COE 405
Behavioral Descriptions in VHDL

Dr. Aiman H. El-Maleh
Computer Engineering Department
King Fahd University of Petroleum & Minerals

Outline

- **Constructs for Sequential Descriptions**
- **Process Statement**
- **Wait Statement**
- **Control Statements: Conditional & Iterative**
- **Behavioral Modeling of Mealy & Moore FSMs**
- **Assertion for Behavioral Checks**
- **Handshaking**
- **Formatted I/O**

Concurrent Versus Sequential Statements

Sequential Statements

•Used Within Process Bodies or SubPrograms

•Order Dependent

•Executed When Control is Transferred to the Sequential Body

- –**Assert**
- –**Signal Assignment**
- –**Procedure Call**
- –**Variable Assignment**
- –**IF Statements**
- –**Case Statement**
- –**Loops**
- –**Wait, Null, Next, Exit, Return**

Concurrent Statements

•Used Within Architectural Bodies or Blocks

•Order Independent

•Executed Once *At the Beginning of Simulation* or Upon Some Triggered Event

- –Assert
- –**Signal Assignment**
- –Procedure Call (*None of Formal Parameters May be of Type Variable*)
- –Process
- –**Block Statement**
- –**Component Statement**
- –**Generate Statement**
- –**Instantiation Statement**

Process Statement …

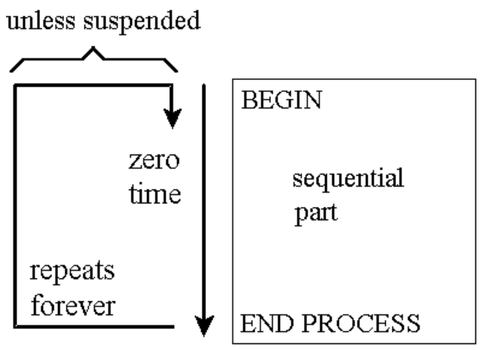
- **Main construct for Behavioral Modeling.**
- **Other concurrent statements can be modeled by an equivalent process.**
- **Process statement is a concurrent construct which performs a set of consecutive (Sequential) actions once it is Activated. Thus, only Sequential Statements are allowed within the Process Body.**

Optional ↓ (Process_Label) ***Optional*** ↓ (Sensitivity_List)

Process_Label: **PROCESS** (***Sensitivity_List***)
Process_Declarations; ← **Constant/Variables *No Signal Declarations Allowed***
Begin
<u>Sequential</u> Statements;
END Process;

… Process Statement …

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



… Process Statement

- **Whenever a SIGNAL in the *Sensitivity_List* of the Process changes, the Process is** Activated**.**
- **After executing the last statement, the Process is** SUSPENDED **Until one (or more) Signal in the Process *Sensitivity_List* changes value where it will be** REACTIVATED**.**
- **A Process Statement** *Without* **a *Sensitivity_List* is ALWAYS ACTIVE, i.e. After the Last Statement is Executed, execution returns to the First Statement and Continues (*Infinite Looping*).**
- **It is ILLEGAL to use WAIT-statement Inside a Process Which Has a *Sensitivity_List* .**
- **In case no *Sensitivity_List* exists, a Process may be activated or suspended using the** *WAIT***-statement.**
- **Conditional and selective signal assignments are strictly concurrent and cannot be used in a process.**

Process Examples

```
Process
Begin
        A<= `1`;
        B <= `0`;
End Process;
```

Sequential Processing:

- **First A is Scheduled to Have a Value `1`**
- **Second B is Scheduled to Have a Value `0`**
- **A & B Get their New Values At the SAME TIME (1 Delta Time Later)**

```
Process
Begin
   A<= `1`;
   IF  (A= `1`) Then Action1;
   Else  Action2;
   End IF;
End Process;
```

Assuming a `0` Initial Value of A,

- **First A is Scheduled to Have a Value `1` One Delta Time Later**
- **Thus, Upon Execution of IF_Statement, A Has a Value of `0` and *Action 2* will be Taken.**
- **If A *was Declared as a Process* Variable, *Action1* Would Have Been Taken**

Wait Statement

- **Syntax of Wait Statement :**
  - WAIT**;** -- **Forever**
  - WAIT ON *Signal_List*; -- **On event on a signal**
  - WAIT UNTIL *Condition***;** **-- until event makes condition true;**
  - WAIT FOR *Time_Out_Expression***;**
  - WAIT FOR **0** *any_time_unit***;** -- Process Suspended for 1 delta
- **When a** WAIT**-statement is Executed, the process** suspends **and conditions for its** Reactivation **are Set.**
- **Process Reactivation conditions may be Mixed as follows**
  - **WAIT ON** *Signal_List* **UNTIL** *Condition* **FOR** *Time_Expression* **;**
  - **wait on** X,Y **until** (Z = 0) **for** 70 NS**; --** Process Resumes After 70 NS **OR** (in Case X or Y Changes Value and Z=0 is True) ***Whichever Occurs First***
  - Process Reactivated IF:
    - Event Occurred on the *Signal_List* while the *Condition* is True*, OR*
    - *Wait* Period Exceeds ``*Time_Expression* ``

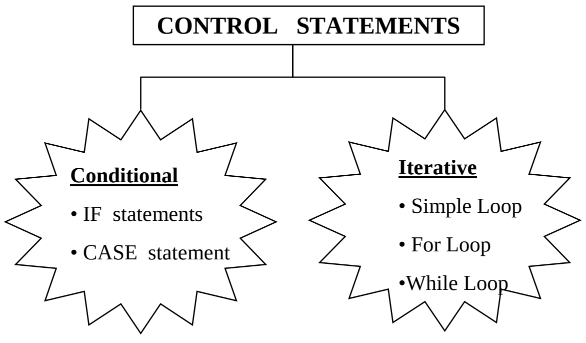
Positive Edge-Triggered D-FF Examples

```
D_FF: PROCESS (CLK)
        Begin
          IF (CLK`Event and CLK = `1`) Then
               Q <= D After TDelay;
          END IF;
        END Process;
```

```
D_FF: PROCESS        -- No Sensitivity_List
        Begin
          WAIT UNTIL  CLK = `1`;
               Q <= D After TDelay;
        END Process;
```

```
D_FF: PROCESS (Clk, Clr)  -- FF With Asynchronous Clear
        Begin
                      IF   Clr= `1`   Then  Q <= `0` After TD0;
                      ELSIF (CLK`Event and CLK = `1`) Then   Q <= D After TD1;
                      END IF;
        END Process;
```

Sequential Statements



Conditional Control – IF Statement

- **Syntax: 3-Possible Forms**

```
(i)   IF condition  Then
          statements;
      End IF;
(ii)  IF condition  Then
          statements;
      Else
          statements;
      End IF;
(iii) IF    condition  Then
          statements;
      Elsif condition  Then
          statements;
             ……….
      Elsif condition  Then
          statements;
      End IF;
```

Conditional Control – Case Statement

- **Syntax:**

```
(i)   CASE  Expression  is
         when  value => statements;
         when  value1 | value2| ...|valuen => statements;
         when  discrete range of values => statements;
         when  others => statements;
      End  CASE;
```

- Values/Choices should not overlap (*Any value of the Expression should Evaluate to only one Arm of the Case statement*).
- All possible choices for the *Expression* should be accounted for Exactly Once.

Conditional Control – Case Statement

- If ``**<u>others</u>**`` is used, it must be the last ``arm`` of the CASE statement.
- There can be Any Number of Arms in Any Order (*Except for the **<u>others</u>** arm which should be Last*)

```
CASE  x  is
          when  1 => y :=0;
          when  2 | 3  => y :=1;
          when  4 to 7 => y :=2;
          when  others  => y :=3;
End  CASE;
```

Loop Control …

- **Simple Loops**
- **Syntax:**

```
[Loop_Label:] LOOP
                      statements;
                   End LOOP [Loop_Label];
```

- **The Loop_Label is Optional**
- **The exit statement may be used to exit the Loop. It has two possible Forms:**
  - exit [Loop_Label]; -- This may be used in an if statement
  - exit [Loop_Label] when condition;

…Loop Control

```
Process
   variable A : Integer :=0;
   variable B : Integer :=1;
Begin
   Loop1: LOOP
               A := A + 1;
               B := 20;
               Loop2: LOOP
                        IF B < (A * A) Then
                           exit Loop2;
                        End IF;
                        B := B - A;
                        End LOOP Loop2;
               exit Loop1 when  A > 10;
          End  LOOP Loop1;
End Process;
```

FOR Loop

- **Syntax:**

Need Not Be Declared

[Loop_Label]: **FOR** *Loop_Variable* **in** *range* **LOOP**

statements;

End LOOP *Loop_Label***;**

```
Process
        variable B : Integer :=1;
Begin
        Loop1: FOR  A  in  1  TO  10   LOOP
                        B := 20;
                        Loop2: LOOP
                                IF B < (A * A) Then
                                    exit Loop2;
                                End IF;
                                B := B - A;
                              End LOOP Loop2;
                  End  LOOP Loop1;
End Process;
```

WHILE Loop

- **Syntax:**

[Loop_Label]: **WHILE** *condition* **LOOP**
statements;
End LOOP *Loop_Label*;

```
Process
        variable B:Integer :=1;
Begin
        Loop1: FOR  A  in  1  TO  10   LOOP
                        B := 20;
                        Loop2: WHILE  B < (A * A)  LOOP
                                B := B - A;
                              End LOOP Loop2;
                  End  LOOP Loop1;
End Process;
```

10-17

Next & Null Statements

- **Syntax:**

Next [*Loop_Label***] [When** *Condition***];**

- **Skip Current Loop Iteration When *Condition* is True**
- **If *Loop_Label* is Absent, innermost Loop is Skipped When *Condition* is True**
- **If *Condition* is Absent, Appropriate Loop Iteration is Skipped.**
- **Applicable for For Loops**

- **Null Statement**

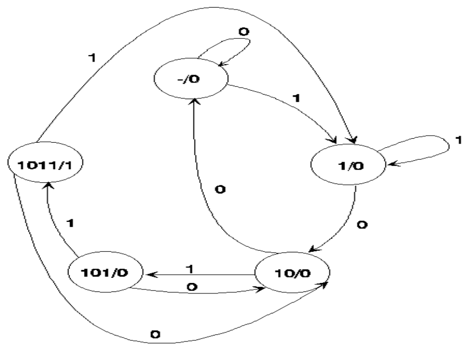
Syntax: **Null;**

- **Does Nothing**
- **Useful in Case Statements If No Action Is Required.**

10-18

A Moore 1011 Detector using Wait

```
ENTITY moore_detector IS
PORT (x, clk : IN BIT;
z : OUT BIT);
END moore_detector;
```



•*Can use WAIT in a Process statement to check for events on clk*

```
ARCHITECTURE behavioral_state_machine OF moore_detector IS
TYPE state IS (reset, got1, got10, got101, got1011);
SIGNAL current : state := reset;
BEGIN
```

A Moore 1011 Detector using Wait

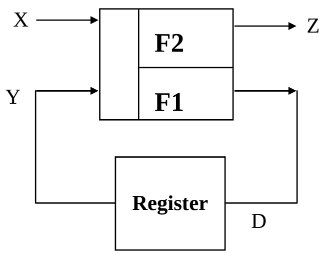
```
PROCESS
BEGIN
CASE current IS
WHEN reset =>  WAIT UNTIL clk = '1';
    IF x = '1' THEN current <= got1; ELSE current <= reset;  END IF;
WHEN got1 =>  WAIT UNTIL clk = '1';
    IF x = '0' THEN current <= got10; ELSE current <= got1;  END IF;
WHEN got10 => WAIT UNTIL clk = '1';
    IF x = '1' THEN 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;
```

A Moore 1011 Detector without Wait

```
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' and CLK'Event) 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;
```

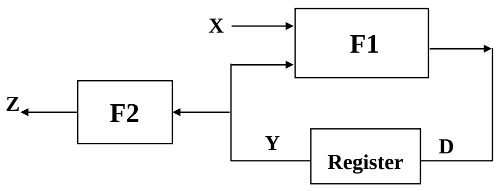
Generalized VHDL Mealy Model

```
Architecture  Mealy of  fsm  is
    Signal D, Y: Std_Logic_Vector( ...); -- Local Signals
Begin
Register: Process(Clk)
   Begin
     IF (Clk`EVENT and Clk = `1`)  Then  Y <= D;
     End IF;
   End Process;
Transitions: Process(X, Y)
   Begin
     D <= F1(X, Y);
   End Process;
Output: Process(X, Y)
   Begin
     Z <= F2(X, Y);
   End Process;
End Mealy;
```



Generalized VHDL Moore Model

```
Architecture  Moore of  fsm  is
    Signal D, Y: Std_Logic_Vector( ...); -- Local Signals
Begin
Register: Process( Clk)
   Begin
     IF (Clk`EVENT and Clk = `1`)  Then  Y <= D;
     End IF;
   End Process;
Transitions: Process(X, Y)
   Begin
     D <= F1(X, Y);
   End Process;
Output: Process(Y)
   Begin
     Z <= F2(Y);
   End Process;
End Moore;
```



FSM Example …

```
Entity   fsm  is
 port ( Clk, Reset    : in   Std_Logic;
                  X                  : in  Std_Logic_Vector(0 to 1);
                  Z                  : out Std_Logic_Vector(1 downto 0));
End   fsm;
```

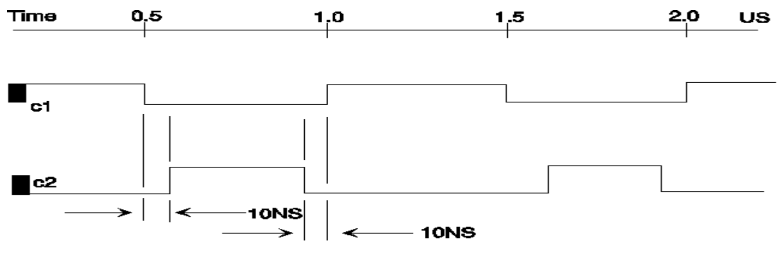
```
Architecture  behavior of fsm  is
              Type States is  (st0, st1, st2, st3);
              Signal Present_State, Next_State : States;
Begin
   reg: Process(Reset, Clk)
   Begin
         IF Reset = `1`  Then
                  Present_State <= st0; --  Machine Reset to st0
         elsIF (Clk`EVENT and Clk = `1`)  Then
                  Present_State <= Next_state;
         End IF;
   End Process;
```

… FSM Example

```
Transitions: Process(Present_State, X)
   Begin
              CASE Present_State is
                        when  st0  =>
                                    Z <= ``00``;
                                    IF X = ``11`` Then Next_State <= st0;
                                    else Next_State <= st1; End IF;
                        when  st1  =>
                                    Z <= ``01``;
                                    IF X = ``11`` Then Next_State <= st0;
                                    else Next_State <= st2; End IF;
                        when  st2  =>
                                    Z <= ``10``;
                                    IF X = ``11`` Then Next_State <= st2;
                                    else Next_State <= st3; End IF;
                        when  st3  =>
                                    Z <= ``11``;
                                    IF X = ``11`` Then Next_State <= st3;
                                    else Next_State <= st0; End IF;
              End CASE;
 End Process;
End behavior;
```

Using Wait for Two-Phase Clocking

```
c1 <= not c1 after 500ns;
phase2: PROCESS
BEGIN
WAIT UNTIL c1 = '0';
WAIT FOR 10 NS;
c2 <= '1';
WAIT FOR 480 NS;
c2 <= '0';
END PROCESS phase2;
. . .
```



Assert Statement …

- **Syntax:**

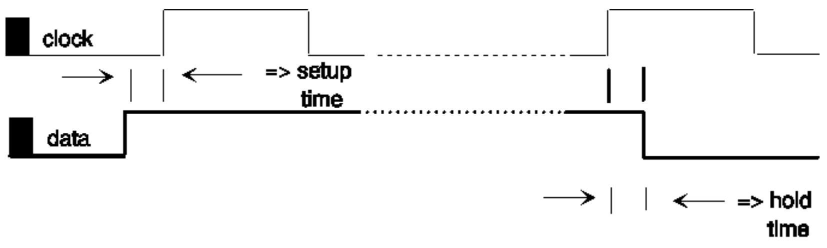
  ASSERT assertion_condition REPORT "reporting_message" SEVERITY severity_level;
- **Semantics**
  - Make sure that assertion_condition is true
  - Otherwise report "reporting message" then
  - Take the severity_level action
- **Severity: FAILURE ERROR WARNING NOTE**
- **Use assert to flag violations**
- **Use assert to report events**
- **Can be sequential or concurrent**

… Assert Statement

```
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

Checking for Setup & Hold Time …



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*

10-29

… Checking for Setup & Hold Time …

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*

10-30

… Checking for Setup & Hold Time …

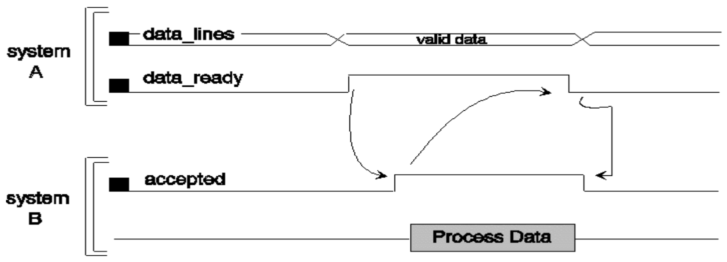
```
ENTITY d_sr_flipflop IS
GENERIC (sq_delay, rq_delay, cq_delay : TIME := 6 NS;
set_up, hold : TIME := 4 NS);
PORT (d, set, rst, clk : IN BIT; q, qb : OUT BIT);
BEGIN
ASSERT (NOT (clk = '1' AND clk'EVENT AND NOT d'STABLE(set_up) ))
REPORT "Set_up time violation"
SEVERITY WARNING;
ASSERT (NOT (d'EVENT AND clk = '1' AND NOT clk'STABLE(hold) ))
REPORT "Hold time violation"
SEVERITY WARNING;
END d_sr_flipflop;
```

- *Concurrent assertion statements*
- *Can be placed also in the architecture*

… Checking for Setup & Hold Time

```
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;
```

Handshaking …



System A

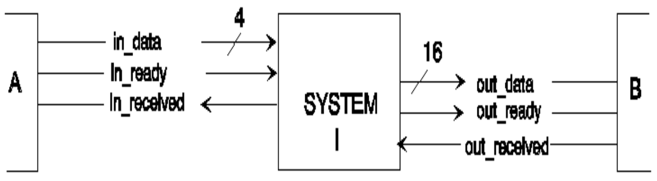
```
-- start the following when ready to send data
data_lines <= newly_prepared_data;
data_ready <= '1';
WAIT UNTIL accepted = '1';
data_ready <= '0';
-- can use data_lines for other purposes
```

System B

```
-- start the following when ready to accept data
WAIT UNTIL data_ready = '1';
 accepted <= '1';
-- start processing the newly received data
WAIT UNTIL data_ready = '0';
accepted <= '0';
```

… 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**



```
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;
```

… Handshaking …

```
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 talk to each other
•Use buffer_full, buffer_picked, and word_buffer for a_talk and b_talk communication

… 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;
```

Handshaking …

```
b_talk: PROCESS
BEGIN
-- communicate with a_talk process
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';
-- communicate with system B
out_ready <= '1';
WAIT UNTIL out_received = '1';
out_ready <= '0';
END PROCESS b_talk;
```

The IF buffer_full = ‘0’ statement is used so that the WAIT Until does not hold the process if buffer_full is already ‘1’ when this statement is reached

Formatted I/O …

- **USE STD.TEXTIO.ALL;**
- **l is LINE, f is FILE**
- **The following functions provided:**
  - READLINE (f, l)
  - READ (l, v)
  - WRITE (l, v),
  - WRITELINE (f, l)
  - ENDFILE (f)
- **READ or WRITE can read values of type:**
  - BIT, BIT_VECTOR, BOOLEAN, CHARACTER, INTEGER, REAL, STRING, TME

… Formatted I/O …

```
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";
BEGIN
FOR i IN sources'RANGE LOOP
WRITE (l, state'IMAGE(sources(I)), LEFT, 7);
END LOOP;
WRITELINE (flush, l);
RETURN sources (sources'LEFT);
END one_of;
```

•Add screen output to resolution function
•The 'IMAGE type attribute translates a state to its corresponding string
•The keyword LEFT specifies left justification
•7 specifies the string length

… Formatted I/O

```
USE STD.TEXTIO.ALL;
PROCEDURE display (SIGNAL value1,
value2 : BIT) IS
FILE flush : TEXT OPEN APPEND_MODE
is "debug.txt";
VARIABLE filler : STRING (1 TO 4) := " ...";
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