Simulation #2: Bus Operation

- This lecture will discuss some topics that will be useful for performing Simulation #2
  - VHDL Variables
  - Modeling of Finite State Machines
  - Modeling of In/Out Ports
  - Bus Operation
- Also see the link to Simulation #2 on the WWW page

VHDL Variables

- VHDL variables come in two varieties: global and local
- Local variables can only be declared in processes, procedures, and functions

  ```
  process (clk, r)
  variable a: integer := 0;
  begin
    a := a + 1;
  end;
  ```
- Variable assignments use the " := " operator
- Variable assignments take place immediately (unlike signal assignments which only place entries on the time queue; the time queue entries are not processed until the process is suspended).

VHDL Variables (continued)

- Local variables are not visible outside of a process
- For a process, the value of a variable is static, i.e., it retains its value between process invocations
- For a procedure or function, the value of the variable is re-initialized each time the procedure or function is called
- A global variable is declared outside of a process using the 'shared' keyword. Will discuss global variables in more detail later.

```
architecture a of myentity is
  shared variable a: integer := 0;
process (clk, r)
begin
  a := a + 1;
  ....
end;
```

Finite State Machines

A Mealy-type FSM is shown below. In a Mealy-type FSM, the outputs are a function of both the present state and the current inputs.

One way to model this to use a separate processes for the state registers and combinational logic.

Entity

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- MEALY FSM machine
-- pstate A=0 A=1
-- entries are nextstate/outputZ

entity mealy_fsm is
  port (a, clk: in std_logic;
        z:      out std_logic);
end mealy_fsm;
```

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

-- mealy fsm in
entity mealy_fsm is
  port (a, clk: in std_logic;
        z: out std_logic);
end;
```

```
architecture mealy_fsm of mealy FSM is
begin
  process (a, clk)
  begin
    if rising_edge(clk) then
      case a is
      when '0' =>
        z <= 'Z';
      when '1' =>
        z <= not z;
      end case;
    end if;
  end process;
end;
```
Keep Track of #Clock Cycles in State ST0: Try #1

```
begin
  -- assign default output values
  z <= '0';
  n_state <= p_state;
  when p_state is
  when st0 => if (A = '1') then n_state <= st3; end if;
  st0_count <= st0_count +1;
  when st1 => if (A = '1') then n_state <= st0; end if;
  when st3 => if (A = '0') then n_state <= st2; end if;
  when st2 => if (A = '1') then n_state <= st1; end if;
  when st1 => if (A = '1') then n_state <= st0; end if;
  when st0 => if (A = '1') then n_state <= st3; end if;
end case;
end process comb_part;
Wrong: st0_count incremented anytime 'A' changes while in state ST0
```

Count # of state transitions from ST0 to ST3

Try #1 (some code deleted for brevity) - put counter in ST0

```
begin
  -- assign default output values
  z <= '0';
  n_state <= p_state;
  when p_state is
  when st0 => if (A = '1') then z <= '1'; n_state <= st3; end if;
  st0_count <= st0_count +1;
  when st1 => if (A = '1') then z <= '1'; n_state <= st3; end if;
  when st3 => if (A = '0') then n_state <= st2; end if;
  when st2 => if (A = '1') then n_state <= st1; end if;
  when st1 => if (A = '1') then n_state <= st0; end if;
  when st0 => if (A = '1') then n_state <= st3; end if;
end case;
end process comb_part;
Note sensitivity list signals
```

Will not work if A goes from '0' to '1' and back to '0' while in state ST0. Will not transition to ST3, yet counter is incremented.
Count # of state transitions from ST0 to ST3

Try again... put counter in State ST3

when st3 =>
    counter := counter + 1;

Wrong, will increment anytime we are in ST3 and process triggers
- not that a change in 'A' can cause the process to trigger.

when st3 =>
    if (p_state'event) then
        counter := counter + 1;
    end if;

This will work because counter only incremented on change to p_state
and only way to get to ST3 is from ST0.

Count # of state transitions from ST0 to ST3

What if ASM chart has more than one way to get to ST3?
(i.e., the ASM chart included a transition from ST2 to ST3,
but only wanted to count transitions from ST0 to ST3).

when st3 =>
    if (p_state'event and (p_state'last_value = st0)) then
        counter := counter + 1;
    end if;

The attribute 'last_value gives the previous value of the signal
- can use it here to see if previous state was ST0.

Port Types

- VHDL port types can be in, out, inout, buffer
  - in intended for input-only ports. Cannot assign values to ports of type in.
  - out intended for output-only port. Cannot read the value of ports of type out.
  - inout intended for bidirectional ports
  - buffer type is like an out port but can be read from

- Do not use buffer ports. Problems are:
  - a buffer port can only have one driver on it
  - a buffer port must be connected to another port of type buffer,
    which means buffer ports propagate through hierarchy

- If you need to read a value from an out port, use the 'driving_value' attribute
  - Will return the driving value of the port, can be used to read the driving value of a port of type out.

Modeling a bidirectional port

- A bidirectional port means that sometimes your model is providing the drive,
  and that sometimes another component is providing the drive for the signal
- Use an inout type for the port type
  - y: inout std_logic;

- Internal to your model, keep track of when your model is supposed to be providing the drive.
  I usually declare two internal signals
  - y_out -- internal signal that model will drive
  - dir -- 'direction' signal that tells me if the model is supposed to be driving or not

- If you need to read a value from an inout port, use the 'driving_value' attribute
  - Will return the driving value of the port, can be used to read the driving value of a port of type out.

Sim #2 bbusy signal

'Z' value is important
- this lets external component override this signal

bbusy is a 'bus busy' signal. When a CPU owns the bus, it should drive the bbusy line to '0'. When a CPU does not own the bus, it should leave the line released (driving a 'Z'). When a CPU is granted the bus, it must watch the bbusy line to see if the bus is free (value of 'H'). If the bus is free and the CPU has been granted the bus, then the CPU can claim the bus by driving bbusy low. Note that bbusy is a bidirectional signal!