Sequential Basics

- Sequential circuits
  - Outputs depend on current inputs and previous inputs
  - Store *state*: an abstraction of the history of inputs
  - Usually governed by a periodic clock signal
D-Flipflops

- 1-bit storage element
  - We will treat it as a basic component

- Other kinds of flipflops
  - SR (set/reset), JK, T (toggle)

Registers

- Store a multi-bit encoded value
  - One D-flipflop per bit
  - Stores a new value on each clock cycle

```vhdl
signal d, q: ...;
...
reg: process (clk) is
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process reg;
```
### Pipelines Using Registers

Total delay = Delay\(_1\) + Delay\(_2\) + Delay\(_3\)

Interval between outputs > Total delay

\[
\text{Total delay} = 3 \times \text{clock period}
\]

Interval between outputs = 1 clock period

---

### Pipeline Example

- Compute the average of corresponding numbers in three input streams
- New values arrive on each clock edge

```vhdl
library ieee;
use ieee.std_logic_1164.all, ieee.fixed_pkg.all;
entity average_pipeline is
  port ( clk : in std_logic;
             a, b, c : in sfixed(5 downto -8);
             avg : out sfixed(5 downto -8) );
end entity average_pipeline;
```
Pipeline Example

architecture rtl of average_pipeline is
  signal a_plus_b, sum, sum_div_3 : sfixed(5 downto -8);
  signal saved_a_plus_b, saved_c, saved_sum : sfixed(5 downto -8);
begin
  a_plus_b <= a + b;
  reg1 : process (clk) is
    begin
      if rising_edge(clk) then
        saved_a_plus_b <= a_plus_b;
        saved_c <= c;
      end if;
    end process reg1;
  ...
  sum <= saved_a_plus_b + saved_c;
  reg2 : process (clk) is
    begin
      if rising_edge(clk) then
        saved_sum <= sum;
      end if;
    end process reg2;
  sum_div_3 <= saved_sum * to_fixed(1.0/3.0, sum_div_3'left, sum_div_3'right);
  reg3 : process (clk) is
    begin
      if rising_edge(clk) then
        avg <= sum_div_3;
      end if;
    end process reg3;
  end architecture average_pipeline;
D-Flipflop with Enable

- Storage controlled by a clock-enable
  - stores only when CE = 1 on a rising edge of the clock

---

Cluster with Enable

- One flipflop per bit
- clk and CE wired in common

```vhdl
signal d, q: ...;
...
reg: process (clk) is
begin
  if rising_edge(clk) then
    if ce = '1' then
      q <= d;
    end if;
  end if;
end process reg;
```
Register with Synchronous Reset

- Reset input forces stored value to 0
- reset input must be stable around rising edge of clk

\[
\begin{array}{c|c|c|c}
\text{D} & \text{Q} & \text{CE} & \text{clk} \\
\hline
\text{reset} & & & \\
\end{array}
\]

Synch Reset in VHDL

**single-bit data**

```vhdl
flipflop: process (clk) is
begin
  if rising_edge(clk) then
    if reset = '1' then
      q <= '0';
    elsif ce = '1' then
      q <= d;
    end if;
  end if;
end process flipflop;
```

**vector data**

```vhdl
reg: process (clk) is
begin
  if rising_edge(clk) then
    if reset = '1' then
      q <= "00000000";
    elsif ce = '1' then
      q <= d;
    end if;
  end if;
end process reg;
```
Register with Asynchronous Reset

- Reset input forces stored value to 0
  - reset can become 1 at any time, and effect is immediate
  - reset should return to 0 synchronously

Asynch Reset in VHDL

```vhdl
reg: process (clk, reset) is
begin
  if reset = '1' then
    q <= "00000000";
  elsif rising_edge(clk) then
    if ce = '1' then
      q <= d;
    end if;
  end if;
end process reg;
```

- reset is an asynchronous control input here
  - include it in the sensitivity list so that the process responds to changes immediately
Example: Accumulator

- Sum a sequence of signed numbers
  - A new number arrives when data_en = 1
  - Clear sum to 0 on synch reset

```vhdl
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;
entity accumulator is
  port ( clk, reset, data_en : in std_logic;
        data_in : in signed(15 downto 0);
        data_out : out signed(19 downto 0) );
end entity accumulator;
```

```vhdl
architecture rtl of accumulator is
  signal sum, new_sum : signed(19 downto 0);
begin
  new_sum <= sum + resize(data_in, sum'length);
  reg: process (clk) is
    begin
      if rising_edge(clk) then
        if reset = '1' then
          sum <= (others => '0');
        elsif data_en = '1' then
          sum <= new_sum;
        end if;
      end if;
    end process reg;
  data_out <= sum;
end architecture rtl;
```
Flipflop and Register Variations

```
library ieee;
use ieee.std_logic_1164.all;
entity flipflop_n is
port (clk_n, CE, pre_n, clr_n, D : in std_logic;
Q, Q_n : out std_logic );
end entity flipflop_n;
architecture behavour of flipflop_n is
begin
ff: process (clk_n, pre_n, clr_n) is
begin
assert not ( pre_n = '0'
and clr_n = '0')
report "Illegal inputs:" & "pre_n and clr_n both '0';"
if pre_n = '0' then
Q <= '1'; Q_n <= '0';
elsif clr_n = '0' then
Q <= '0'; Q_n <= '1';
elsif falling_edge(clk_n) then
if CE = '1' then
Q <= D; Q_n <= not D;
end if;
end if;
end process ff;
end architecture behavour;
```

Shift Registers

- Performs shift operation on stored data
  - Arithmetic scaling
  - Serial transfer of data

```
D_in D(n-1)
D(n-2) D
D(0) D
CE
clk
load_en
Q

D_in
D(n-1)
D(n-2)
D(0)
D Q
CE
clk
load_en
Q

D Q
CE
clk
Q

D Q
CE
clk
Q
```

Digital Logic Design. Chapter 4
Example: Sequential Multiplier

- 16×16 multiply over 16 clock cycles, using one adder
  - Shift register for multiplier bits
  - Shift register for lsb's of accumulated product

Latches

- Level-sensitive storage
  - Data transmitted while enable is '1'
    - transparent latch
  - Data stored while enable is '0'
Feedback Latches

- Feedback in gate circuits produces latching behavior
  - Example: reset/set (RS) latch
    
    ![RS latch diagram]

- Current RTL synthesis tools don’t accept VHDL models with unclocked feedback

Latches in VHDL

- Latching behavior is usually an error!

  ```vhdl
  mux_block : process (sel, a1, b1, a2, b2) is
  begin
    if sel = '0' then
      z1 <= a1; z2 <= b1;
    else
      z1 <= a2; z3 <= b2;
    end if;
  end process mux_block;
  
  Oops!
  Should be
  z2 <= ...
  
  Values must be stored
  - for z2 while sel = '1'
  - for z3 while sel = '0'
  ```
Counters

- Stores an unsigned integer value
  - increments or decrements the value
- Used to count occurrences of
  - events
  - repetitions of a processing step
- Used as timers
  - count elapsed time intervals by incrementing periodically

Free-Running Counter

- Increments every rising edge of clk
  - up to $2^n - 1$, then wraps back to 0
  - i.e., counts modulo $2^n$
- This counter is *synchronous*
  - all outputs governed by clock edge
Example: Periodic Control Signal

- Count modulo 16 clock cycles
  - Control output = '1' every 8th and 12th cycle
  - decode count values 0111 and 1011

```vhdl
library ieee; use ieee.std_logic_1164.all, ieee.numeric_std.all;

entity decoded_counter is
  port ( clk : in std_logic; ctrl : out std_logic );
end entity decoded_counter;

architecture rtl of decoded_counter is
  signal count_value : unsigned(3 downto 0);
begin
  counter : process (clk) is
  begin
    if rising_edge(clk) then
      count_value <= count_value + 1;
    end if;
  end process counter;

  ctrl <= '1' when count_value = "0111" or count_value = "1011" else '0';
end architecture rtl;
```

Digital Logic Design. Chapter 4
Count Enable and Reset

- Use a register with control inputs
  
  ![Diagram of a register with control inputs](image)

  - Increments when CE = '1' on rising clock edge
  - Reset: synch or asynch

Terminal Count

- Status signal indicating final count value
  
  ![Diagram of a terminal count](image)

  - TC is '1' for one cycle in every $2^n$ cycles
    - frequency = clock frequency / $2^n$
  - Called a clock divider
Divider Example

- Alarm clock beep: 500Hz from 1MHz clock

Divide by \( k \)

- Decode \( k-1 \) as terminal count and reset counter register
  - Counter increments modulo \( k \)
- Example: decade counter
  - Terminal count = 9
Decade Counter in VHDL

```vhdl
library ieee; use ieee.std_logic_1164.all, ieee.numeric_std.all;
entity decade_counter is
  port ( clk : in std_logic; q : out unsigned(3 downto 0) );
end entity decade_counter;

architecture rtl of decade_counter is
  signal count_value : unsigned(3 downto 0);
  begin
    count : process (clk) is
    begin
      if rising_edge(clk) then
        count_value <= (count_value + 1) mod 10;
      end if;
    end process count;
    q <= count_value;
end architecture rtl;
```

Down Counter with Load

- Load a starting value, then decrement
- Terminal count = 0
- Useful for interval timer
Loadable Counter in VHDL

```vhdl
library ieee; use ieee.std_logic_1164.all, ieee.numeric_std.all;
entity interval_timer is
  port ( clk, load : in std_logic; data : in unsigned(9 downto 0);
         tc : out std_logic);
end entity interval_timer;
architecture rtl of decade_counter is
  signal count_value : unsigned(9 downto 0);
begin
  count : process (clk) is
  begin
    if rising_edge(clk) then
      if load = '1' then
        count_value <= data;
      else
        count_value <= count_value - 1;
      end if;
    end if;
  end process count;
  tc <= '1' when count_value = 0 else '0';
end architecture rtl;
```

Reloading Counter in VHDL

```vhdl
architecture repetitive of down_counter is
  signal load_value, count_value : unsigned(9 downto 0);
begin
  count : process (clk) is
  begin
    if rising_edge(clk) then
      if load = '1' then
        load_value <= data;
        count_value <= data;
      elsif count_value = 0 then
        count_value <= load_value;
      else
        count_value <= count_value - 1;
      end if;
    end if;
  end process count;
  tc <= '1' when count_value = 0 else '0';
end architecture repetitive;
```
Ripple Counter

- Each bit toggles between 0 and 1
  - when previous bit changes from 1 to 0

Ripple or Synch Counter?

- Ripple counter is ok if
  - length is short
  - clock period long relative to flipflop delay
  - transient wrong values can be tolerated
  - area must be minimal
  - E.g., alarm clock
- Otherwise use a synchronous counter
Datapaths and Control

- Digital systems perform sequences of operations on encoded data
- **Datapath**
  - Combinational circuits for operations
  - Registers for storing intermediate results
- **Control section**: control sequencing
  - Generates *control signals*
    - Selecting operations to perform
    - Enabling registers at the right times
  - Uses *status signals* from datapath

Example: Complex Multiplier

- Cartesian form, fixed-point
  - Operands: 4 pre-, 12 post-binary-point bits
  - Result: 8 pre-, 24 post-binary-point bits
- Subject to tight area constraints
  
  \[ a = a_r + j a_i \quad b = b_r + j b_i \]
  
  \[ p = ab = p_r + j p_i = (a_r b_r - a_i b_i) + j(a_i b_r + a_r b_i) \]

- 4 multiplies, 1 add, 1 subtract
  - Perform sequentially using 1 multiplier, 1 adder/subtractor
**Complex Multiplier Datapath**

![Datapath Diagram]

**Complex Multiplier in VHDL**

```vhdl
library ieee; use ieee.std_logic_1164.all, ieee.fixed_pkg.all;

entity multiplier is
  port ( clk, reset : in std_logic;
         input_rdy : in std_logic;
         a_r, a_i, b_r, b_i : in sfixed(3 downto -12);
         p_r, p_i : out sfixed(7 downto -24) );
end entity multiplier;

architecture rtl of multiplier is
  signal a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce : std_logic;  -- control signals
  signal a_operand, b_operand : sfixed(3 downto -12);
  signal pp, pp1, pp2, sum : sfixed(7 downto -24);
  ...
begin
```

Complex Multiplier in VHDL

```vhdl
a_operand <= a_r when a_sel = '0' else a_i;  -- mux
b_operand <= b_r when b_sel = '0' else b_i;  -- mux
pp <= a_operand * b_operand;  -- multiplier

pp1_reg : process (clk) is  -- partial product register 1
begin
  if rising_edge(clk) then
    if pp1_ce = '1' then
      pp1 <= pp;
    end if;
  end if;
end process pp1_reg;

pp2_reg : process (clk) is  -- partial product register 2
begin
  if rising_edge(clk) then
    if pp2_ce = '1' then
      pp2 <= pp;
    end if;
  end if;
end process pp2_reg;

sum <= pp1 + pp2 when sub = '0' else pp1 - pp2;  -- add/sub
p_r_reg : process (clk) is  -- result real-part register
begin
  if rising_edge(clk) then
    if p_r_ce = '1' then
      p_r <= sum;
    end if;
  end if;
end process p_r_reg;
p_i_reg : process (clk) is  -- result imag-part register
begin
  if rising_edge(clk) then
    if p_i_ce = '1' then
      p_i <= sum;
    end if;
  end if;
end process p_i_reg;

...  -- control circuit
end architecture rtl;
```

Multiplier Control Sequence

- Avoid resource conflict
- First attempt
  1. $a_r \times b_r \rightarrow pp1_{\text{reg}}$
  2. $a_i \times b_i \rightarrow pp2_{\text{reg}}$
  3. $pp1 - pp2 \rightarrow p_r_{\text{reg}}$
  4. $a_r \times b_i \rightarrow pp1_{\text{reg}}$
  5. $a_i \times b_r \rightarrow pp2_{\text{reg}}$
  6. $pp1 + pp2 \rightarrow p_i_{\text{reg}}$
- Takes 6 clock cycles

Multiplier Control Sequence

- Merge steps where no resource conflict
- Revised attempt
  1. $a_r \times b_r \rightarrow pp1_{\text{reg}}$
  2. $a_i \times b_i \rightarrow pp2_{\text{reg}}$
  3. $pp1 - pp2 \rightarrow p_r_{\text{reg}}$
  4. $a_i \times b_r \rightarrow pp2_{\text{reg}}$
  5. $pp1 + pp2 \rightarrow p_i_{\text{reg}}$
- Takes 5 clock cycles
### Multiplier Control Signals

<table>
<thead>
<tr>
<th>Step</th>
<th>a_sel</th>
<th>b_sel</th>
<th>pp1_ce</th>
<th>pp2_ce</th>
<th>sub</th>
<th>p_r_ce</th>
<th>p_i_ce</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>−</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>−</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>−</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>−</td>
<td>−</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Finite-State Machines

- Used the implement control sequencing
- Based on mathematical automaton theory
- A FSM is defined by
  - set of inputs: $\Sigma$
  - set of outputs: $\Gamma$
  - set of states: $S$
  - initial state: $s_0 \in S$
  - transition function: $\delta: S \times \Sigma \rightarrow S$
  - output function: $\omega: S \times \Sigma \rightarrow \Gamma$ or $\omega: S \rightarrow \Gamma$
**FSM in Hardware**

- **Mealy FSM**: \( \omega: S \times \Sigma \rightarrow \Gamma \)
- **Moore FSM**: \( \omega: S \rightarrow \Gamma \)

**FSM Example: Multiplier Control**

- **One state per step**
- **Separate idle state?**
  - Wait for \( \text{input\_rdy} = '1' \)
  - Then proceed to steps 1, 2, ...
  - But this wastes a cycle!
- **Use step 1 as idle state**
  - Repeat step 1 if \( \text{input\_rdy} \neq '1' \)
  - Proceed to step 2 otherwise
- **Output function**
  - Defined by table on slide 43
  - Moore or Mealy?

Transition function:

<table>
<thead>
<tr>
<th>current_state</th>
<th>input_rdy</th>
<th>next_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>step1</td>
<td>0</td>
<td>step1</td>
</tr>
<tr>
<td>step1</td>
<td>1</td>
<td>step2</td>
</tr>
<tr>
<td>step2</td>
<td>–</td>
<td>step3</td>
</tr>
<tr>
<td>step3</td>
<td>–</td>
<td>step4</td>
</tr>
<tr>
<td>step4</td>
<td>–</td>
<td>step5</td>
</tr>
<tr>
<td>step5</td>
<td>–</td>
<td>step1</td>
</tr>
</tbody>
</table>
State Encoding

- Encoded in binary
  - $N$ states: use at least $\lceil \log_2 N \rceil$ bits
- Encoded value used in circuits for transition and output function
  - encoding affects circuit complexity
- Optimal encoding is hard to find
  - CAD tools can do this well
- One-hot works well in FPGAs
- Often use 000...0 for idle state
  - reset state register to idle

FSMs in VHDL

- Use an enumeration type for state values
  - abstract, avoids specifying encoding

```vhdl
type multiplier_state is (step1, step2, step3, step4, step5);
signal current_state, next_state : multiplier_state;
```
Multiplier Control in VHDL

**state_reg** : process (clk, reset) is
begin
if reset = '1' then
    current_state <= step1;
elsif rising_edge(clk) then
    current_state <= next_state;
end if;
end process state_reg;

next_state_logic : process (current_state, input_rdy) is
begin
    case current_state is
    when step1 =>
        if input_rdy = '0' then
            next_state <= step1;
        else
            next_state <= step2;
        end if;
    when step2 =>
        next_state <= step3;
    when step3 =>
        next_state <= step4;
    when step4 =>
        next_state <= step5;
    when step5 =>
        next_state <= step1;
    end case;
end process next_state_logic;

output_logic : process (current_state) is
begin
    case current_state is
    when step1 =>
        a_sel <= '0'; b_sel <= '0'; pp1_ce <= '1'; pp2_ce <= '0';
        sub <= '0'; p_r_ce <= '0'; p_i_ce <= '0';
    when step2 =>
        a_sel <= '1'; b_sel <= '1'; pp1_ce <= '0'; pp2_ce <= '1';
        sub <= '0'; p_r_ce <= '0'; p_i_ce <= '0';
    when step3 =>
        a_sel <= '0'; b_sel <= '1'; pp1_ce <= '1'; pp2_ce <= '0';
        sub <= '1'; p_r_ce <= '1'; p_i_ce <= '0';
    when step4 =>
        a_sel <= '1'; b_sel <= '0'; pp1_ce <= '0'; pp2_ce <= '1';
        sub <= '0'; p_r_ce <= '0'; p_i_ce <= '0';
    when step5 =>
        a_sel <= '0'; b_sel <= '0'; pp1_ce <= '0'; pp2_ce <= '0';
        sub <= '0'; p_r_ce <= '0'; p_i_ce <= '1';
    end case;
end process output_logic;
State Transition Diagrams

- Bubbles to represent states
- Arcs to represent transitions

Example
- \( S = \{s_1, s_2, s_3\} \)
- Inputs \((a_1, a_2)\): \( \Sigma = \{(0,0), (0,1), (1,0), (1,1)\} \)
- \( \delta \) defined by diagram

Example
- \( x_1, x_2 \): Moore-style
- \( y_1, y_2, y_3 \): Mealy-style outputs
Multiplier Control Diagram

- Input: input_rdy

- Outputs
  - a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce

![Multiplier Control Diagram](image)

Bubble Diagrams or VHDL?

- Many CAD tools provide editors for bubble diagrams
  - Automatically generate VHDL for simulation and synthesis

- Diagrams are visually appealing
  - but can become unwieldy for complex FSMs

- Your choice...
  - or your manager's!
Register Transfer Level

- RTL — a level of abstraction
  - data stored in registers
  - transferred via circuits that operate on data

Clocked Synchronous Timing

- Registers driven by a common clock
  - Combinational circuits operate during clock cycles (between rising clock edges)

\[ t_{co} + t_{pd} + t_{su} < t_c \]
Control Path Timing

\[ t_{co} + t_{pd-s} + t_{pd-o} + t_{pd-c} + t_{su} < t_c \]

\[ t_{co} + t_{pd-s} + t_{pd-ns} + t_{su} < t_c \]

Ignore \( t_{pd-s} \) for a Moore FSM

Timing Constraints

- Inequalities must hold for all paths
- If \( t_{co} \) and \( t_{su} \) the same for all paths
  - Combinational delays make the difference
- Critical path
  - The combinational path between registers with the longest delay
  - Determines minimum clock period for the entire system
- Focus on it to improve performance
  - Reducing delay may make another path critical
Interpretation of Constraints

1. Clock period depends on delays
   - System can operate at any frequency up to a maximum
   - OK for systems where high performance is not the main requirement

2. Delays must fit within a target clock period
   - Optimize critical paths to reduce delays if necessary
   - May require revising RTL organization

Clock Skew

- Need to ensure clock edges arrive at all registers at the same time
- Use CAD tools to insert clock buffers and route clock signal paths
Off-Chip Connections

- Delays going off-chip and inter-chip
  - Input and output pad delays, wire delays
- Same timing rules apply
  - Use input and output registers to avoid adding external delay to critical path

Asynchronous Inputs

- External inputs can change at any time
  - Might violate setup/hold time constraints
  - Can induce metastable state in a flipflop
  - Unbounded time to recover
    - May violate setup/hold time of subsequent flipflop

\[ MTBF = \frac{e^{k_2 t}}{k_1 f_1 f_2} \quad \text{if} \quad k_2 \gg 0 \]
Synchronizers

- If input changes outside setup/hold window
  - Change is simply delayed by one cycle
- If input changes during setup/hold window
  - First flipflop has a whole cycle to resolve metastability
- See data sheets for metastability parameters

Switch Inputs and Debouncing

- Switches and push-buttons suffer from contact bounce
  - Takes up to 10ms to settle
  - Need to *debounce* to avoid false triggering
- Requires two inputs and two resistors
- Must use a break-before-make double-throw switch
Switch Inputs and Debouncing

- Alternative
  - Use a single-throw switch
  - Sample input at intervals longer than bounce time
  - Look for two successive samples with the same value

- Assumption
  - Extra circuitry inside the chip is cheaper than extra components and connections outside

Debouncing in VHDL

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

entity debouncer is
  port ( clk, reset : in std_logic;  -- clk frequency = 50MHz
         pb : in std_logic; pb_debounced : out std_logic );
end entity debouncer;

architecture rtl of debouncer is
  signal count50000 : integer range 0 to 49999;
  signal clk_100Hz : std_logic;
  signal pb_sampled : std_logic;
begin
```
Debouncing in VHDL

```vhdl
div_100Hz : process (clk, reset) is
begin
  if reset = '1' then
    clk_100Hz <= '0';
    count500000 <= 0;
  elsif rising_edge(clk) then
    if count500000 = 499999 then
      count500000 <= 0;
      clk_100Hz <= '1';
    else
      count500000 <= count500000 + 1;
      clk_100Hz <= '0';
    end if;
  end if;
end process div_100Hz;

debounce_pb : process (clk) is
begin
  if rising_edge(clk) then
    if clk_100Hz = '1' then
      if pb = pb_sampled then
        pb_debounced <= pb;
      end if;
    end if;
  end if;
end process debounce_pb;
end architecture rtl;
```

Verifying Sequential Circuits

- DUV may take multiple and varying number of cycles to produce output
- Checker needs to
  - synchronize with test generator
  - ensure DUV outputs occur when expected
  - ensure DUV outputs are correct
  - ensure no spurious outputs occur
Example: Multiplier Testbench

entity multiplier_testbench is
end entity multiplier_testbench;

library ieee; use ieee.std_logic_1164.all,
     ieee.fixed_pkg.all, ieee.math_complex.all;
architecture verify of multiplier_testbench is
constant t_c : time := 50 ns;
signal clk, reset : std_logic;
signal input_rdy : std_logic;
signal a_r, a_i, b_r, b_i : sfixed(3 downto -12);
signal p_r, p_i : sfixed(7 downto -24);
signal a, b : complex;
beginn
duv : entity work.multiplier(rtl)
  port map ( clk, reset, input_rdy,
            a_r, a_i, b_r, b_i,
            p_r, p_i );
clk_gen : process is
begin
  wait for t_c / 2; clk <= '1';
  wait for t_c / 2; clk <= '0';
extend process clk_gen;
reset <= '1', '0' after 2 * t_c ns;
apply_test_cases : process is
begin
  wait until falling_edge(clk) and reset = '0';
a <= cmplx(0.0, 0.0); b <= cmplx(1.0, 2.0); input_rdy <= '1';
wait until falling_edge(clk); input_rdy <= '0';
for i in 1 to 5 loop
  wait until falling_edge(clk);
end loop;
a <= cmplx(1.0, 1.0); b <= cmplx(1.0, 1.0); input_rdy <= '1';
wait until falling_edge(clk); input_rdy <= '0';
for i in 1 to 6 loop
  wait until falling_edge(clk);
end loop;
-- further test cases ...
wait;
extend process apply_test_cases;
a_r <= to_sfixed(a.re, a_r'left, a_r'right);
a_i <= to_sfixed(a.im, a_i'left, a_i'right);
b_r <= to_sfixed(b.re, b_r'left, b_r'right);
b_i <= to_sfixed(b.im, b_i'left, b_i'right);
**Example: Multiplier Testbench**

```vhdl
check_outputs : process is
    variable p : complex;
begin
    wait until rising_edge(clk) and input_rdy = '1';
    p := a * b;
    for i in 1 to 5 loop
        wait until falling_edge(clk);
    end loop;
    assert abs(to_real(p_r) - p.re) < 2.0**(-12)
        and abs(to_real(p_i) - p.im) < 2.0**(-12);
end process check_outputs;
end architecture verify;
```

**Asynchronous Timing**

- Clocked synchronous timing requires
  - global clock distribution with minimal skew
  - path delay between registers < clock period
- Hard to achieve in complex multi-GHz systems
- Globally async, local synch (GALS) systems
  - Divide the systems into local clock domains
  - Inter-domain signals treated as async inputs
  - Simplifies clock managements and constraints
  - Delays inter-domain communication
- Delay-insensitive asynchronous systems
  - no clock signals
Other Clock-Related Issues

- Inter-chip clocking
  - Distributing high-speed clocks on PCBs is hard
  - Often use slower off-chip clock, with on-chip clock a multiple of off-chip clock
  - Synchronize on-chip with phase-locked loop (PLL)
- In multi-PCB systems
  - treat off-PCB signals as asynch inputs
- Low power design
  - Continuous clocking wastes power
  - Clock gating: turn off clock to idle subsystems

Summary

- Registers for storing data
  - synchronous and asynchronous control
  - clock enable, reset, preset
- Latches: level-sensitive
  - usually unintentional in VHDL
- Counters
  - free-running dividers, terminal count, reset, load, up/down
Summary

- RTL organization of digital systems
  - datapath and control section
- Finite-State Machine (FSM)
  - states, inputs, transition/output functions
  - Moore and Mealy FSMs
  - bubble diagrams
- Clocked synch timing and constraints
  - critical path and optimization
- Asynch inputs, switch debouncing
- Verification of sequential systems