Modeling Complex Behavior

- Concurrent signal assignment statements can easily capture the gate level behavior of digital systems
- Higher level digital components have more complex behaviors
  - input/output behavior not easily captured by concurrent signal assignment statements
  - models utilize state information
  - incorporate data structures
- We need more powerful constructs

--

The Process Statement

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4 is
  port ( In0, In1, In2, In3 : in std_logic_vector (7 downto 0);
         Sel : in std_logic_vector(1 downto 0);
         Z : out std_logic_vector (7 downto 0));
end entity mux4;

architecture behavioral-3 of mux4 is
begin
  process (Sel, In0, In1, In2, In3) is
    variable Zout: std_logic;
  begin
    if (Sel = "00") then Zout := In0;
    elsif (Sel = "01") then Zout := In1;
    elsif (Sel = "10") then Zout := In2;
    else Zout:= In3;
    end if;
    Z <= Zout;
  end process;
end architecture behavioral;
```
The Process Construct

- Statements in a process are executed sequentially
- A process body is structured much like conventional C or Pascal programs
  - declaration and use of variables
  - if-then, if-then-else, case, for and while constructs
  - process can contain signal assignment statements
- A process executes concurrently with other concurrent signal assignment statements
- A process takes 0 simulation time to execute and may schedule events in the future
- We can think of a process as a complex signal assignment statement!

Concurrent Processes: Full Adder

- Each of the components of the full adder can be modeled using a process
- Processes execute concurrently
- Processes communicate via signals
Concurrent Processes: Full Adder

library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port (In1, c_in, In2: in std_logic;
      sum, c_out: out std_logic);
end entity full_adder;

architecture behavioral of full_adder is
signal s1, s2, s3: std_logic;
constant delay: Time := 5 ns;
begin
HA1: process (In1, In2) is
begin
s1 <= (In1 xor In2) after delay;
s3 <= (In1 and In2) after delay;
end process HA1;
end architecture behavioral;

Concurrent Processes: Half Adder

library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
port (a, b: in std_logic;
      sum, carry: out std_logic);
end entity half_adder;

architecture behavior of half_adder is
begin
sum_proc: process(a,b) is
begin
if (a = b) then
  sum <= '0' after 5 ns;
else
  sum <= (a or b) after 5 ns;
end if;
end process;
carry_proc: process (a,b) is
begin
  case a is
  when '0' =>
    carry <= a after 5 ns;
  when '1' =>
    carry <= b after 5 ns;
  when others =>
    carry <= 'X' after 5 ns;
  end case;
end process carry_proc;
end architecture behavior;
Processes + CSAs

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity memory is
port (address, write_data: in std_logic_vector (7 downto 0);
       MemWrite, MemRead, clk, reset: in std_logic;
       read_data: out std_logic_vector (7 downto 0));
end entity memory;

architecture behavioral of memory is
signal dmem0, dmem1, dmem2, dmem3: std_logic_vector (7 downto 0);
begin
mem_proc: process (clk) is
begin
-- process body
end process mem_proc;

-- read operation CSA
end architecture behavioral;

Process + CSAs

mem_proc: process (clk) is
begin
if (rising_edge(clk)) then -- wait until next clock edge
if reset = "1" then -- initialize values on reset
  dmem0 <= x"00"; -- memory locations are initialized to
  dmem1 <= x"11"; -- some random values
  dmem2 <= x"22";
  dmem3 <= x"33";
elsif MemWrite = '1' then -- if not reset then check for memory write
  case address (1 downto 0) is
  when "00" => dmem0 <= write_data;
  when "01" => dmem1 <= write_data;
  when "10" => dmem2 <= write_data;
  when "11" => dmem3 <= write_data;
  when others => dmem0 <= x"ff";
  end case;
end if;
end if;
end process read_proc;
Process + CSAs

- memory read is implemented with a conditional signal assignment

```
read_data <= dmem0 when address (1 downto 0) = "00" and MemRead = '1' else
            dmem1 when address (1 downto 0) = "01" and MemRead = '1' else
            dmem2 when address (1 downto 0) = "10" and MemRead = '1' else
            dmem3 when address (1 downto 0) = "11" and MemRead = '1' else
            x"00";
```

- processes appear as CSA statements (or vice versa)
- capture the same external behavior as CSA statements
- processes describe more complex event generation behavior

Process Behavior

- All processes executed once at start-up
- Thereafter dataflows determine process initiation
- Use of signals vs. variables
  - view processes as components with an interface/function

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

entity sig_var is
  port (x, y, z: in std_logic;
        res1, res2: out std_logic);
end entity sig_var;

architecture behavior of sig_var is
begin
  proc1: process (x, y, z) is -- Process 1
  variable var_s1, var_s2: std_logic;
  begin
    L1: var_s1 := x and y;
    L2: var_s2 := var_s1 xor z;
    L3: res1 <= var_s1 nand var_s2;
    end process;

  proc2: process (x, y, z) -- Process 2
  begin
    L1: sig_s1 <= x and y;
    L2: sig_s2 <= sig_s1 xor z;
    L3: res2 <= sig_s1 nand sig_s2;
    end process;

end architecture behavior;
```
Variables vs. Signals: Example

proc1: process (x, y, z) is -- Process 1
variable var_s1, var_s2: std_logic;
begin
L1: var_s1 := x and y;
L2: var_s2 := var_s1 xor z;
L3: res1 <= var_s1 and var_s2;
end process;

proc2: process (x, y, z) -- Process 2
begin
L1: sig_s1 <= x and y;
L2: sig_s2 <= sig_s1 xor z;
L3: res2 <= sig_s1 and sig_s2;
end process;

• Distinction is between computing values vs. time/value pairs
The Wait Statement

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
port (D, Clk : in std_logic;
    Q, Qbar : out std_logic);
end entity dff;
architecture behavioral of dff is
begin
output: process is
begin
    wait until (Clk'event and Clk = '1'); -- wait for rising edge
    Q <= D after 5 ns;
    Qbar <= not D after 5 ns;
end process output;
end architecture behavioral;
```

- The wait statements can describe synchronous or asynchronous timing operations

Forms of the Wait Statement

- wait for `<time expression>`
  - example: wait for gate_delay, wait for 20 ns
- wait on `<signal>`
  - wait on R, S, clk
- , wait until `<boolean expression>`
  - wait until (rising_edge(clk)
- Both sensitivity lists and wait statements cannot be used!
The Wait Statement: Waveform Generation

library IEEE;
use IEEE.std_logic_1164.all;
entity two_phase is
port(phi1, phi2, reset: out std_logic);
end entity two_phase;

architecture behavioral of two_phase is
begin
clock_process: process is
begin
phi1 <= '1', '0' after 10 ns;
phi2 <= '0', '1' after 12 ns, '0' after 18 ns;
wait for 20 ns;
end process clock_process;
end architecture behavioral;

Wait Statement: Asynchronous Inputs

library IEEE; use IEEE.std_logic_1164.all;
entity asynch_dff is
port (R, S, D, Clk: in std_logic;
    Q, Qbar: out std_logic);
end entity asynch_dff;
architecture behavioral of asynch_dff is
begin
output: process (R, S, Clk) is
begin
if (R = '0') then
    Q <= '0' after 5 ns;
    Qbar <= '1' after 5 ns;
elsif S = '0' then
    Q <= '1' after 5 ns;
    Qbar <= '0' after 5 ns;
elsif (rising_edge(Clk)) then
    Q<= D after 5 ns;
    Qbar <= (not D) after 5 ns;
end if;
end process output;
end architecture behavioral;
The Wait Statement

- A process can have multiple wait statements
- A process cannot have both a wait statement and a sensitivity list (it should have one or the other): why?
- Explicit control over suspension and resumption of processes
- Representation of synchronous and asynchronous events in a digital systems

Attributes

- Data can be obtained about VHDL items such as types, arrays and signals.
  
  object’attribute

- Example: consider the implementation of a signal

  ![Diagram of a signal driver with value-time pairs]

  - what types of information about this signal are useful?
    - occurrence of an event
    - elapsed time since last event
    - previous value, i.e., prior to the last event
Classes of Attributes

- Value attributes
  - returns a constant value
- Function attributes
  - invokes a function that returns a value
- Signal attributes
  - creates a new signal
- Type attributes
  - returns a type
- Range attributes
  - returns a range

Value Attributes

- Return a constant value
  
  type statetype is (state0, state1, state2 state3);
  - state_type’left = state0
  - state_type’right = state3

- Examples

<table>
<thead>
<tr>
<th>Value attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_name’left</td>
<td>returns the left most value of type_name in its defined range</td>
</tr>
<tr>
<td>type_name’right</td>
<td>returns the right most value of type_name in its defined range</td>
</tr>
<tr>
<td>type_name’high</td>
<td>returns the highest value of type_name in its range</td>
</tr>
<tr>
<td>type_name’low</td>
<td>returns the lowest value of type_name in its range</td>
</tr>
<tr>
<td>array_name’length</td>
<td>returns the number of elements in the array array_name</td>
</tr>
</tbody>
</table>
Function Attributes

• Use of attributes invokes a function call which returns a value
  if (Clk’event and Clk = ‘1’)

  function call

• Examples: function signal attributes

<table>
<thead>
<tr>
<th>Function attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal_name’event</td>
<td>Return a Boolean value signifying a change in value on this signal</td>
</tr>
<tr>
<td>signal_name’active</td>
<td>Return a Boolean value signifying an assignment made to this signal. This assignment may not be a new value.</td>
</tr>
<tr>
<td>signal_name’last_event</td>
<td>Return the time since the last event on this signal</td>
</tr>
<tr>
<td>signal_name’last_active</td>
<td>Return the time since the signal was last active</td>
</tr>
<tr>
<td>signal_name’last_value</td>
<td>Return the previous value of this signal</td>
</tr>
</tbody>
</table>

• Function array attributes

<table>
<thead>
<tr>
<th>Function attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_name’left</td>
<td>returns the left bound of the index range</td>
</tr>
<tr>
<td>array_name’right</td>
<td>returns the right bound of the index range</td>
</tr>
<tr>
<td>array_name’high</td>
<td>returns the upper bound of the index range</td>
</tr>
<tr>
<td>array_name’low</td>
<td>returns the lower bound of the index range</td>
</tr>
</tbody>
</table>

  type mem_array is array(0 to 7) of bit_vector(31 downto 0)
  • mem_array’left = 0
  • mem_array’right = 7
  • mem_array’length = 8 (value kind attribute)
Range Attributes

- Returns the index range of a constrained array
  
  ```
  for i in value_array`range loop
  ...
  my_var := value_array(i);
  ...
  end loop;
  ```

- Makes it easy to write loops

Signal Attributes

- Creates a new “implicit” signal

<table>
<thead>
<tr>
<th>Signal attribute</th>
<th>Implicit Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal_name`delayed(T)</td>
<td>Signal delayed by T units of time</td>
</tr>
<tr>
<td>signal_name`transaction</td>
<td>Signal whose value toggles when signal_name is active</td>
</tr>
<tr>
<td>signal_name`quiet(T)</td>
<td>True when signal_name has been quiet for T units of time</td>
</tr>
<tr>
<td>signal_name`stable(T)</td>
<td>True when event has not occurred on signal_name for T units of time</td>
</tr>
</tbody>
</table>

- Internal signals are useful modeling tools

  ```
  wait on ReceiveData`transaction
  if ReceiveData`delayed = ReceiveData then
  ...
  ```
Example

```vhdl
clk_process: process
begin
wait until (clk'event and clk = '1');
if reset = '1' then
state <= statetype'left;
else state <= next_state;
end if;
end process clk_process;
```

- The signal state is an enumerated type

```vhdl
type statetype is (state0, state1, state3, state4);
signal state:statetype':= statetype'left;
```

State Machines

- Basic components
  - combinational component: output function and next state function
  - sequential component
- Natural process implementation
Example: State Machine

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
extenity state_machine is
port(reset, clk, x : in std_logic;
     z : out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
type statetype is (state0, state1);
signal state, next_state : statetype := state0;
begin
comb_process: process(state, x) is
begin
--- process description here
end process comb_process;

clk_process: process is
begin
--- process description here
end process clk_process;
end architectural behavioral;
```

Example: Output and Next State Functions

```vhdl
comb_process: process(state, x) is
begin
   case state is
   when state0 =>
      -- set output signals and next state
      if x = '0' then
         next_state <= state1;
         z <= '1';
      else
         next_state <= state0;
         z <= '0';
      end if;
   when state1 =>
      if x = '1' then
         next_state <= state0;
         z <= '0';
      else
         next_state <= state1;
         z <= '1';
      end if;
   end case;
end process comb_process;
```

- Combination of the next state and output functions
Example: Clock Process

``` VHDL 
clk_process: process is 
begin 
  wait until (clk'event and clk = '1'); -- wait until the rising edge
  if reset = '1' then -- check for reset and initialize state
    state <= statetype'left;
  else
    state <= next_state;
  end if;
end process clk_process;
end behavioral;
```

- Use of asynchronous reset to initialize into a known state

Summary

- Processes
  - variables and sequential statements
  - `if-then`, `if-then-else`, `case`, `while`, `for`
  - concurrent processes
  - sensitivity list
- The Wait statement
  - wait until, wait for, wait on
- Attributes
- Modeling State machines