</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0xx</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1xx</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0xx</td>
<td>00</td>
<td>0</td>
</tr>
</tbody>
</table>
Verifying Sequential Circuits

To fully verify a sequential circuit, must check all state transitions (including “non-transitions”).

» use state diagram to plan input sequence
» for transitions with don’t cares, check all possibilities

1. check all self-loops in 00.
2. switch to 10 and check self-loops
3. check transitions back to 00
4. switch to 01 and check self-loops
5. check transitions back to 00
Timing Analysis of Sequential Circuits

- Determine if circuit subject to internal hold time violations; if so, eliminate by adding delay.
- Ignoring input signals, find smallest clock period for which setup time conditions are always met.
- Determine time periods (relative to clock edge) during which inputs must be stable.
- Determine time periods (relative to clock edge) during which outputs may be changing.
- Input and output conditions used to ensure that connected sequential circuits interoperate correctly.
  » if circuit A connects to circuit B, verify that output of A is not changing when B requires that its input be stable
  » simplifies timing analysis of larger circuits
Timing Analysis Procedure

- Internal hold time violations.
  - for every ff-to-ff path, check
    \((\text{ff prop. delay}) + (\text{comb. circuit delay}) > (\text{hold time}) + (\text{clock skew})\)
  - use minimum values for delays

- Minimum clock period
  - find ff-to-ff path with largest value of
    \((\text{ff prop. delay}) + (\text{comb. circuit delay}) + (\text{setup time}) + (\text{clock skew})\)
  - use maximum values for delays

- Input timing analysis
  - each input must be stable from
    \((\text{clock}_\text{edge}) - ((\text{max. input-to-ff delay}) + (\text{setup time}))\)
    to \(((\text{clock}_\text{edge}) + (\text{hold time})) - (\text{min. input-to-ff delay})\)

- Output timing analysis
  - outputs can change from
    \((\text{clock}_\text{edge}) + (\text{min delay})\) to \((\text{clock}_\text{edge}) + (\text{max delay})\)

- To account for uncertainty in clock edge timing, extend input and output timing ranges by clock skew.
Timing Analysis of Sequential Comparator

- Let gate delay be .4 to 1.5 ns.
- Let ff setup time be 2 ns, hold time 1 ns, prop. delay 1 to 3 ns.
- Let clock skew be 1 ns.
- Internal hold time violation?
  » yes - $1 + 2(\cdot.4) < 1 + 1$
  » add inverter pair to feedback inputs of ORs
- Minimum clock period - $3 + 4 \times 1.5 + 2 + 1 = 12$ ns.
- Input timing requirements
  » $A$ and $B$ must be stable from \texttt{clock\_edge}− (2 + 4×1.5) until \texttt{clock\_edge}+(1 − 3 × .4), so from 8 ns before clock edge to .2 ns
  » if clock can change anytime between −1 and +1, inputs must be stable from −9 ns to +.8 ns
- Output timing - outputs can change from 1 to 3 ns after clock.
Serial Subtraction Circuit

- A serial subtraction circuit has two data inputs (A, B), an enable input (E) and a single output (A − B).
  - when enable is low, the output is zero
  - when enable is high, the circuit subtracts B from A numerically (assuming the values are presented with the least-significant bit, first) and outputs the difference, serially.

**Example:**

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
<th>B</th>
<th>A − B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
<th>B</th>
<th>A − B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Diagrams:**

- Waveforms for E, A, B, and A − B.
- Flowchart for borrow and no borrow cases.
Serial Subtractor Design

Output FF for synchronous output
Sequential Circuits in VHDL

library IEEE;
use IEEE.std_logic_1164.all;

entity parityv is
  port ( 
    clk, en, x: in STD_LOGIC;
    y: out STD_LOGIC
  );
end parityv;

architecture parityv_arch of parityv is
signal s: std_logic;
begin
  process (clk) begin
    if clk'event and clk = '1' then
      if en = '0' then 
        s <= '0';
      else 
        s <= x xor s;
      end if;
    end if;
  end process;
y <= s;
end parityv_arch;

- Process construct used to define more complex logic.
- Test for rising clock edge synchronizes actions.
- If statement used to express complex sets of conditions.
- Internal signal definition.
- Storage implied since new signal value depends on previous value.
In serial parity circuit, signal \( s \) must be stored, since its value depends on its previous value.

In other cases, need for storage may not be obvious.

```vhdl
architecture foo of foobar is
  signal s: std_logic;
begin
  process (clk) begin
    if clk'event and clk = '1' then
      if a = '1' then
        if a = '1' then
          s <= b;
        end if;
      end if;
    end if;
  end process;
end foo;
```

Subtle dependence of \( s \) on previous value, since it must keep old value if \( a \neq 1 \).
Serial Comparator in VHDL

- Same basic structure as serial parity circuit.
  - signals for flip flops
  - if defines next state logic
    - no change to s1, s0 when none of specified conditions holds
    - so, no code needed for self-loops in state diagram
  - signal assign. for outputs
Simpler Form of Seq. Comparator

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

entity seqcmpv is
  port (  
    A, B, E, Clk: in STD_LOGIC;
    A_gt_B: out STD_LOGIC
  );
end seqcmpv;

architecture seqcmpv_arch of seqcmpv is
  type state_type is (unknown, Abigger, Bbigger);
signal state: state_type;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if E = '0' then
        state <= unknown;
      elsif state = unknown then
        if A = '1' and B = '0' then
          state <= Abigger;
        elsif A = '0' and B = '1' then
          state <= Bbigger;
        end if;
      end if;
    end if;
  end process;
  A_gt_B <= '1' when state = Abigger else '0';
end seqcmpv_arch;
```

State type with named values.

- use of state names makes code easier to understand
- synthesizer can optimize state assignment
Flip Flops with Asynchronous Resets

- To simplify initialization, flip flops are often equipped with asynchronous resets.
  - asynchronous resets clear flip flop independent of clock
- D flip flop with asynchronous reset.

![D flip flop with asynchronous reset diagram]

- To specify asynchronous initialization of a sequential circuit in VHDL, different code is needed.
Asynchronous Resets in VHDL

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

entity parityv is
  port (  
    clk, reset, x: in STD_LOGIC;
    y: out STD_LOGIC
  );
end parityv;

architecture parityv_arch of parityv is
begin
  process(clk,reset) begin
    if reset = '1' then
      s <= '0';
    elsif clk'event and clk = '1' then
      s <= x xor s;
    end if;
  end process;
  y <= s;
end parityv_arch;
```

- The process responds to changes in `clk` and `reset`.
- Initialization does not depend on `clk`.
- Normal state changes only allowed when `reset=0`.
Common Forms for State Machines

- process begin
  wait until clk = '1'  
  ...  
  end process;

- process(clk) begin  
  if clk'event and clk='1' then
    ...  
  end if;
  end process;

- process(clk,reset) begin
  if reset='1' then
    ...  
  elsif clk'event and clk='1' then
    ...  
  end if;
  end process;
Recommended Procedure for State Machines

1. Determine the inputs and outputs of your circuit.
2. Determine and name the states in your circuit.
3. Create entity with all of inputs, outputs, reset and clock.
4. Create an enumerated state_type with all your state names; example: `type state_type is (start, add, shift, wait)`
5. Write a process that will update the present state with the next state on a clock edge.
   ```
   process(clk,reset) begin
     if reset = '1' then
       -- add reset logic here
     elsif clk'event and clk = '1' then
       -- add next state logic here
     end if;
   end process;
   ```
6. Outside the process, write concurrent, logical statements for each output based on the states and, as necessary, the inputs.
Forms that **Should** Work but **Don't**

- process(clk) begin
  
  if clk='1' then
    ...  
  end if;
  
  end process;

- process begin
  
  ...  
  
  wait on clk;
  
  end process;

- process(a,b) begin
  
  if a'event then
    ...  
  end if;
  
  end process;

- process begin
  
  wait until clk='1'
  
  ...  
  
  wait until clk='0'  
  
  ...  
  
  end process;

  -- synthesized circuit responds while clk='1'

  -- not allowed by synthesizer

  -- not allowed by synthesizer