CPE/EE 422/522 Advanced Logic Design L08

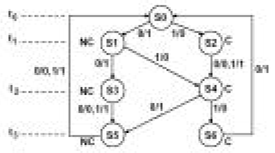
Electrical and Computer Engineering
University of Alabama in Huntsville

Outline

- What we know
  - How to model Combinational Networks in VHDL
    - Structural, Dataflow, Behavioral
  - How to model Flip-flops in VHDL
  - Processes
  - Delays (delta, transport, inertial)
  - How to model FSM in VHDL
  - Wait statements
  - Variables, Signals, Arrays
- What we do not know
  - VHDL Operators
  - Procedures, Functions
  - Packages, Libraries
  - Additional Topics (if time)

Review: Modeling a Sequential Machine

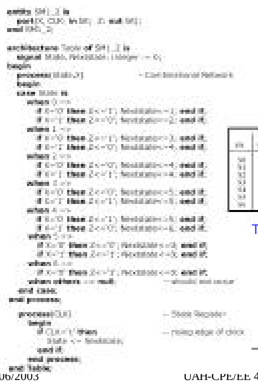
Mealy Machine for
8421 BCD to 8421 BCD + 3 bit serial converter



	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

How to model this in VHDL?

Review: Behavioral VHDL Model



```
entity SM1_2 is
  port(X, CLK: in bit; Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
  signal State, Nextstate: integer := 0;
begin
  process(State,X)                  --Combinational Network
  begin
  case State is
    when 0 =>
      if X='0' then Z<='1'; Nextstate<=1; end if;
      if X='1' then Z<='0'; Nextstate<=2; end if;
    when 1 =>
      if X='0' then Z<='1'; Nextstate<=3; end if;
      if X='1' then Z<='0'; Nextstate<=4; end if;
    when 2 =>
      if X='0' then Z<='0'; Nextstate<=4; end if;
      if X='1' then Z<='1'; Nextstate<=4; end if;
    when 3 =>
      if X='0' then Z<='0'; Nextstate<=5; end if;
      if X='1' then Z<='1'; Nextstate<=5; end if;
    when 4 =>
      if X='0' then Z<='1'; Nextstate<=5; end if;
      if X='1' then Z<='0'; Nextstate<=6; end if;
    when 5 =>
      if X='0' then Z<='0'; Nextstate<=0; end if;
      if X='1' then Z<='1'; Nextstate<=0; end if;
    when 6 =>
      if X='0' then Z<='1'; Nextstate<=0; end if;
    when others => null;            --should not occur
  end case;
end process;
  process(CLK)                      --State Register
    begin
      if CLK='1' then               --rising edge of clock
        State <= Nextstate;
      end if;
    end process;
end Table;
```

[illegible]

Two processes:
- the first represents the combinational network;
- the second represents the state register

Review: Wait Statements

- ... an alternative to a sensitivity list
  - Note: a process cannot have both wait statement(s) and a sensitivity list
- Generic form of a process with wait statement(s)

```
process
begin
   sequential-statements
   wait statement
   sequential-statements
   wait-statement
   ...
end process;
```

How wait statements work?

- Execute seq. statement until a wait statement is encountered.
- Wait until the specified condition is satisfied.
- Then execute the next set of sequential statements until the next wait statement is encountered.
- ...
- When the end of the process is reached start over again at the beginning.

Review: Forms of Wait Statements

```
wait on sensitivity-list;
wait for time-expression;
wait until boolean-expression;
```

- Wait on
  - until one of the signals in the sensitivity list changes
- Wait for
  - waits until the time specified by the time expression has elapsed
  - What is this:
    `wait for 0 ns;`
- Wait until
  - the boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE

Review: Variables

- What are they for:
  Local storage in processes, procedures, and functions
- Declaring variables

```
variable list_of_variable_names : type_name
[ := initial value ];
```

- Variables must be declared within the process in which they are used and are local to the process
  - Note: exception to this is SHARED variables

Review: Signals

- Signals must be declared outside a process
- Declaration form

```
signal list_of_signal_names : type_name
[ := initial value ];
```

- Declared in an architecture can be used anywhere within that architecture

Review: Constants

- Declaration form

```
constant constant_name : type_name := constant_value;

constant delay1 : time := 5 ns;
```

- Constants declared at the start of an architecture can be used anywhere within that architecture
- Constants declared within a process are local to that process

Review: Variables vs. Signals

- Variable assignment statement

```
variable_name := expression;
```

  – expression is evaluated and the variable is instantaneously updated
  (no delay, not even delta delay)

- Signal assignment statement

```
signal_name <= expression [after delay];
```

  – expression is evaluated and the signal is scheduled to change after delay; if no delay is specified the signal is scheduled to be updated after a delta delay

Review: Variables vs. Signals (cont'd)

Process Using Variables

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
    variable var1: integer:=1;
    variable var2: integer:=2;
    variable var3: integer:=3;
    begin
        wait on trigger;
        var1 := var2 + var3;
        var2 := var1;
        var3 := var2;
        sum <= var1 + var2 + var3;
    end process;
end var;
```

Sum = ?

Process Using Signals

```
entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
    begin
      wait on trigger;
      sig1 <= sig2 + sig3;
      sig2 <= sig1;
      sig3 <= sig2;
      sum <= sig1 + sig2 + sig3;
    end process;
end sig;
```

Sum = ?

Predefined VHDL Types

- Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type
- Predefined Types
  – bit – {'0', '1'}
  – boolean – {TRUE, FALSE}
  – integer – [$-2^{31} - 1 .. 2^{31} - 1$]
  – real – floating point number in range –1.0E38 to +1.0E38
  – character – legal VHDL characters including lower- uppercase letters, digits, special characters, ...
  – time – an integer with units fs, ps, ns, us, ms, sec, min, or hr

User Defined Type

- Common user-defined type is *enumerated*

```
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;
```

  - If no initialization, the default initialization is the leftmost element in the enumeration list (S0 in this example)

- VHDL is strongly typed language =>
  signals and variables of different types cannot be mixed in the same assignment statement,
  and no automatic type conversion is performed

Arrays

- Example

```
type SHORT_WORD is array (15 downto 0) of bit;

signal DATA_WORD : SHORT_WORD;
variable ALT_WORD : SHORT_WORD := “0101010101010101”;
constant ONE_WORD : SHORT_WORD := (others => ‘1’);
```

  - ALT_WORD(0) – rightmost bit
  - ALT_WORD(5 downto 0) – low order 6 bits

- General form

```
type arrayTypeName is array index_range of element_type;
signal arrayName : arrayTypeName [:=InitialValues];
```

Arrays (cont’d)

- Multidimensional arrays

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 :=
((1,2,3), (4,5,6), (7,8,9), (10,11,12));
```

  - matrixA(3, 2) = ?

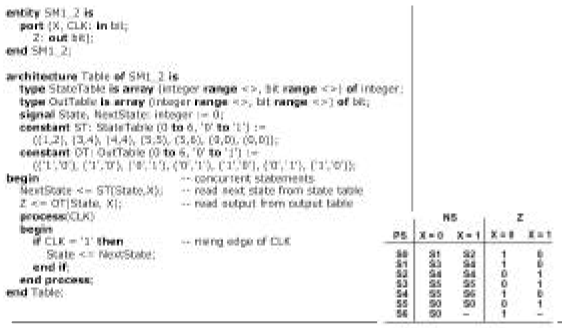
- Unconstrained array type

```
type intvec is array (natural range<>) of integer;

type matrix is array (natural range<>,natural range<>)
of integer;
```

  - range must be specified when the array object is declared

```
signal intvec5 : intvec(1 to 5) := (3,2,6,8,1);
```

Sequential Machine Model Using State Table

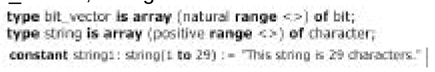


```
entity SM1_2 is
   port (X, CLK: in bit;
      Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
   type StateTable is array (integer range <>, bit range <>) of integer;
   type OutTable is array (integer range <>, bit range <>) of bit;
   signal State, NextState: integer := 0;
   constant ST: StateTable (0 to 6, '0' to '1') :=
      ((1,2), (3,4), (4,4), (5,5), (5,6), (0,0), (0,0));
   constant OT: OutTable (0 to 6, '0' to '1') :=
      (('1','0'), ('1','0'), ('0','1'), ('0','1'), ('1','0'), ('0','1'), ('1','0'));
begin                                  -- concurrent statements
   NextState <= ST(State,X);           -- read next state from state table
   Z <= OT(State, X);                  -- read output from output table
   process(CLK)
   begin
      if CLK = '1' then                -- rising edge of CLK
         State <= NextState;
      end if;
   end process;
end Table;
```

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

Predefined Unconstrained Array Types

- Bit_vector, string



```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
constant string1 : string(1 to 29) := "This string is 29 characters."
```

```
constant A : bit_vector(0 to 5) := “10101”;
-- (‘1’, ‘0’, ‘1’, ‘0’, ‘1’);
```

- Subtypes
  - include a subset of the values specified by the type

```
subtype SHORT_WORD is : bit_vector(15 to 0);
```

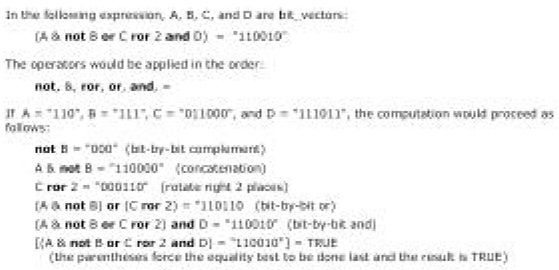
- Predefined subtypes of type integer
  - POSITIVE (all positive integers)
  - NATURAL (all positive integers and 0)

VHDL Operators

1. Binary logical operators: **and or nand nor xor xnor**
2. Relational: = /= < <= > >=
3. Shift: **sll srl sla sra rol ror**
4. Adding: + - & (concatenation)
5. Unary sign: + -
6. Multiplying: * / **mod rem**
7. Miscellaneous: **not abs** **

- Class 7 has the highest precedence (applied first), followed by class 6, then class 5, etc

Example of VHDL Operators



In the following expression, A, B, C, and D are bit_vectors:

(A & **not** B **or** C **ror** 2 **and** D) = "110010"

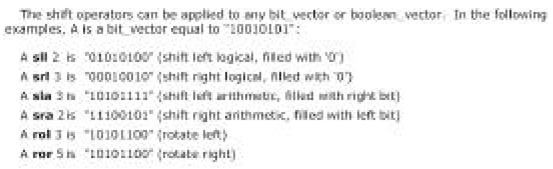
The operators would be applied in the order:

not, &, **ror**, **or**, **and**, =

If A = "110", B = "111", C = "011000", and D = "111011", the computation would proceed as follows:

not B = "000" (bit-by-bit complement)
A & **not** B = "110000" (concatenation)
C **ror** 2 = "000110" (rotate right 2 places)
(A & **not** B) **or** (C **ror** 2) = "110110 (bit-by-bit or)
(A & **not** B **or** C **ror** 2) **and** D = "110010" (bit-by-bit and)
[(A & **not** B **or** C **ror** 2 **and** D) = "110010"] = TRUE
(the parentheses force the equality test to be done last and the result is TRUE)

Example of Shift Operators



The shift operators can be applied to any bit_vector or boolean_vector. In the following examples, A is a bit_vector equal to "10010101":

A **sll** 2 is "01010100" (shift left logical, filled with '0')
A **srl** 3 is "00010010" (shift right logical, filled with '0')
A **sla** 3 is "10101111" (shift left arithmetic, filled with right bit)
A **sra** 2 is "11100101" (shift right arithmetic, filled with left bit)
A **rol** 3 is "10101100" (rotate left)
A **ror** 5 is "10101100" (rotate right)

VHDL Functions

- Functions execute a sequential algorithm and return a single value to calling program

```
function rotate_right (reg: bit_vector)
    return bit_vector is
begin
    return  reg ror 1;
end rotate_right;
```

- A = “10010101”

```
B <= rotate_right(A);
```

- General form

```
function function-name (formal-parameter-list)
      return return-type is
      [declarations]
begin
      sequential statements  -- must include return return-value;
end function-name;
```

For Loops

General form of a for loop:

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

Exit statement has the form:

```
exit;                    -- or
exit when condition;
```

For Loop Example:

```
-- compare two 8-character strings and return TRUE if equal
function comp_string(string1, string2: string(1 to 8))
      return boolean is

variable B: boolean;
begin
      loopex: for j in 1 to 8 loop
            B := string1(j) = string2(j);
            exit when B=FALSE;
      end loop loopex;
      return B;
end comp_string;
```

Add Function

```
-- This function adds 2 4-bit vectors and a carry.
-- It returns a 5-bit sum

function add4 (A,B: bit_vector(3 downto 0); carry: bit)
   return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
loop1: for i in 0 to 3 loop
   cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
   Sum(i) := A(i) xor B(i) xor cin;
   cin := cout;
end loop loop1;
Sum(4):= cout;
return Sum;
end add4;
```

Example function call:

```
Sum1 <= add4(A1, B1, cin);
```

VHDL Procedures

- Facilitate decomposition of VHDL code into modules
- Procedures can return any number of values using output parameters
- General form

```
procedure procedure_name (formal-parameter-list) is
   [declarations]
   begin
     Sequential-statements
   end procedure_name;


procedure_name (actual-parameter-list);
```

Procedure for Adding Bit_vectors

```
-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry.  Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.

procedure Addvec
   (Add1,Add2: in bit_vector;
      Cin: in bit;
      signal Sum: out bit_vector;
      signal Cout: out bit;
      n:in positive) is
      variable C: bit;
begin
   C := Cin;
   for i in 0 to n-1 loop
      Sum(i) <= Add1(i) xor Add2(i) xor C;
      C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
   end loop;
   Cout <= C;
end Addvec;
```

Example procedure call:

```
Addvec(A1, B1, Cin, Sum1, Cout, 4);
```

Parameters for Subprogram Calls

		Actual Parameter	
Mode	Class	Procedure Call	Function Call
in[1]	constant[2]	expression	expression
	signal	signal	signal
	variable	variable	n/a
out/inout	signal	signal	n/a
	variable[3]	variable	n/a

[1] default mode for functions [2] default for in mode [3] default for out/inout mode

Packages and Libraries

- Provide a convenient way of referencing frequently used functions and components
- Package declaration

```
package package-name is
   package declarations
end [package][package-name];
```

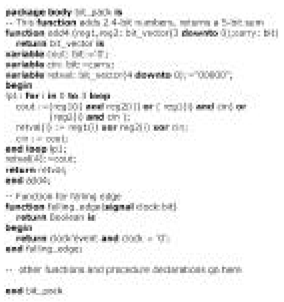
- Package body [optional]

```
package body package-name is
   package body declarations
end [package body][package name];
```

Library BITLIB – bit_pack package

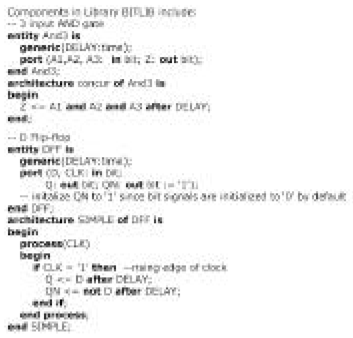
```
package bit_pack is
   function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
      return bit_vector;
   function falling_edge(signal clock:bit)
      return Boolean ;
   function rising_edge(signal clock:bit)
      return Boolean ;
   function vec2int(vec1: bit_vector)
      return integer;
   function int2vec(int1,NBits: integer)
      return bit_vector;
   procedure Addvec
      (Add1,Add2: in bit_vector;
         Cin: in bit;
         signal Sum: out bit_vector;
         signal Cout: out bit;
         n: in natural);

   component jkff
      generic(DELAY:time := 10 ns);
      port(SN, RN, J,K,CLK: in bit; Q, QN: inout bit);
   end component;

   component dff
      generic(DELAY:time := 10 ns);
      port (D, CLK: in bit; Q: out bit; QN: out bit := '1');
   end component;

   component and2
      generic(DELAY:time := 10 ns);
      port(A1, A2: in bit; Z: out bit);
   end component;

   component and3
      generic(DELAY:time := 10 ns);
      port(A1, A2, A3: in bit; Z: out bit);
   end component;

   component and4
      generic(DELAY:time := 10 ns);
      port(A1, A2, A3, A4: in bit; Z: out bit);
   end component;

   component or2
      generic(DELAY:time := 10 ns);
      port(A1, A2: in bit; Z: out bit);
   end component;

   component or3
      generic(DELAY:time := 10 ns);
      port(A1, A2, A3: in bit; Z: out bit);
   end component;
--
   (other component declarations go here)
--

end bit_pack;
```

Library BITLIB – bit_pack package



```
package body bit_pack is
-- This function adds 2 4-bit numbers, returns a 5-bit sum
function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
   return bit_vector is
variable cout: bit:='0';
variable cin: bit:=carry;
variable retval: bit_vector(4 downto 0):="00000";
begin
lp1: for i in 0 to 3 loop
   cout :=(reg1(i) and reg2(i)) or ( reg1(i) and cin) or
          (reg2(i) and cin );
   retval(i) := reg1(i) xor reg2(i) xor cin;
   cin := cout;
end loop lp1;
retval(4):=cout;
return retval;
end add4;

-- Function for falling edge
function falling_edge(signal clock:bit)
   return Boolean is
begin
   return clock'event and clock = '0';
end falling_edge;

-- other functions and procedure declarations go here

end bit_pack
```

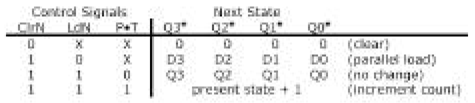
Library BITLIB – bit_pack package



```
Components in Library BITLIB include:
-- 3 input AND gate
entity And3 is
   generic(DELAY:time);
   port (A1,A2, A3:  in bit; Z: out bit);
end And3;
architecture concur of And3 is
begin
   Z <= A1 and A2 and A3 after DELAY;
end;

-- D flip-flop
entity DFF is
   generic(DELAY:time);
   port (D, CLK: in bit;
         Q: out bit; QN: out bit := '1');
   -- initialize QN to '1' since bit signals are initialized to '0' by default
end DFF;
architecture SIMPLE of DFF is
begin
   process(CLK)
   begin
      if CLK = '1' then  --rising edge of clock
         Q <= D after DELAY;
         QN <= not D after DELAY;
      end if;
   end process;
end SIMPLE;
```

VHDL Model for a 74163 Counter

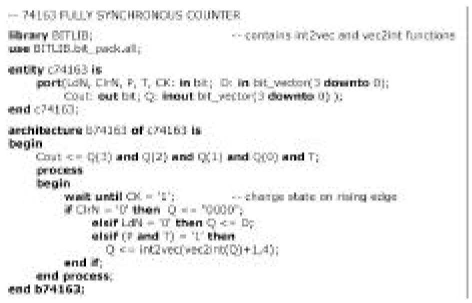
- 74613 – 4-bit fully synchronous binary counter
- Counter operations



Control Signals			Next State				
ClrN	LdN	P•T	Q3*	Q2*	Q1*	Q0*	
0	X	X	0	0	0	0	(clear)
1	0	X	D3	D2	D1	D0	(parallel load)
1	1	0	Q3	Q2	Q1	Q0	(no change)
1	1	1	present state + 1				(increment count)

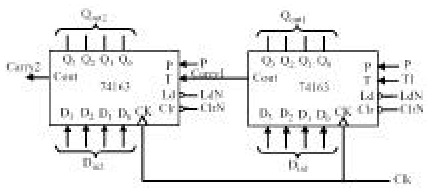
- Generate a Cout in state 15 if T=1
  - Cout = $Q_3Q_2Q_1Q_0T$

VHDL Model for a 74163 Counter

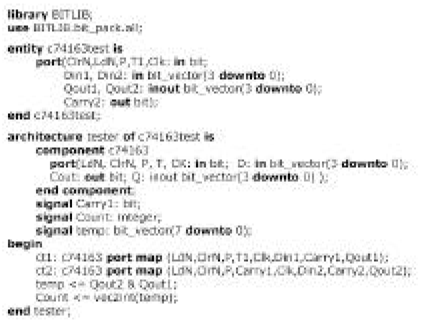


```
-- 74163 FULLY SYNCHRONOUS COUNTER
library BITLIB;                    -- contains int2vec and vec2int functions
use BITLIB.bit_pack.all;

entity c74163 is
      port(LdN, ClrN, P, T, CK: in bit;  D: in bit_vector(3 downto 0);
            Cout: out bit; Q: inout bit_vector(3 downto 0) );
end c74163;

architecture b74163 of c74163 is
begin
      Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
      process
      begin
            wait until CK = '1';           -- change state on rising edge
            if ClrN = '0' then  Q <= "0000";
                  elsif LdN = '0' then Q <= D;
                  elsif (P and T) = '1' then
                     Q <= int2vec(vec2int(Q)+1,4);
            end if;
      end process;
end b74163;
```

Cascaded Counters



Cascaded Counters (cont'd)



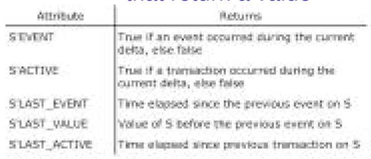
```
library BITLIB;
use BITLIB.bit_pack.all;

entity c74163test is
      port(ClrN,LdN,P,T1,Clk: in bit;
            Din1, Din2: in bit_vector(3 downto 0);
            Qout1, Qout2: inout bit_vector(3 downto 0);
            Carry2: out bit);
end c74163test;

architecture tester of c74163test is
      component c74163
         port(LdN, ClrN, P, T, CK: in bit;  D: in bit_vector(3 downto 0);
         Cout: out bit; Q: inout bit_vector(3 downto 0) );
      end component;
      signal Carry1: bit;
      signal Count: integer;
      signal temp: bit_vector(7 downto 0);
begin
      ct1: c74163 port map (LdN,ClrN,P,T1,Clk,Din1,Carry1,Qout1);
      ct2: c74163 port map (LdN,ClrN,P,Carry1,Clk,Din2,Carry2,Qout2);
      temp <= Qout2 & Qout1;
      Count <= vec2int(temp);
end tester;
```

Additional Topics in VHDL

- Attributes
- Transport and Inertial Delays
- Operator Overloading
- Multivalued Logic and Signal Resolution
- IEEE 1164 Standard Logic
- Generics
- Generate Statements
- Synthesis of VHDL Code
- Synthesis Examples
- Files and Text IO

Signal Attributes

Attributes associated with signals
that return a value



Attribute	Returns
S'EVENT	True if an event occurred during the current delta, else false
S'ACTIVE	True if a transaction occurred during the current delta, else false
S'LAST_EVENT	Time elapsed since the previous event on S
S'LAST_VALUE	Value of S before the previous event on S
S'LAST_ACTIVE	Time elapsed since previous transaction on S

A'event – true if a change in S has just occurred

A'active – true if A has just been reevaluated, even if A does not change

Signal Attributes (cont’d)

- Event
  - – occurs on a signal every time it is changed
- Transaction
  - – occurs on a signal every time it is evaluated
- Example:

```
A <= B  - -  B changes at time T
```

	A’event	B’event
T		
T + 1d		

Signal Attributes (cont’d)

```
entity test is
end;
architecture bmtest of test is
   signal A : bit;
   signal B : bit;
   signal C : bit;
begin
   A <= not A after 20 ns;
   B <= '1';
   C <= A and B;
  process(A, B, C)
   variable Aev : bit;
   variable Aac : bit;
   variable Bev : bit;
   variable Bac : bit;
   variable Cev : bit;
   variable Cac : bit;
```

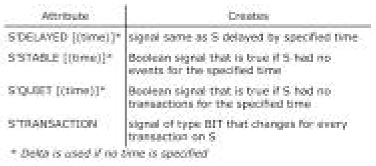
```
begin
   if (A'event ) then Aev := '1';
   else Aev := '0';
   end if;
   if (A'active) then Aac := '1';
   else Aac := '0';
   end if;
   if (B'event ) then Bev := '1';
   else Bev := '0';
   end if;
   if (B'active) then Bac := '1';
   else Bac := '0';
   end if;
   if (C'event ) then Cev := '1';
   else Cev := '0';
   end if;
   if (C'active) then Cac := '1';
   else Cac := '0';
   end if;

end process;

end bmtest;
```

Signal Attributes (cont’d)

```
ns          /test/a   /test/line__15/bev
         delta     /test/b   /test/line__15/bac
                     /test/c   /test/line__15/cev
            /test/line__15/aev   /test/line__15/cac
              /test/line__15/aac
  0  +0        0 0 0 0 0           0 0 0 0
  0  +1        0 1 0 0 0           1 1 0 1
 20  +0        1 1 0 1 1           0 0 0 0
 20  +1        1 1 1 0 0           0 0 1 1
 40  +0        0 1 1 1 1           0 0 0 0
 40  +1        0 1 0 0 0           0 0 1 1
```

Signal Attributes (cont’d)

Attributes that create a signal



Attribute	Creates
S'DELAYED [(time)]*	signal same as S delayed by specified time
S'STABLE [(time)]*	Boolean signal that is true if S had no events for the specified time
S'QUIET [(time)]*	Boolean signal that is true if S had no transactions for the specified time
S'TRANSACTION	signal of type BIT that changes for every transaction on S

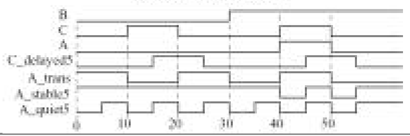
* *Delta is used if no time is specified*

Examples of Signal Attributes

VHDL Code for Attribute Test

```
entity attr_ex is
   port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
   signal A, C_delayed5, A_trans : bit;
   signal A_stable5, A_quiet5 : boolean;
begin
   A <= B and C;
   C_delayed5 <= C'delayed(5 ns);
   A_trans <= A'transaction;
   A_stable5 <= A'stable(5 ns);
   A_quiet5 <= A'quiet(5 ns);
end test;
```

Waveforms for Attribute Test

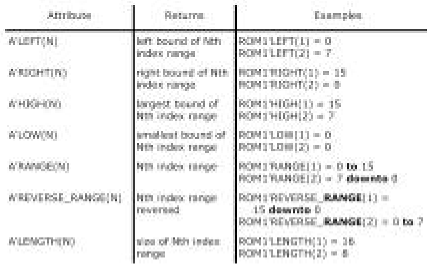


Using Attributes for Error Checking

```
check: process
begin
   wait until rising_edge(Clk);
   assert (D’stable(setup_time))
        report(“Setup time violation”)
        severity error;
   wait for hold_time;
   assert (D’stable(hold_time))
        report(“Hold time violation”)
        severity error;
end process check;
```

Array Attributes

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```



Attribute	Returns	Examples
A'LEFT(N)	left bound of Nth index range	ROM1'LEFT(1) = 0 ROM1'LEFT(2) = 7
A'RIGHT(N)	right bound of Nth index range	ROM1'RIGHT(1) = 15 ROM1'RIGHT(2) = 0
A'HIGH(N)	largest bound of Nth index range	ROM1'HIGH(1) = 15 ROM1'HIGH(2) = 7
A'LOW(N)	smallest bound of Nth index range	ROM1'LOW(1) = 0 ROM1'LOW(2) = 0
A'RANGE(N)	Nth index range	ROM1'RANGE(1) = 0 **to** 15 ROM1'RANGE(2) = 7 **downto** 0
A'REVERSE_RANGE(N)	Nth index range reversed	ROM1'REVERSE_**RANGE**(1) = 15 **downto** 0 ROM1'REVERSE_**RANGE**(2) = 0 **to** 7
A'LENGTH(N)	size of Nth index range	ROM1'LENGTH(1) = 16 ROM1'LENGTH(2) = 8

A can be either an array name or an array type.

Array attributes work with signals, variables, and constants.

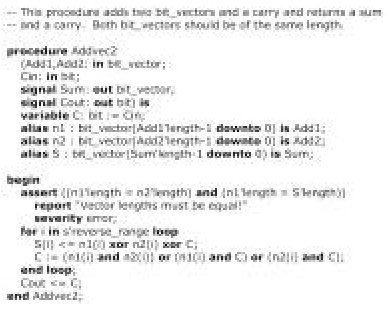
Recap: Adding Vectors

```
-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry.  Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.

procedure Addvec
   (Add1,Add2: in bit_vector;
      Cin: in bit;
      signal Sum: out bit_vector;
      signal Cout: out bit;
      n:in positive) is
      variable C: bit;
begin
   C := Cin;
   for i in 0 to n-1 loop
      Sum(i) <= Add1(i) xor Add2(i) xor C;
      C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
   end loop;
   Cout <= C;
end Addvec;
```

Note: Add1 and Add2 vectors must be dimensioned as *N-1 downto 0.*

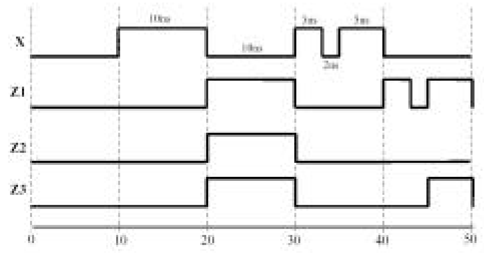
Use attributes to write more general procedure that places
no restrictions on the range of vectors other than the lengths must be same.

Procedure for Adding Bit Vectors



```
-- This procedure adds two bit_vectors and a carry and returns a sum
-- and a carry.  Both bit_vectors should be of the same length.

procedure Addvec2
   (Add1,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit) is
   variable C: bit := Cin;
   alias n1 : bit_vector(Add1'length-1 downto 0) is Add1;
   alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
   alias S : bit_vector(Sum'length-1 downto 0) is Sum;

begin
   assert ((n1'length = n2'length) and (n1'length = S'length))
      report "Vector lengths must be equal!"
      severity error;
   for i in s'reverse_range loop
      S(i) <= n1(i) xor n2(i) xor C;
      C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
   end loop;
   Cout <= C;
end Addvec2;
```

Transport and Inertial Delay



```
Z1 <= transport X after 10 ns;    -- transport delay
Z2 <= X after 10 ns;              -- inertial delay
Z3 <= reject 4 ns X after 10 ns;  -- delay with specified rejection pulse width
```

Transport and Inertial Delay (cont'd)

```
Z3 <= reject 4 ns X after 10 ns;
```

Reject is equivalent to a combination of inertial and transport delay:

```
Zm <= X after 4 ns;
Z3 <= transport Zm after 6 ns;
```

Statements executed at time T
– B at T+1, C at T+2

```
A <= transport B after 1 ns;
A <= transport C after 2 ns;
```

Statements executed at time T
– C at T + 2:

```
A <= B after 1 ns;
A <= C after 2 ns;
```

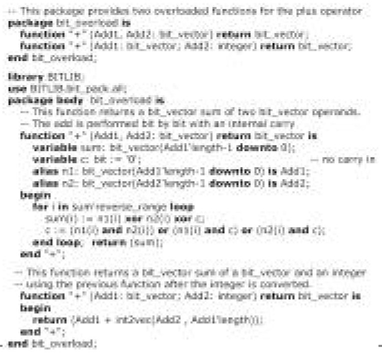
Statements executed at time T
– C at T + 1:

```
A <= transport B after 2 ns;
A <= transport C after 1 ns;
```

Operator Overloading

- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction
  – addvec, subvec
- Operator overloading allows using + operator to implicitly call an appropriate addition function
- How does it work?
  – When compiler encounters a function declaration in which the function name is an operator enclosed in double quotes, the compiler treats the function as an operator overloading ("+")
  – when a "+" operator is encountered, the compiler automatically checks the types of operands and calls appropriate functions

VHDL Package with Overloaded Operators



```
-- This package provides two overloaded functions for the plus operator
package bit_overload is
   function "+" (Add1, Add2: bit_vector) return bit_vector;
   function "+" (Add1: bit_vector; Add2: integer) return bit_vector;
end bit_overload;

library BITLIB;
use BITLIB.bit_pack.all;
package body bit_overload is
   -- This function returns a bit_vector sum of two bit_vector operands.
   -- The add is performed bit by bit with an internal carry
   function "+" (Add1, Add2: bit_vector) return bit_vector is
      variable sum: bit_vector(Add1'length-1 downto 0);
      variable c: bit := '0';                          -- no carry in
      alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
      alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
   begin
      for i in sum'reverse_range loop
         sum(i) := n1(i) xor n2(i) xor c;
         c := (n1(i) and n2(i)) or (n1(i) and c) or (n2(i) and c);
      end loop;   return (sum);
   end "+";

  -- This function returns a bit_vector sum of a bit_vector and an integer
  -- using the previous function after the integer is converted.
   function "+" (Add1: bit_vector; Add2: integer) return bit_vector is
   begin
      return (Add1 + int2vec(Add2 , Add1'length));
   end "+";
end bit_overload;
```

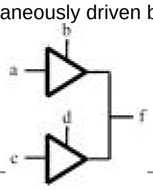
Overloaded Operators

- A, B, C – bit vectors
- A <= B + C + 3 ?
- A <= 3 + B + C ?

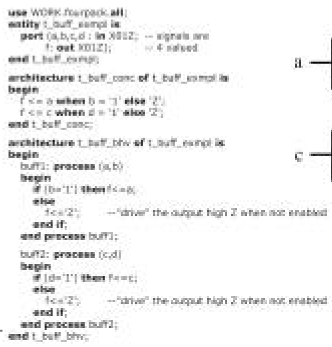
- Overloading can also be applied to procedures and functions
  - procedures have the same name – type of the actual parameters in the procedure call determines which version of the procedure is called

Multivalued Logic

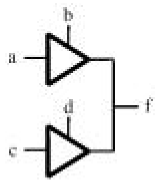
- Bit (0, 1)
- Tristate buffers and buses => high impedance state 'Z'
- Unknown state 'X'
  - e. g., a gate is driven by 'Z', output is unknown
  - a signal is simultaneously driven by '0' and '1'



Tristate Buffers



```
use WORK.fourpack.all;
entity t_buff_exmpl is
   port (a,b,c,d : in X01Z;  -- signals are
         f: out X01Z);        -- 4 valued
end t_buff_exmpl;

architecture t_buff_conc of t_buff_exmpl is
begin
   f <= a when b = '1' else 'Z';
   f <= c when d = '1' else 'Z';
end t_buff_conc;

architecture t_buff_bhv of t_buff_exmpl is
begin
   buff1: process (a,b)
   begin
      if (b='1') then f<=a;
      else
         f<='Z';      --"drive" the output high Z when not enabled
      end if;
   end process buff1;

   buff2: process (c,d)
   begin
      if (d='1') then f<=c;
      else
         f<='Z';      --"drive" the output high Z when not enabled
      end if;
   end process buff2;
end t_buff_bhv;
```



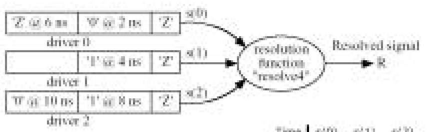
Resolution function to determine the actual value of f since it is driven from two different sources

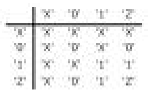
Signal Resolution

- VHDL signals may either be resolved or unresolved
- Resolved signals have an associated resolution function
- Bit type is unresolved –
  - there is no resolution function
  - if you drive a bit signal to two different values in two concurrent statements, the compiler will generate an error

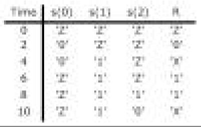
Signal Resolution (cont’d)

```
signal R : X01Z := ‘Z’; ...
R <= transport ‘0’ after 2 ns, ‘Z’ after 6 ns;
R <= transport ‘1’ after 4 ns;
R <= transport ‘1’ after 8 ns, ‘0’ after 10 ns;
```



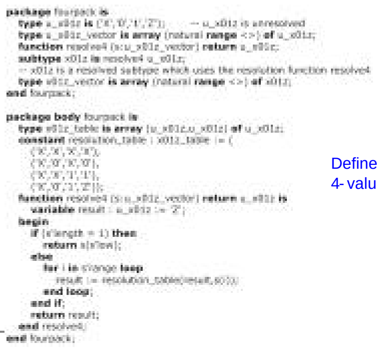


	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'X'	'X'
'0'	'X'	'0'	'X'	'0'
'1'	'X'	'X'	'1'	'1'
'Z'	'X'	'0'	'1'	'Z'



Time	s(0)	s(1)	s(2)	R
0	'Z'	'Z'	'Z'	'Z'
2	'0'	'Z'	'Z'	'0'
4	'0'	'1'	'Z'	'X'
6	'Z'	'1'	'Z'	'1'
8	'Z'	'1'	'1'	'1'
10	'Z'	'1'	'0'	'X'

Resolution Function for X01Z



```
package fourpack is
  type u_x01z is ('X','0','1','Z');       -- u_x01z is unresolved
  type u_x01z_vector is array (natural range <>) of u_x01z;
  function resolve4 (s:u_x01z_vector) return u_x01z;
  subtype x01z is resolve4 u_x01z;
  -- x01z is a resolved subtype which uses the resolution function resolve4
  type x01z_vector is array (natural range <>) of x01z;
end fourpack;

package body fourpack is
  type x01z_table is array (u_x01z,u_x01z) of u_x01z;
  constant resolution_table : x01z_table := (
    ('X','X','X','X'),
    ('X','0','X','0'),
    ('X','X','1','1'),
    ('X','0','1','Z'));
  function resolve4 (s:u_x01z_vector) return u_x01z is
    variable result : u_x01z := 'Z';
  begin
    if (s'length = 1) then
      return s(s'low);
    else
      for i in s'range loop
        result := resolution_table(result,s(i));
      end loop;
    end if;
    return result;
  end resolve4;
end fourpack;
```

Define AND and OR for 4-valued inputs?

AND and OR Functions Using X01Z

AND	‘X’	‘0’	‘1’	‘Z’
‘X’	‘X’	‘0’	‘X’	‘X’
‘0’	‘0’	‘0’	‘0’	‘0’
‘1’	‘X’	‘0’	‘1’	‘X’
‘Z’	‘X’	‘0’	‘X’	‘X’

OR	‘X’	‘0’	‘1’	‘Z’
‘X’	‘X’	‘X’	‘1’	‘X’
‘0’	‘X’	‘0’	‘1’	‘X’
‘1’	‘1’	‘1’	‘1’	‘1’
‘Z’	‘X’	‘X’	‘1’	‘X’

IEEE 1164 Standard Logic

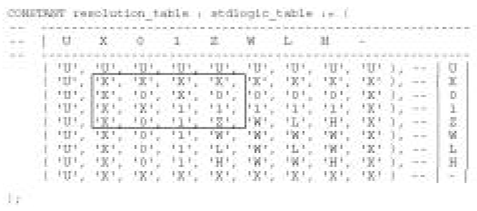
- 9-valued logic system
  - ‘U’ – Uninitialized
  - ‘X’ – Forcing Unknown
  - ‘0’ – Forcing 0
  - ‘1’ – Forcing 1
  - ‘Z’ – High impedance
  - ‘W’ – Weak unknown
  - ‘L’ – Weak 0
  - ‘H’ – Weak 1
  - ‘-’ – Don’t care

If forcing and weak signal are tied together, the forcing signal dominates.

Useful in modeling the internal operation of certain types of ICs.

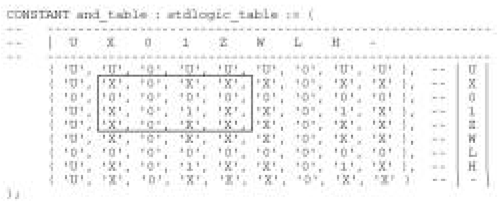
In this course we use a subset of the IEEE values: X10Z

Resolution Function for IEEE 9-valued



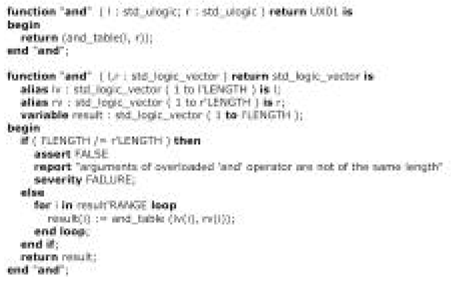
```
CONSTANT resolution_table : stdlogic_table := (
--  ---------------------------------------------------------
--  | U    X    0    1    Z    W    L    H    -
--  ---------------------------------------------------------
    ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
    ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
    ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
    ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
    ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
    ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
    ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
);
```

AND Table for IEEE 9-valued



```
CONSTANT and_table : stdlogic_table := (
--  ---------------------------------------------------------
--  | U    X    0    1    Z    W    L    H    -
--  ---------------------------------------------------------
    ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
    ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
    ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
    ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
    ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
    ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);
```

AND Function for std_logic_vectors



```
function "and" ( l : std_ulogic; r : std_ulogic ) return UX01 is
begin
   return (and_table(l, r));
end "and";

function "and" ( l,r : std_logic_vector ) return std_logic_vector is
   alias lv : std_logic_vector ( 1 to l'LENGTH ) is l;
   alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
   variable result : std_logic_vector ( 1 to l'LENGTH );
begin
   if ( l'LENGTH /= r'LENGTH ) then
      assert FALSE
      report "arguments of overloaded 'and' operator are not of the same length"
      severity FAILURE;
   else
      for i in result'RANGE loop
         result(i) := and_table (lv(i), rv(i));
      end loop;
   end if;
   return result;
end "and";
```