PART 8

BEHAVIORAL_DESCRIPTIONS

- Constructs for sequential descriptions
- Process statement is a key construct
- Assertion for behavioral checks
- Handshaking constructs
- Timing control
- Formatted I/O
- Several MSI parts will be modeled
- Will present a complete MSI based design

In General:

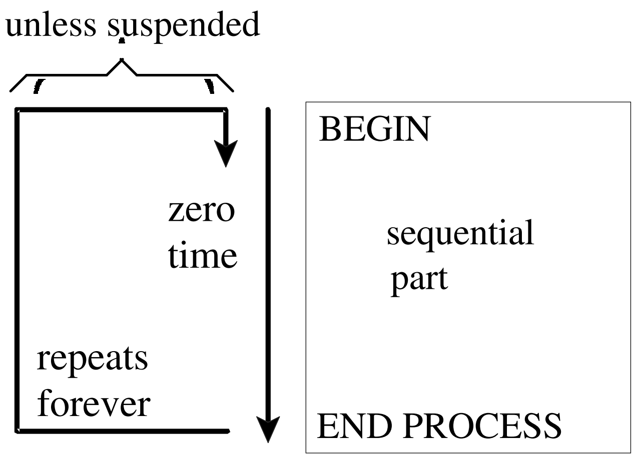
- Constructs for I/O mapping
- Without hardware datails

Behavioral.process

```
PROCESS
  . . .
  declarative_part
  . . .
BEGIN
  . . .
  sequential_part
  . . .
END PROCESS
```

- PROCESS:
- A concurrent statement, enclosing sequential statements
- Declarative part contains only variables and constants
- Use only sequential constructs

Behavioral.process



- Unless sequential part is suspended
- It executes in zero real and delta time
- It repeats itself forever

Behavioral.process

```
ARCHITECTURE sequentiality_demo OF partial_process IS
BEGIN
   PROCESS
   BEGIN
     . . .
     x  <= a;
     y <= b;
     . . .
   END PROCESS;
END sequentiality_demo;
```

- First: *a* is scheduled for *x*
- Next: *b* is scheduled for *y*
- *x* and *y* receive values at the same time
- Both assignments occur a delta later
- Zero time between both scheduling

Behavioral.process

```
ARCHITECTURE execution_time_demo OF partial_process IS
BEGIN
   PROCESS
   BEGIN
     . . .
     x  <= a AFTER 10 NS;
     y <= b AFTER 6 NS;
     . . .
   END PROCESS;
END execution_time_demo;
```

- First: *a* is scheduled for *x*
- Next: *b* is scheduled for *y*
- *y* receives *b* sooner than *x* receiving *a*

Behavioral.process

```
ARCHITECTURE data_availability_demo OF partial_process IS
   SIGNAL x_sig : BIT := '0';
BEGIN
   PROCESS
   BEGIN
     . . .
     x_sig <= '1';
     IF x_sig = '1' THEN
       perform  action_1
     ELSE
       perform  action_2
     END IF;
     . . .
   END PROCESS;
END data_availability_demo;
```

- Assume *x_sig* is initially '0'
- Assignment of '1' to *x_sig* takes a delta
- Action_2 will be taken
- Variable *x_var* had to be declared inside the Process statement
- If *x_var* was used instead of *x_sig*, action_1 would be taken

Behavioral.process.sensitivity

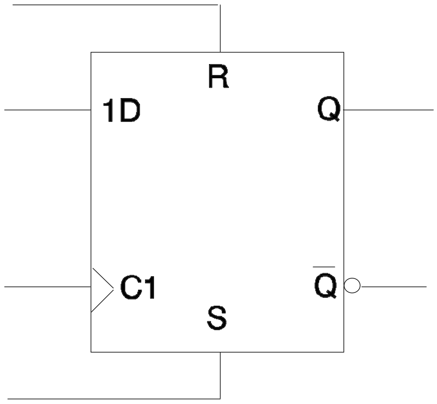
```
ARCHITECTURE ...
BEGIN
...

a <= b;
...
...
c <= d;
...
END ...
```

```
ARCHITECTURE ...
BEGIN
        ...
PROCESS (b)
        a <= b;
END PROCESS;
...
c <= d;
...
END ...
```

- Process is a concurrent statement
- Signal assignment is a concurrent statement
- Signal assignment is also a sequential statement
- **On the left:** events on *b* cause assignment
- Process is executed when an event occurs on *b*
- **On the right:** *(b)* is sensitivity list of process
- Process statement executes only once for every event on *b*
- Process suspends till next event on *b*

Behavioral.process.sensitivity




```
ENTITY d_sr_flipflop IS
  GENERIC (sq_delay, rq_delay, cq_delay : TIME := 6 NS);
  PORT (d, set, rst, clk : IN BIT; q, qb : OUT BIT);
END d_sr_flipflop;
--
ARCHITECTURE behavioral OF d_sr_flipflop IS
  SIGNAL state : BIT := '0';
BEGIN
  dff: PROCESS (rst, set, clk)
  BEGIN
    IF set = '1' THEN          state <= '1' AFTER sq_delay;
    ELSIF rst = '1' THEN   state <= '0' AFTER rq_delay;
    ELSIF clk = '1' AND clk'EVENT THEN    state <= d AFTER cq_delay;
    END IF;
  END PROCESS dff;
  q <= state;     qb <= NOT state;
END behavioral;
```

- Three concurrent processes
- *dff* process is sensitive to *(rst, set, clk)*
- Internal *state* receives proper value
- Events on *state* cause events on *q* and *qb* after δ

Behavioral.process.sensitivity

```
ARCHITECTURE average_delay_behavioral OF d_sr_flipflop IS
BEGIN
  dff: PROCESS (rst, set, clk)
    VARIABLE state : BIT := '0';
  BEGIN
    IF set = '1' THEN
      state := '1';
    ELSIF rst = '1' THEN
      state := '0';
    ELSIF clk = '1' AND clk'EVENT THEN
      state := d;
    END IF;
    q <= state AFTER (sq_delay + rq_delay + cq_delay)/3;
    qb <= NOT state AFTER (sq_delay + rq_delay + cq_delay)/3;
  END PROCESS dff;
END average_delay_behavioral;
```

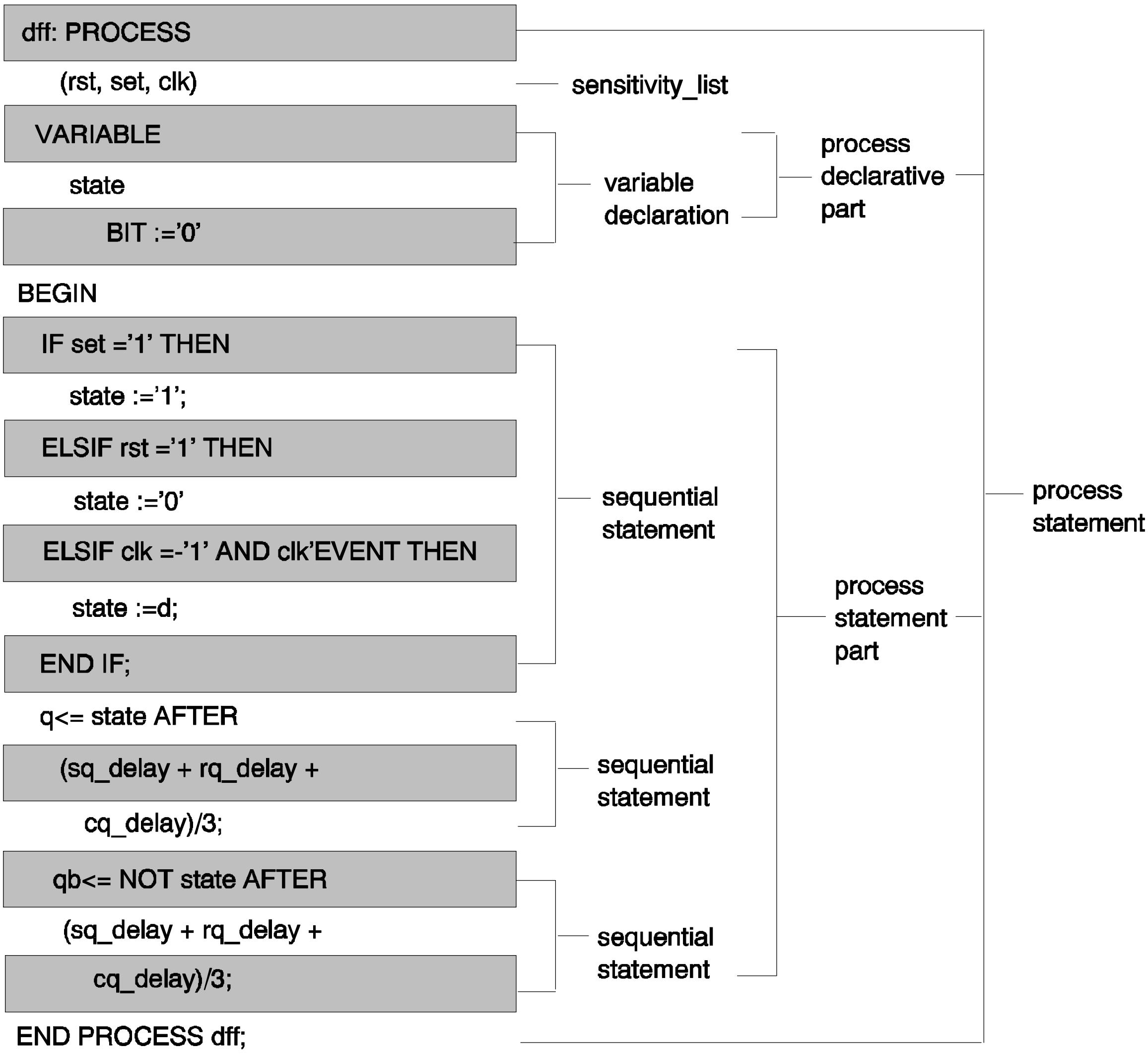
- Single process assigns values to *q* and *qb*
- This description eliminates the δ delay of the last description
- Less precise timing

Behavioral.process.sensitivity

TIME (NS)	SIGNALS SS	RR	CC	DD	Q1	Q2	QB1	QB2
0	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
+*d*	...	...	...	...	...	...	'1'	...
6	...	...	...	...	...	...	...	'1'
200	'1'	...	...	...	...	...	...	...
206	...	...	...	...	...	'1'	...	'0'
+*d*	...	...	...	...	'1'	...	'0'	...
500	...	...	'1'	...	...	...	...	...
1000	...	...	'0'	...	...	...	...	...
1200	'0'	...	...	...	...	...	...	...
1400	...	'1'	...	...	...	...	...	...
1406	...	...	...	...	...	'0'	...	'1'
+*d*	...	...	...	...	'0'	...	'1'	...
1500	...	...	'1'	...	...	...	...	...
2000	...	...	'0'	...	...	...	...	...
2200	...	'0'	...	...	...	...	...	...
2400	...	...	...	'1'	...	...	...	...
2500	...	...	'1'	...	...	...	...	...
2506	...	...	...	...	...	'1'	...	'0'
+*d*	...	...	...	...	'1'	...	'0'	...
3000	...	...	'0'	...	...	...	...	...
3300	...	...	...	'0'	...	...	...	...
3500	...	...	'1'	...	...	...	...	...
3506	...	...	...	...	...	'0'	...	'1'
+*d*	...	...	...	...	'0'	...	'1'	...
4000	...	...	'0'	...	...	...	...	...

- Simulation run compares flip-flop descriptions
- The *3 process* description has a δ delay
- However, potential of more precise timing

Behavioral.process



- Syntax details include sensitivity list

Behavioral.flow_constructs

```
long_running : LOOP
   . . .
   IF x = 25 THEN EXIT;
   END IF;
   . . .
END LOOP long_running;
```

```
NEXT loop_label WHEN condition;
```

```
EXIT WHEN x = 25;
```

- Loop is a sequential statement
- Example runs forever unless exited
- EXIT & NEXT control flow of loops
- EXIT & NEXT can be conditioned

Behavioral.flow_constructs

```
loop_1 : FOR i IN 5 TO 25 LOOP
   sequential_statement_1;
   sequential_statement_2;
   loop_2 : WHILE j <= 90 LOOP
       sequential_statement_3;
       sequential_statement_4;
       NEXT loop_1 WHEN condition_1;
       sequential_statement_5;
       sequential_statement_6;
   END LOOP loop_2;
   sequential_statement_7;
   sequential_statement_8;
END LOOP loop_1;
```

- FOR, WHILE are controlled forms of loop
- Can still use NEXT and EXIT
- The above NEXT statement causes looping to continue with statements 1

Behavioral.assertion

```
ASSERT assertion_condition REPORT
      "reporting_message" SEVERITY
      severity_level;
```

Make sure that assertion_condition *is true*
Otherwise report "reporting message" *then*
Take the severity_level *action*

- Use assert to flag violations
- Use assert to report events
- Can be sequential or concurrent
- Severity: FAILURE ERROR WARNING NOTE

Behavioral.assertion

```
BEGIN
  dff: PROCESS (rst, set, clk)
  BEGIN
    ASSERT
      (NOT (set = '1' AND rst = '1'))
    REPORT
      "set and rst are both 1"
    SEVERITY NOTE;
    IF set = '1' THEN
      state <= '1' AFTER sq_delay;
    ELSIF rst = '1' THEN
      state <= '0' AFTER rq_delay;
    ELSIF clk = '1' AND clk'EVENT THEN
      state <= d AFTER cq_delay;
    END IF;
  END PROCESS dff;
  q <= state;
  qb <= NOT state;
END behavioral;
```

- Conditions are checked only when process is activated
- Make sure that set='1' AND rst='1' does not happen
- Severity NOTE issues message

Behavioral.assertion

ASSERT

Good Conditions

REPORT

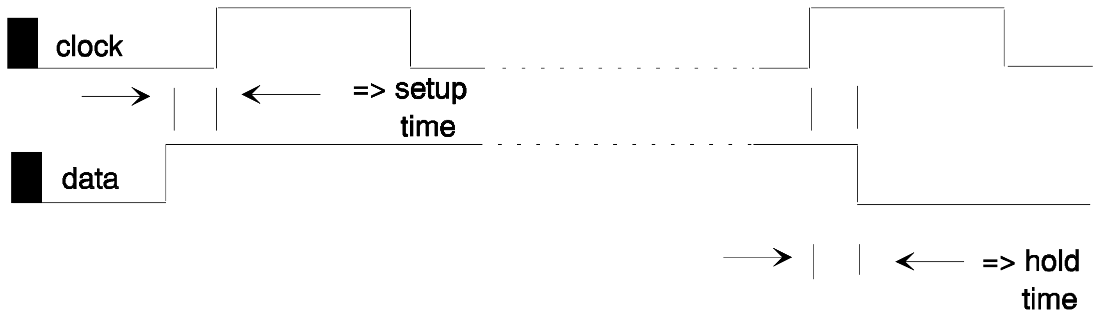
Violation of good condition

SEVERITY

Level;

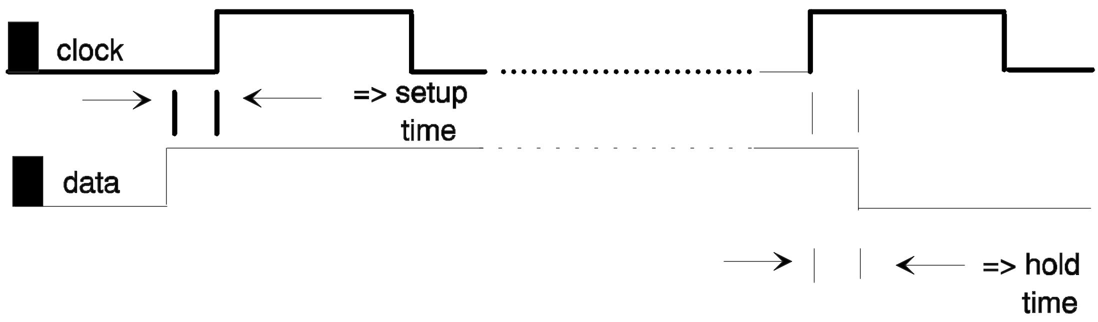
- Good conditions may be too many to list
- Good conditions = NOT (Bad conditions)
- Easier to use NOT of unwanted cases

Behavioral.assert



- Use assert to check setup and hold
- ASSERT set_up_violation check REPORT...
- ASSERT hold_violation check REPORT...

Behavioral.assert



Setup check in English:

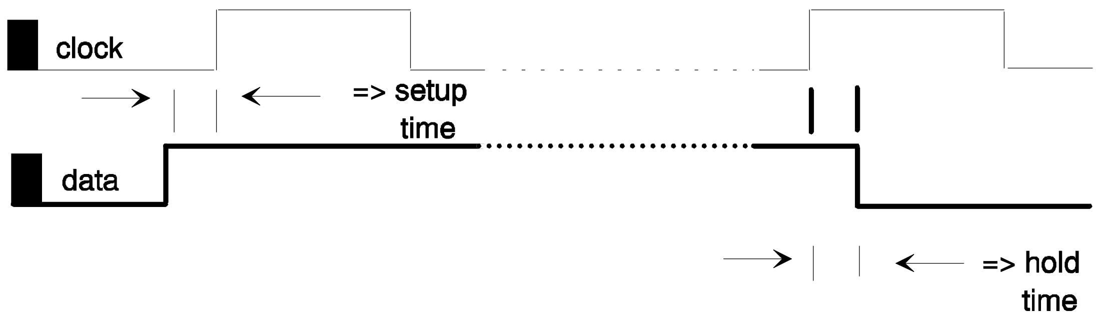
When *(clock changes from zero to 1)*, if *(data input has not been stable at least for the amount of the setup time),* then a setup time violation has occurred.

Setup check in VHDL:

```
(clock='1' AND NOT clock'STABLE)
AND
(NOT data'STABLE (setup_time)
```

- When the clock changes, check for stable data
- Check is placed after **clock** changes

Behavioral.assert



Hold check in English:

When *(there is a change on the data input)* if *(logic value on the clock is '1')* and *(clock has got a new value more recent than the amount of hold time)* then a hold time violation has occurred.

Hold check in VHDL:

```
(data'EVENT)
AND
(clock='1')
AND
(NOT clock'STABLE (hold_time))
```

- When data changes while clock is '1', check for stable clock
- Check is placed after **data** changes

Behavioral.assert

```
ENTITY d_sr_flipflop IS
  GENERIC (sq_delay, rq_delay, cq_delay : TIME := 6 NS;
        set_up, hold : TIME := 4 NS);
  PORT (d, set, rst, clk : IN BIT; q, qb : OUT BIT);
BEGIN
  ASSERT (NOT (clk = '1' AND clk'EVENT AND NOT d'STABLE(set_up) ))
    REPORT   "Set_up time violation"
    SEVERITY WARNING;
  ASSERT   (NOT (d'EVENT AND clk = '1' AND NOT clk'STABLE(hold) ))
    REPORT    "Hold time violation"
    SEVERITY WARNING;
END d_sr_flipflop;
--
ARCHITECTURE behavioral OF d_sr_flipflop IS
  SIGNAL state : BIT := '0';
BEGIN
  dff: PROCESS (rst, set, clk)
  BEGIN
    ASSERT   (NOT (set = '1' AND rst = '1'))
      REPORT   "set and rst are both 1"
      SEVERITY NOTE;
    IF set = '1' THEN    state <= '1' AFTER sq_delay;
    ELSIF rst = '1' THEN    state <= '0' AFTER rq_delay;
    ELSIF clk = '1' AND clk'EVENT THEN    state <= d AFTER cq_delay;
    END IF;
  END PROCESS dff;
  q <= state;   qb <= NOT state;
END behavioral;
```

- Concurrent assertion statements

Behavioral.wait

```
WAIT FOR waiting_time;

WAIT ON waiting_sensitivity_list;

WAIT UNTIL waiting_condition;

WAIT;
```

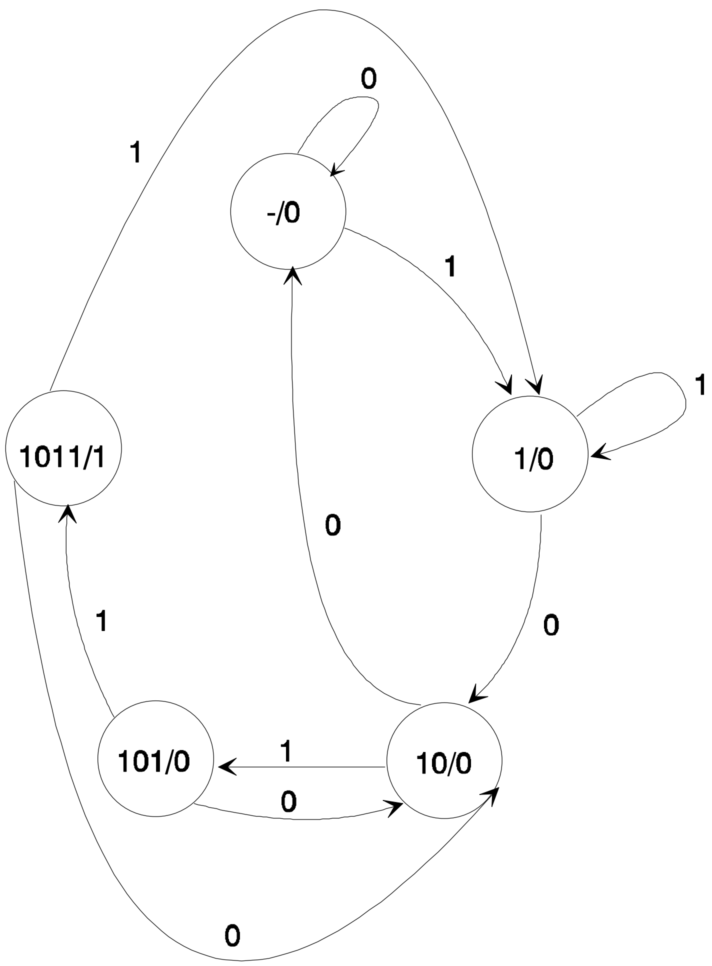
- Sequential statements
- Used for handshaking and delay modeling
- WAIT FOR real_time;
- WAIT FOR; --"a long time"
- WAIT ON (event on a signal);
- WAIT UNTIL event makes condition true;
- WAIT; --"forever"

Behavioral.wait

PROCESS	PROCESS **(a, b, c)**
BEGIN	BEGIN
.	.
.	.
.	.
WAIT ON (a, b, c);	.
END PROCESS;	END PROCESS;

- WAIT ON at the end is equivalent to using sensitivity list
- Cannot use WAIT in a process with sensitivity list
- WAIT suspends a Process

Behavioral.wait.state_machine



- A Moore 1011 detector
- Can use WAIT in a Process statement

Behavioral.wait.state_machine

```
ENTITY moore_detector IS
  PORT (x, clk : IN BIT; z : OUT BIT);
END moore_detector;
--
ARCHITECTURE behavioral_state_machine OF moore_detector IS
  TYPE state IS (reset, got1, got10, got101, got1011);
  SIGNAL current : state := reset;
BEGIN
  PROCESS
  BEGIN
    CASE current IS
      WHEN reset =>
        WAIT UNTIL clk = '1';
        IF x = '1' THEN current <= got1;  ELSE current <= reset;
        END IF;
      WHEN got1 =>
        WAIT UNTIL clk = '1';
        IF x = '0' THEN current <= got10;  ELSE current <= got1;
        END IF;
      WHEN got10 =>    WAIT UNTIL clk = '1';
        IF x = '1' THEN current <= got101;   ELSE current <= reset;
        END IF;
      WHEN got101 =>  WAIT UNTIL clk = '1';
        IF x = '1' THEN current <= got1011;  ELSE current <= got10;
        END IF;
      WHEN got1011 =>   z <= '1';
        WAIT UNTIL clk = '1';
        IF x = '1' THEN current <= got1;   ELSE current <= got10;
        END IF;
    END CASE;
    WAIT FOR 1 NS;   z <= '0';
  END PROCESS;
END behavioral_state_machine;
```

- Each choice corresponds to a state

Behavioral.wait.state_machine

```
ARCHITECTURE behavioral_state_machine OF moore_detector IS
  TYPE state IS (reset, got1, got10, got101, got1011);
  SIGNAL current : state := reset;
BEGIN
  PROCESS
  BEGIN
    CASE current IS
        . . .
        WHEN got1 =>
        WAIT UNTIL clk = '1';
        IF x = '0' THEN current <= got10;
        ELSE current <= got1;
        END IF;
        . . .
    END CASE;
    WAIT FOR 1 NS;
    z <= '0';
  END PROCESS;
END behavioral_state_machine;
```

- WAIT for rising edge of clk
- Assign new state to *current*
- Wait for transaction on *current*
- Repeat forever
- Can use *WAIT ON current 'TRANSACTION* instead
- Timing check flexibility in each state

Behavioral.wait.state_machine

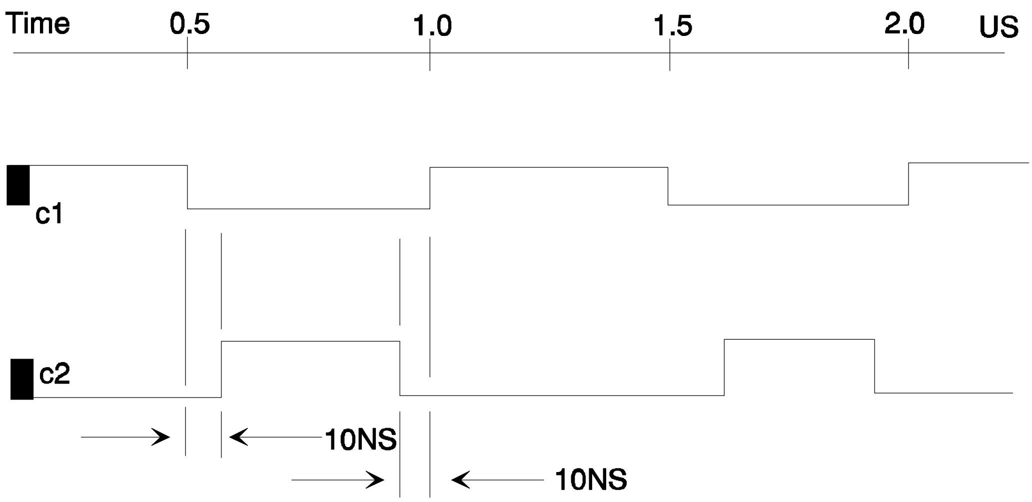
```
ENTITY moore_detector IS
  PORT (x, clk : IN BIT; z : OUT BIT);
END moore_detector;
--
ARCHITECTURE most_behavioral_state_machine OF moore_detector IS
  TYPE state IS (reset, got1, got10, got101, got1011);
  SIGNAL current : state := reset;
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk = '1' THEN
      CASE current IS
      WHEN reset =>
        IF x = '1' THEN current <= got1;
        ELSE current <= reset;  END IF;
      WHEN got1 =>
        IF x = '0' THEN current <= got10;
        ELSE current <= got1;  END IF;
      WHEN got10 =>
        IF x = '1' THEN current <= got101;
        ELSE current <= reset;  END IF;
      WHEN got101 =>
        IF x = '1' THEN current <= got1011;
        ELSE current <= got10;  END IF;
      WHEN got1011 =>
        IF x = '1' THEN current <= got1;
        ELSE current <= got10;  END IF;
      END CASE;
    END IF;
  END PROCESS;
  z <= '1' WHEN current = got1011 ELSE '0';
END most_behavioral_state_machine;
```

- Most behavioral state machine
- Least flexible, single active state

© Zainalabedin Navabi, 1994

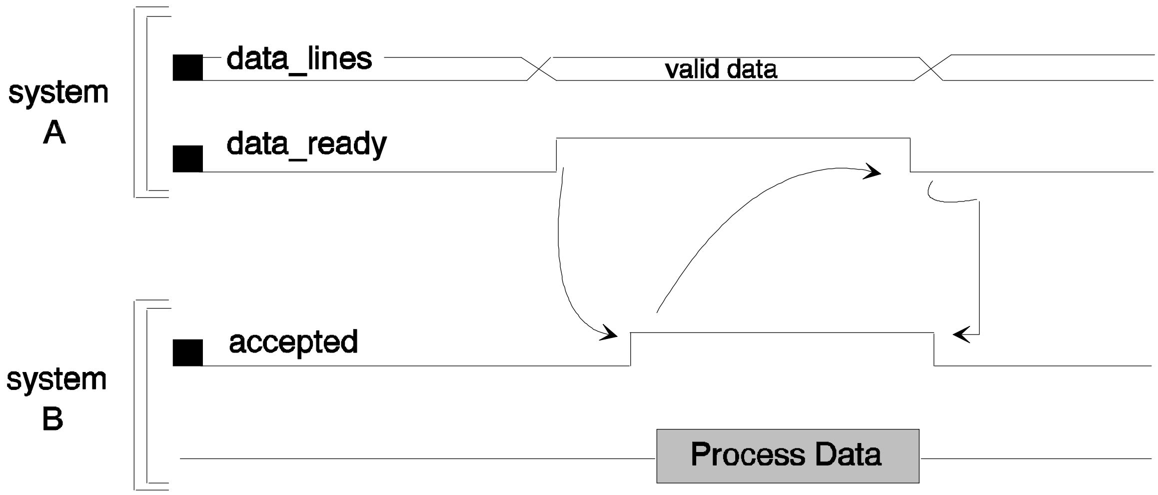
Behavioral.wait.clocking

```
. . .
phase2: PROCESS
BEGIN
    WAIT UNTIL c1 = '0';
    WAIT FOR 10 NS;
    c2 <= '1';
    WAIT FOR 480 NS;
    c2 <= '0';
END PROCESS phase2;
. . .
```



- Use WAIT for two phase clocking

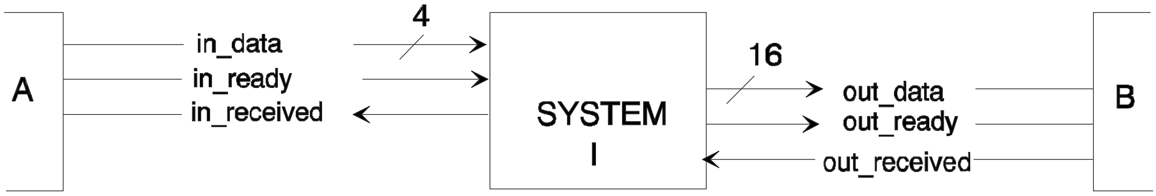
Behavioral.wait.handshaking



	-- start the following when ready to send data
	data_lines <= newly_prepared_data;
System	data_ready <= '1';
A	WAIT UNTIL accepted = '1';
	data_ready <= '0';
	-- can use data_lines for other purposes
	-- start the following when ready to accept data
	WAIT UNTIL data_ready = '1';
System	accepted <= '1';
B	-- start processing the newly received data
	WAIT UNTIL data_ready - '0';
	accepted <= '0';

- Systems *A* & *B* talk
- *A* prepares data, *B* accepts data
- *B* releases *A* when data is picked

Behavioral.wait.handshaking



- Use handshaking mechanism in an interface
- *A* prepares 4 bit data, *B* needs 16 bit data
- Create interface system *I*
- Talk to *A* to get data, talk to *B* to put data

Behavioral.wait.handshaking

```
ENTITY system_i IS
  PORT (in_data : IN BIT_VECTOR (3 DOWNTO 0);
    out_data : OUT BIT_VECTOR (15 DOWNTO 0);
    in_ready, out_received : IN BIT;  in_received, out_ready : OUT BIT);
END system_i;
--
ARCHITECTURE waiting OF system_i IS
  SIGNAL buffer_full, buffer_picked : BIT := '0';
  SIGNAL word_buffer : BIT_VECTOR (15 DOWNTO 0);
BEGIN
  a_talk: PROCESS
  BEGIN
    . . .
    -- Talk to A, collect 4 4-bit data, keep a count
    -- When ready, pass 16-bit data to b_talk
    . . .
  END PROCESS a_talk;

  b_talk: PROCESS
  BEGIN
    . . .
    -- Wait for 16-bit data from a_talk
    -- When data is received, send to B using proper handshaking
    . . .
  END PROCESS b_talk;
END waiting;
```

- *a_talk* process & *b_talk* process also talk to each other
- Use *buffer_full*, *buffer_picked*, and *word_buffer* for *a_talk* and *b_talk* communication

Behavioral.wait.handshaking

```
a_talk: PROCESS
  VARIABLE count : INTEGER RANGE 0 TO 4 := 0;
BEGIN
  WAIT UNTIL in_ready = '1';
  count := count + 1;
  CASE count IS
    WHEN 0 => NULL;
    WHEN 1 => word_buffer (03 DOWNTO 00) <= in_data;
    WHEN 2 => word_buffer (07 DOWNTO 04) <= in_data;
    WHEN 3 => word_buffer (11 DOWNTO 08) <= in_data;
    WHEN 4 => word_buffer (15 DOWNTO 12) <= in_data;
              buffer_full <= '1';    WAIT UNTIL buffer_picked = '1';
              buffer_full <= '0';    count := 0;
  END CASE;
  in_received <= '1';
  WAIT UNTIL in_ready = '0';
  in_received <= '0';
END PROCESS a_talk;

b_talk: PROCESS
BEGIN
  IF buffer_full = '0' THEN WAIT UNTIL buffer_full = '1'; END IF;
  out_data <= word_buffer;
  buffer_picked <= '1';
  WAIT UNTIL buffer_full = '0';
  buffer_picked <= '0';
  out_ready <= '1';
  WAIT UNTIL out_received = '1';
  out_ready <= '0';
END PROCESS b_talk;
```

- *a_talk* gets data from *A* and talks to *b_talk*
- *b_talk* talks to *a_talk* and sends data to *B*

Behavioral.ASCII_IO

```
USE STD.TEXTIO.ALL;

...
READLINE (f, l)
READ (l, v1),    READ (l, v2), ...

...
WRITE (l, v1),    WRITE (l, v2), ...
WRITELINE (f, l)

END FILE (f)
```

- READ or WRITE:
  BIT, BIT_VECTOR, BOOLEAN,
  CHARACTER, INTEGER, REAL,
  STRING, TME
- *l* is LINE, *f* is FILE

Behavioral.ASCII_IO

```
TYPE state IS (reset, got1, got10, got101);
TYPE state_vector IS ARRAY (NATURAL RANGE <>) OF state;
. . .
FUNCTION one_of (sources : state_vector) RETURN state IS
  USE STD.TEXTIO.ALL;
  VARIABLE l : LINE;
  FILE flush : TEXT IS OUT "/dev/tty";
  VARIABLE state_string : STRING (1 TO 7);
BEGIN
  FOR i IN sources'RANGE LOOP
    CASE sources(i) IS
      WHEN reset  => state_string := "reset  ";
      WHEN got1   => state_string := "got1   ";
      WHEN got10  => state_string := "got10  ";
      WHEN got101 => state_string := "got101 ";
    END CASE;
    WRITE (l, state_string, LEFT, 7);
  END LOOP;
  WRITELINE (flush, l);
  RETURN sources (sources'LEFT);
END one_of;
```

- Add screen output to resolution function
- Print after enumeration to string type conversion
- Keep appending to *l*, write all sources

Behavioral.ASCII_IO

```
USE STD.TEXTIO.ALL;
  PROCEDURE display (SIGNAL value1, value2 : BIT) IS
    FILE flush : TEXT IS OUT "/dev/tty;
    VARIABLE filler : STRING (1 TO 3) := "  ..";
    VARIABLE l : LINE;
  BEGIN
    WRITE (l, NOW, RIGHT, 8, NS);
    IF value1'EVENT THEN
      WRITE (l, value1, RIGHT, 3);
      WRITE (l, filler, LEFT, 0);
    ELSE
      WRITE (l, filler, LEFT, 0);
      WRITE (l, value2, RIGHT, 3);
    END IF;
    WRITELINE (flush, l);
  END display;
```

- An EVENT on *value1* or *value2* puts the following in *l*:

  NOW
- An EVENT on *value1* puts the following in *l*:

  v1 ...
- An EVENT on *value2* puts the following in *l*:

  ... v2
- WRITELINE writes:

  time v1 ...
  time ... v2

Behavioral.ASCII_IO

```
ENTITY two_phase_clock IS END two_phase_clock;
--
ARCHITECTURE input_output OF two_phase_clock IS
  -- The above display procedure goes here
  SIGNAL c1 : BIT := '1';
  SIGNAL c2 : BIT := '0';
BEGIN
  phase1: c1 <= NOT c1 AFTER 500 NS WHEN NOW < 4 US ELSE c1;
  phase2: PROCESS
  BEGIN
    WAIT UNTIL c1 = '0';
    WAIT FOR 10 NS;
    c2 <= '1';
    WAIT FOR 480 NS;
    c2 <= '0';
  END PROCESS phase2;
  display (c1, c2);
END input_output;
```

- Using display to display two clock phases
- FILE is reinitialized each time
- Because file is reinitialized, can only output to screen

Behavioral.ASCII_IO

```
USE STD.TEXTIO.ALL;
ENTITY two_phase_clock IS  END two_phase_clock;
--
ARCHITECTURE input_output OF two_phase_clock IS
  SIGNAL c1 : BIT := '1';    SIGNAL c2 : BIT := '0';
BEGIN
  . . .
    -- Generate c1 and c2 phases of the clock
  . . .
  writing: PROCESS (c1, c2)
    FILE flush : TEXT IS OUT "clock.out";
    VARIABLE filler : STRING (1 TO 3) := "  . .";
    VARIABLE l : LINE;
  BEGIN
    WRITE (l, NOW, RIGHT, 8, NS);
    IF c1'EVENT THEN
      WRITE (l, c1, RIGHT, 3);
      WRITE (l, filler, LEFT, 0);
    ELSE
      WRITE (l, filler, LEFT, 0);
      WRITE (l, c2, RIGHT, 3);
    END IF;
    WRITELINE (flush, l);
  END PROCESS writing;
END input_output;
```

- Declare FILE in process to keep it open during IO
- Same file (*clock.out*) will be preserved

Behavioral.ASCII_IO

```
   0 ns ..  0
 500 ns  0 ..
 510 ns ..  1
 990 ns ..  0
1000 ns  1 ..
1500 ns  0 ..
1510 ns ..  1
1990 ns ..  0
2000 ns  1 ..
2500 ns  0 ..
2510 ns ..  1
2990 ns ..  0
3000 ns  1 ..
3500 ns  0 ..
3510 ns ..  1
3990 ns ..  0
4000 ns  1 ..
```

- Output produced in the *clock.out* file
- Same output is also produced by display on the screen

Behavioral.ASCII_IO

```
ARCHITECTURE input_output OF two_phase_clock IS
  . . .
  SIGNAL print_tick : BIT := '0';
  CONSTANT print_resolution : TIME := 5 NS;
BEGIN
  -- Generate c1 and c2 phases of the clock as before
  print_tick <=  NOT print_tick AFTER print_resolution WHEN NOW <= 2 US
                        ELSE print_tick;
  plotting: PROCESS (print_tick, c1, c2)
    FILE flush : TEXT IS OUT "clock4.out";
    VARIABLE header : STRING (1 TO 18) :=  "          c1    c2   ";
    VARIABLE l : LINE;
    PROCEDURE append_wave_slice (SIGNAL s : BIT) IS
      VARIABLE lo_value : STRING (1 TO 3) :=  "|  ";
      VARIABLE hi_value : STRING (1 TO 3) :=  "  |";
      VARIABLE lo_to_hi : STRING (1 TO 3) :=  ".-+";
      VARIABLE hi_to_lo : STRING (1 TO 3) :=  "+-.";
    BEGIN
      IF s'LAST_EVENT < print_resolution AND s'LAST_VALUE /= s THEN
        IF s = '1' THEN    WRITE (l, lo_to_hi, RIGHT, 5);  -- Event occurs
        ELSE    WRITE (l, hi_to_lo, RIGHT, 5);  END IF;
      ELSE
        IF s = '1' THEN    WRITE (l, hi_value, RIGHT, 5);  -- No event
        ELSE    WRITE (l, lo_value, RIGHT, 5);  END IF;
      END IF;
    END append_wave_slice;
  BEGIN
    IF NOW = 0 US THEN   WRITE (l, header, LEFT, 0);   WRITELINE (flush,
     l);
    END IF;
    WRITE (l, NOW, RIGHT, 8, NS);
    append_wave_slice (c1);  append_wave_slice (c2);
    WRITELINE (flush, l);
  END PROCESS plotting;
END input_output;
```

- Waveform generation

Behavioral.ASCII_IO

```
           c1   c2
        ...
   480 ns     |    |
   485 ns     |    |
   490 ns     |    |
   495 ns     |    |
   500 ns  +-.     |
   505 ns  |       |
   510 ns  |       |
   510 ns  |      .-+
   515 ns  |          |
   520 ns  |          |
   525 ns  |          |
```

- Plotting is activated every 5 NS
- Write " |" for '1'; "| " for '0'
- Write "+-." for 1 to 0
- Write ".-+" for 0 to 1

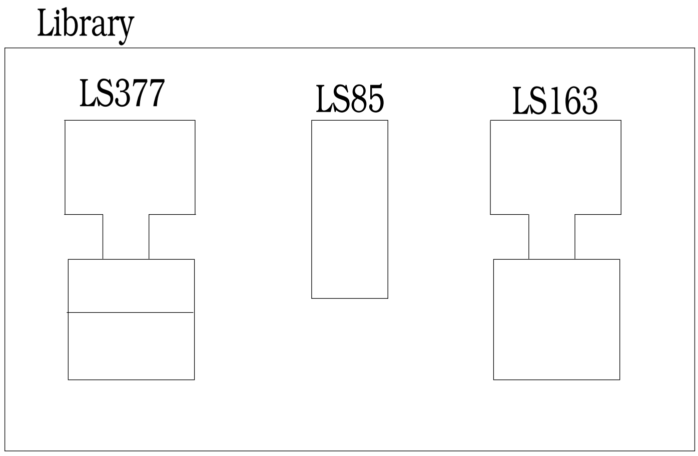
Behavioral.ASCII_IO

```
PROCEDURE   reading_from_file     (
                                  VARIABLE in_file: IN TEXT;
                                  ... other parameters ...
                                  );
```

```
PROCEDURE  writing_to_file  (
                            VARIABLE out_file : OUT TEXT;
                            ... other parameters...
                            );
```

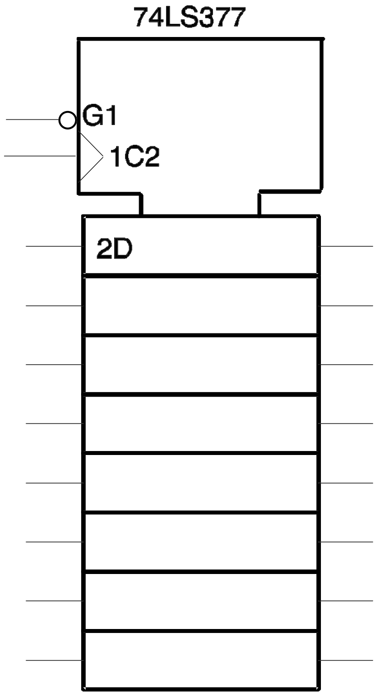
- To use procedure, declared file must be passed
- Declare FILE outside procedure
- Pass as IN or OUT variable
- Declare LINE inside procedure

Behavioral.MSI_design



- Design based on MSI parts
- Assume above parts are available

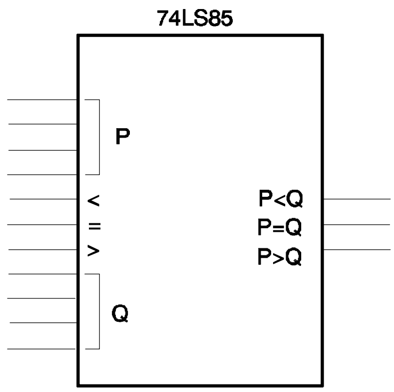
Behavioral.MSI_based_design




```
ENTITY ls377_register IS
  CONSTANT prop_delay : TIME := 7 NS;
  PORT (clk, g_bar : IN BIT; d8 : IN BIT_VECTOR (7 DOWNTO 0);
       q8 : OUT BIT_VECTOR (7 DOWNTO 0));
END ls377_register;
--
ARCHITECTURE dataflow OF ls377_register IS
BEGIN
  GUARD <= NOT clk'STABLE AND clk = '1' AND (g_bar = '0')
  q8 <= GUARDED d8 AFTER prop_delay;
END dataflow;
```

- 74LS377, clocked register
- Default delays can be used or reconfigured

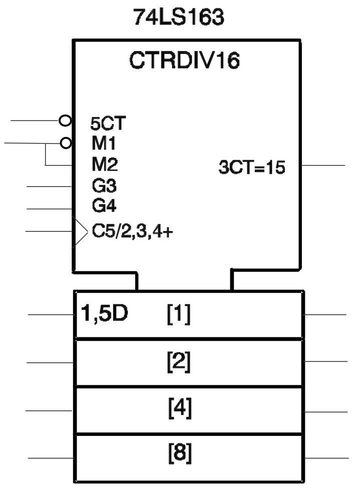
Behavioral.MSI_based_design




```
USE WORK.basic_utilities.ALL;
ENTITY ls85_comparator IS
  GENERIC (prop_delay : TIME := 10 NS);
  PORT (a, b : IN BIT_VECTOR (3 DOWNTO 0);  gt, eq, lt : IN BIT;
       a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END ls85_comparator;
--
ARCHITECTURE behavioral OF ls85_comparator IS
BEGIN
  PROCESS (a, b, gt, eq, lt)
    VARIABLE ai, bi : INTEGER;
  BEGIN
    bin2int (a, ai);   bin2int (b, bi);
    IF ai > bi THEN       a_eq_b <= '0' AFTER prop_delay;
       a_gt_b <= '1' AFTER prop_delay;       a_lt_b <= '0' AFTER prop_delay;
    ELSIF ai < bi THEN       a_eq_b <= '0' AFTER prop_delay;
       a_gt_b <= '0' AFTER prop_delay;       a_lt_b <= '1' AFTER prop_delay;
    ELSIF ai = bi THEN       a_eq_b <= eq AFTER prop_delay;
       a_gt_b <= gt AFTER prop_delay;       a_lt_b <= lt AFTER prop_delay;
    END IF;
  END PROCESS;
END behavioral;
```

- 74LS85, four bit comparator
- Default delays can be configured later

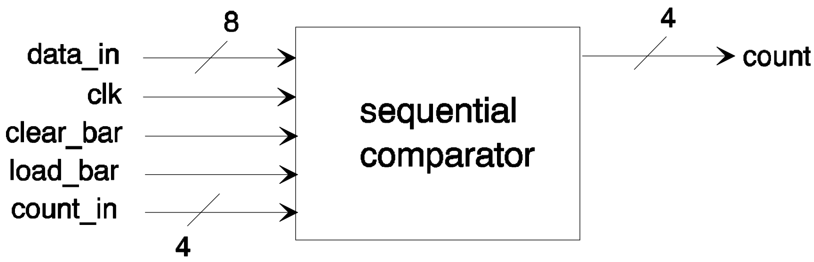
Behavioral.MSI_based_design

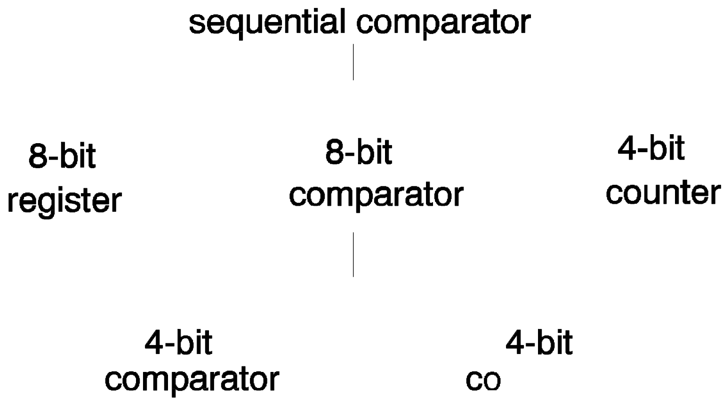



```
ENTITY ls163_counter IS
  GENERIC (prop_delay : TIME := 12 NS);
  PORT (clk, clr_bar, ld_bar, enp, ent : IN BIT;
       abcd : IN BIT_VECTOR (3 DOWNTO 0);
       q_abcd : OUT BIT_VECTOR (3 DOWNTO 0); rco : OUT BIT);
END ls163_counter;
--
ARCHITECTURE behavioral OF ls163_counter IS
BEGIN
  counting : PROCESS (clk)
    VARIABLE internal_count : BIT_VECTOR (3 DOWNTO 0) := "0000";
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
      IF (clr_bar = '0') THEN  internal_count := "0000";
      ELSIF (ld_bar = '0') THEN  internal_count := abcd;
      ELSIF (enp = '1' AND ent = '1') THEN
        internal_count := inc (internal_count);
        IF (internal_count = "1111") THEN   rco <= '1' AFTER prop_delay;
        ELSE   rco <= '0';   END IF;
      END IF;
      q_abcd <= internal_count AFTER prop_delay;
    END IF;
  END PROCESS counting;
END behavioral;
```

- 74LS163, four bit synchronous counter
- Default delays can be overwritten

Behavioral.MSI_based_design





ERROR: syntaxerror
OFFENDING COMMAND: ----nostringval----

STACK:

4