Lecture 6
Behavioral Modeling: Processes

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030
Higher Levels of Abstraction

• Conditional CSA’s allow us to move from the gate level to the register transfer level (registers, multiplexers, adders etc.)
• Impractical for higher levels of abstraction where we want to focus on high level function rather than structural implementation.
• For example:
  – 16-bit microprocessor ??
    • or maybe something simpler like a
  – 9-stage, 16 bit FIR filter ??
FIR Filter

\[
\begin{align*}
\text{din} & \quad \rightarrow \quad \text{reg0} \quad \rightarrow \quad \text{reg1} \quad \rightarrow \quad \text{reg2} \quad \rightarrow \quad \text{reg3} \quad \rightarrow \quad \text{reg4} \quad \rightarrow \quad \text{reg5} \quad \rightarrow \quad \text{reg6} \quad \rightarrow \quad \text{reg7} \\
& \quad \uparrow \text{clk} \quad \downarrow \text{clk} \quad \downarrow \text{clk} \quad \downarrow \text{clk} \quad \downarrow \text{clk} \quad \downarrow \text{clk} \quad \downarrow \text{clk} \quad \downarrow \text{clk} \\
X & \quad \rightarrow \quad X \quad \rightarrow \quad X \quad \rightarrow \quad X \quad \rightarrow \quad X \quad \rightarrow \quad X \quad \rightarrow \quad X \quad \rightarrow \quad X \quad \rightarrow \quad X \\
\text{a0} & \quad \rightarrow \quad \text{a1} \quad \rightarrow \quad \text{a2} \quad \rightarrow \quad \text{a3} \quad \rightarrow \quad \text{a4} \quad \rightarrow \quad \text{a5} \quad \rightarrow \quad \text{a6} \quad \rightarrow \quad \text{a7} \quad \rightarrow \quad \text{a8} \\
& \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\Sigma & \quad \rightarrow \quad \text{dout} \quad 16
\end{align*}
\]
Process Construct

• Sequentially executed block of code
  – much like conventional programming languages
  – executes in zero time
• Supports **variables** as well as **signals**
• Powerful control flow constructs
• More control over when assignments are executed
entity NANDXOR is
  port (
    A, B : in std_logic;
    C : in std_logic;
    D : out std_logic);
end NANDXOR;

architecture RTL of NANDXOR is
  signal T : std_logic;
begin
  p0 : T <= A nand B after 2 ns;
  p1 : process (T, C)
  begin
    D <= T xor C after 3 ns;
  end process p1;
end RTL;
p1 : process \((T, C)\)

- \((T,C)\) is the process sensitivity list of process p1
- A process is executed whenever an event occurs on any signal in the sensitivity list
- Statements in the process are executed sequentially
- Process is then suspended until an event occurs on one of signals in process sensitivity list
Concurrent Signal Assignment or Process?

- These two representations are equivalent!
- CSA’s are implemented as processes
- A CSA is a short-hand method of defining a process that schedules events on only one output signal
- Each process can be thought of as a concurrent assignment that can:
  – do complex sequential processing to calculate a result
  – schedule events on more than one signal
• An if statement selects a sequence of statements for execution based on the value of a condition (Boolean value).

```
if boolean-expression then
  sequential-statements
{elsif boolean-expression then
  sequential-statements}
[else
  sequential-statement]
end if;
```

• Only first clause (boolean expression) found to be true is executed – order of these clauses matters
• Note that elsif is one word but end if is two words
entity cmp_8 is
    port (  
a, b: in std_logic_vector (7 downto 0);
en: in std_logic;
eq: out std_logic);
end cmp_8;
architecture behavior of cmp_8 is
begin
    cmp_proc : process (a, b, en)
    begin
        if en='0' then
            eq <= '0' after 4 ns;
        elsif a=b then
            eq <= '1' after 7 ns;
        else
            eq <= '0' after 7 ns;
        end if;
    end process cmp_proc;
end behavior;
- Rising edge triggered sequential circuit
- D flip-flop captures value of D when clk goes from ‘0’ to ‘1’
- S and R are asynchronous over-riding set and reset
- In order to model, we need to know on which input an event has occurred
### Sidebar: Attributes

- Attributes return information about a signal, e.g.:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal_name’event</td>
<td>returns the Boolean value $True$ if an event on the signal occurred at current time, otherwise returns value $False$</td>
</tr>
<tr>
<td>signal_name’active</td>
<td>returns the Boolean value $True$ if there has been a transaction (assignment) on the signal at the current time, otherwise returns the value $False$</td>
</tr>
<tr>
<td>signal_name’last_event</td>
<td>returns the time elapsed since the last event on the signal</td>
</tr>
<tr>
<td>signal_name’last_active</td>
<td>returns the time elapsed since the last transaction on the signal</td>
</tr>
<tr>
<td>signal_name’last_value</td>
<td>returns the value of the signal before the last event occurred on the signal</td>
</tr>
<tr>
<td>signal_name’delayed(T)</td>
<td>returns a signal that is the delayed version (by time $T$) of the original one. [T is optional, default $\Delta$]</td>
</tr>
<tr>
<td>signal_name’stable(T)</td>
<td>returns the Boolean value $True$ if no event has occurred on the signal during the interval $T$, otherwise returns $False$. [T is optional, default $\Delta$]</td>
</tr>
<tr>
<td>signal_name’quiet(T)</td>
<td>returns the Boolean value $True$ if no transaction has occurred on the signal during the interval $T$, otherwise returns $False$. [T is optional, default $\Delta$]</td>
</tr>
</tbody>
</table>
dx <= ‘0’ after 0ns, ‘1’ after 5ns, ‘0’ after 10ns, ‘0’ after 15ns, ‘1’ after 20ns, ‘0’ after 30ns;

- `dx’event` has the value TRUE at t=10ns
- `dx’event` has the value FALSE at t=15ns
- `dx’active` has the value TRUE at t=15ns
- `dx’last_event` has the value 5ns at t=15ns
- `dx’last_value` has the value ‘1’ at t=15ns
- `dx’delayed(8ns)` has the value ‘1’ at t=15ns
- `dx’stable(8ns)` has the value FALSE at t=15ns
- `dx’stable(2ns)` has the value TRUE at t=15ns
- `dx’delayed(8ns)’event` has the value TRUE only at times 13, 18, 28, and 38ns.
Example: D Flip-Flop

entity Dff is
    port (
        clk,D,Rb,Sb: in std_logic;
        Q,Qb: out std_logic);
end entity Dff;

architecture DA1 of Dff is
begin
    ff_proc: process (clk,Rb,Sb)
    begin
        if Rb='0' then
            Q<='0' after 5 ns;
            Qb<='1' after 5 ns;
        elsif Sb='0' then
            Q<='1' after 5 ns;
            Qb<='0' after 5 ns;
        elsif clk'event and clk='1' then
            Q<=D after 7 ns;
            Qb<= not D after 7 ns;
        end if;
    end process ff_proc;
end architecture DA1;
A case statement selects one of several branches for execution based on the value of expression:

```
case expression is
  when choices => sequential-statements
  when choices => sequential-statements
  -- can have any number of branches
  [when others => sequential-statements]
end case;
```

The set of `choices` must be mutually exclusive and cover all possible values of the `expression`.

- Because choices must be mutually exclusive, order of clauses does not matter.
Example: 4-way 8-bit multiplexer

entity mux4 is
  port ( IN0, IN1, IN2, IN3: in std_logic_vector (7 downto 0);
     sel: in std_logic_vector (1 downto 0);
     en: in std_logic;
     Z: out std_logic_vector (7 downto 0));
end entity mux4;

architecture using_case of mux4 is
begin
  P1: process (IN0,IN1,IN2,IN3,sel,en)
  begin
    if en='0' then
      Z<= x"00" after 5ns;
    else
      case sel is
        when "00" => Z<= IN0 after 5ns;
        when "01" => Z<= IN1 after 5ns;
        when "10" => Z<= IN2 after 5ns;
        when "11" => Z<= IN3 after 5ns;
        when others => Z<="XXXXXXXX" after 5ns;
      end case;
    end if;
  end process;
end architecture using_case;
Sidebar: Bit String Literals

- Special forms of string literals that are used to represent binary, octal, or hexadecimal numeric data values. The numerical value is given in double quotes (") and the representation is specified by a character preceding the quoted value.
- The underscore character can be used for convenience and clarity - it does not change the represented value. For example:
  - Binary data: `B"0110_1101_1111_0010"
  - Octal data: `O"16_67_62"`
  - Hexadecimal data: `X"6DF2"
  - Binary data: `"00101101111010"

- Note that binary is assumed when no base specified, but underscore cannot be used in this case.
Null Statement

- The null statement is used to explicitly say “do nothing”

Null;

- Particularly useful in case statement when no action is required in response to one of the choices. For example:

```vhDL
case sel is
  when "00" => Z <= IN0 after 5ns;
  when "01" => Z <= IN1 after 5ns;
  when "10" => Z <= IN2 after 5ns;
  when "11" => Null;
end case;
```
The loop statement is used to iterate through a set of sequential statements.

```
[loop-label : ] iteration-scheme loop
    sequential-statements
end loop [loop-label];
```

Three types of iteration scheme:

1. **for** `identifier` **in** `range`
2. **while** `boolean-expression`
3. No iteration scheme specified
Example: 8-stage, 16-bit register pipeline

entity pipe8 is
  port ( din: in std_logic_vector(15 downto 0);
        clk: in std_logic;
        dout: out std_logic_vector(15 downto 0));
end entity pipe8;

architecture pipe_be of pipe8 is
  type sig8x16 is array (0 to 7) of std_logic_vector(15 downto 0);
  signal regfile: sig8x16;
begin
  rproc: process (clk) is
  begin
    if clk='1' then
      regfile(0)<=din after 5ns;
      for i in 1 to 7 loop
        regfile(i)<=regfile(i-1) after 5ns;
      end loop;
    end if;
  end process;
  dout<=regfile(7);
end architecture pipe_be;
Loop Statement: For Iteration

[loop-label : ] for index in range loop
  sequential-statements
end loop [loop-label];

- index is implicitly declared in the loop statement
- index is local to the loop and read-only

- Loop statement is most powerful when used with variables
  (next lecture)
Model a single-bit ALU that has four functions: ZERO, AND, OR and ADD selected by a two-bit input opcode. Use two processes: one to describe the basic operations and one to describe the multiplexer.
Model a register file with sixteen registers, where each register is 32 bits. Implement the register file with two processes: one process reads the register file, while another writes the register file.

The register file has 32-bit din and dout ports and a 4-bit address port. The read operation should be asynchronous. The write operation should be synchronous, occurring on the rising edge of a write signal.

An enable signal allows expansion of the register file to a larger address range. When the enable signal is 0, dout is tri-state (high impedance) and write operations are inhibited.