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)

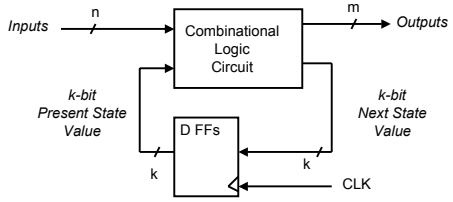
- Local variables are not visible outside of a process
- For a process, the value of a variable is static, i.e., it retains its value between process invocations
- For a procedure or function, the value of the variable is re-initialized each time the procedure or function is called
- A global variable 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;
```

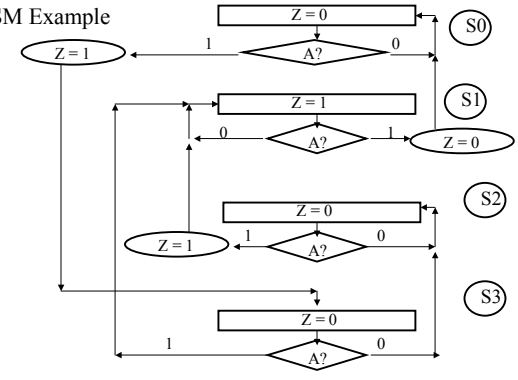

Finite State Machines

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.



FSM Example



Entity

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


-- MEALY FSM machine
-- pstate     A=0           A=1
----------------------------------------
-- ST0    |   ST0 / 0       ST3/1   == entries are nextstate/outputZ
-- ST1    |   ST1 / 1       ST0/0   == entries are nextstate/outputZ
-- ST2    |   ST2 / 0       ST1/1   == entries are nextstate/outputZ
-- ST3    |   ST2 / 0       ST1/0   == entries are nextstate/outputZ
--
entity mealy_fsm is
 port (
       a, clk: in std_logic;
       z:      out std_logic
);
end mealy_fsm;
```

Architecture

```
architecture behv of mealy_fsm is
  -- enumerated type for state definitions
  type mystate  is (ST0,ST1,ST2,ST3);
  signal p_state, n_state : mystate;

  begin

    seq_part: process(clk)
      begin
          if (clk = '0' and clk'event) then
             p_state <= n_state;
          end if;
      end process seq_part;

     -- note that this process triggers only on changes to any of its inputs,
     -- namely 'p_state' and 'a'
     comb_part: process (p_state,a)
       begin
           -- assign default output values
           z <= '0';             -- default is output
           n_state <= p_state;  -- stay in same state by default
           case p_state is
             when st0 => if (A = '1') then z <= '1';  n_state <= st3;  end if;
             when st1 => if (A = '1') then n_state <= st0;
                         else  z <= '1';   end if;
             when st2 => if (A = '1') then z <= '1'; n_state <= st1; end if;
             when st3 => if (A = '0') then   n_state <= st2;
                         else  n_state <= st1; end if;
           end case;
       end process comb_part;
end behv;
```

falling edge state register

Keep Track of #Clock Cycles in State ST0: Try #1

```
comb_part: process (p_state,a)
variable st0_count: integer := 0;

begin
 -- assign default output values
 z <= '0';             -- default is output
 n_state <= p_state;  -- stay in same state by default
 case 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 n_state <= st0;
              else  z <= '1';   end if;
  when st2 => if (A = '1') then z <= '1'; n_state <= st1; end if;
  when st3 => if (A = '0') then   n_state <= st2;
              else  n_state <= st1; end if;
 end case;
end process comb_part;
```

Wrong: st0_count incremented anytime 'A' changes while in state ST0

Keep Track of #Clock Cycles in State ST0: Try #2

```
comb_part: process (p_state,a)
variable st0_count: integer := 0;

begin
 -- assign default output values
 z <= '0';             -- default is output
 n_state <= p_state;  -- stay in same state by default
 case p_state is
  when st0 => if (A = '1') then z <= '1';  n_state <= st3;  end if;
              if (pstate'event) then
                st0_count := st0_count +1 ;
              end if;
  when st1 => if (A = '1') then n_state <= st0;
              else  z <= '1';   end if;
    .......
```

Wrong: st0_count incremented anytime we enter ST0 from a different state (pstate changes) but does not count clock cycles that A=0 and we are in state ST0.

Keep Track of #Clock Cycles in State ST0: Try #3

```
comb_part: process (p_state,a,clk)
variable st0_count: integer := 0;

begin
 -- assign default output values
 z <= '0';             -- default is output
 n_state <= p_state;  -- stay in same state by default
 case p_state is
  when st0 => if (A = '1') then z <= '1';  n_state <= st3;  end if;
              if (clk'event and clk = '1') then
                st0_count := st0_count +1 ;
              end if;
  when st1 => if (A = '1') then n_state <= st0;
              else  z <= '1';   end if;
    .......
```

added clock

rising edge check

Will work; *clk* must be added to sensitivity list, and the check occurring on rising edge will cause the variable to be updated in the middle of the clock cycle. If you check the falling edge of *clk*, then the variable is updated at the end of the clock cycle. The process is executed twice every clock cycle (once for each change in *clk*).

Keep Track of #Clock Cycles in State ST0: Try #4

```
comb_part: process (p_state'transaction,a)
variable st0_count: integer := 0;

begin
 -- assign default output values
 z <= '0';             -- default is output
 n_state <= p_state;  -- stay in same state by default
 case p_state is
  when st0 => if (A = '1') then z <= '1';  n_state <= st3;  end if;
              if (pstate'active) then
                st0_count := st0_count +1 ;
              end if;
  when st1 => if (A = '1') then n_state <= st0;
              else  z <= '1';   end if;
    .......
```

Will work; *'active* returns true anytime a transaction (an assignment) is made to *pstate*, which happens every clock cycle via the *seq_part* process. Added *pstate'transaction* to sensitivity list because *'transaction* returns a signal that toggles for each transaction. This process executes only once per clock cycle, is more efficient than previous solution.

Count # of state transitions from ST0 to ST3

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

Note sensitivity list signals

```
comb_part: process (p_state,a)
 -- 'counter' will count transitions from 'st0' to 'st3'
 variable  counter: integer := 0;

begin
 -- assign default output values
 z <= '0';             -- default is output
 n_state <= p_state;  -- stay in same state by default
 case p_state is
  when st0 => if (A = '1') then z <= '1';  n_state <= st3;
              counter := counter + 1;
             end if;
 when st1 => if (A = '1') then n_state <= st0;
             else  z <= '1';   end if;
    .......
```

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 =>
 -- the signal attribute 'last_value returns the last value of the
 -- signal; it is useful here.
  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

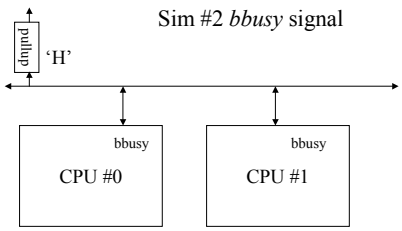
- A bidirectional port means that sometimes your model is providing the drive, and that sometimes another component is providing the drive for the signal
- Use an *inout* type for the port type
  - `y: inout std_logic ;`
- Internal to your model, keep track of when your model is supposed to be providing 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

Modeling a *bidirectional* port

'Z' value is important – this lets external component overdrive this signal

```
entity example is
 port (clk: std_logic;
       y: inout std_logic));
end example;

architecture a of example is
 signal dir : std_logic := '0';  -- '0' reading, '1' driving
 signal y_out : std_logic := 'Z';
begin
  y <= y_out when (dir = '1') else 'Z';

  .....

  process (clk)
  .... some process
      when STATE_A:   dir <= '1';  y_out <= '1';   -- drive some value
 ......
      when STATE_K:  dir <= '0';  --- let some other model drive this signal

 ........
      when STATE_N:   if (y = '1') then....  -- can read an 'inout' port
                                             -- will read the resolved value!!
```


Sim #2 *bbusy* 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 have 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!