Agenda

- **Introduction**
- **CIC Design Flow**
- **VHDL Fundamentals**
  - Structure
  - Simulation
  - Data Types
  - Attribute
  - Expression
  - Hierarchy
  - Generic

Agenda

- **VHDL Syntax**
  - Sequential Statements
  - Concurrent Statements
- **Modeling logic circuits**
  - Combinational Logic
  - Sequential Logic
  - Finite State Machine
- **Testbench**
- **Gate-Level Simulation**

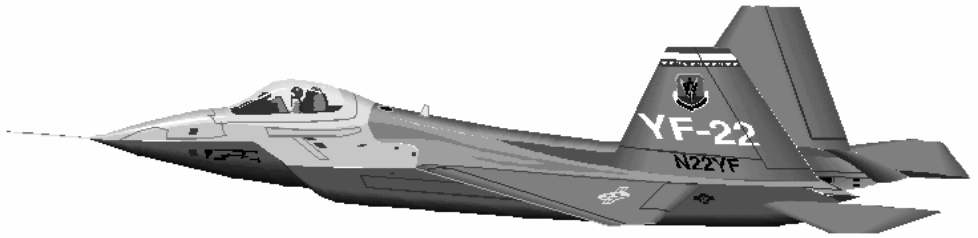
Introduction

Introduction (1/4)

- **VHDL**
  - ***V***ery High Speed Integrated Circuit ***H***ardware ***D***escription ***L***anguage
- **1980**
  - The USA Department of Defense **(*DOD*)**
  - project under the Very High Speed Integrated Circuit **(*VHSIC*)**

Introduction (2/4)

- **1983**
  - IBM, Texas Instruments and Intermetrics.
  - The DOD mandated that all digital electronic circuits be described in VHDL.
  - IEEE standard 1076 . (1987)
- **The F22 advanced tactical fighter aircraft**



Introduction (3/4)

- **1993**
  - IEEE 1076 '93.
- **1996**
  - Both commercial simulation and synthesis tools became available adhering to IEEE 1076 '93 standard.
  - A VHDL package for use with synthesis tools, IEEE 1076.3 **(*NUMERIC_STD*)**
  - IEEE 1076.4 **(*VITAL*)**

Introduction (4/4)

- **1997**
  - VHDL-AMS(***IEEE1076.1***)
    - supporting the description and simulation of digital, analog, and mixed-signal systems in a single language.

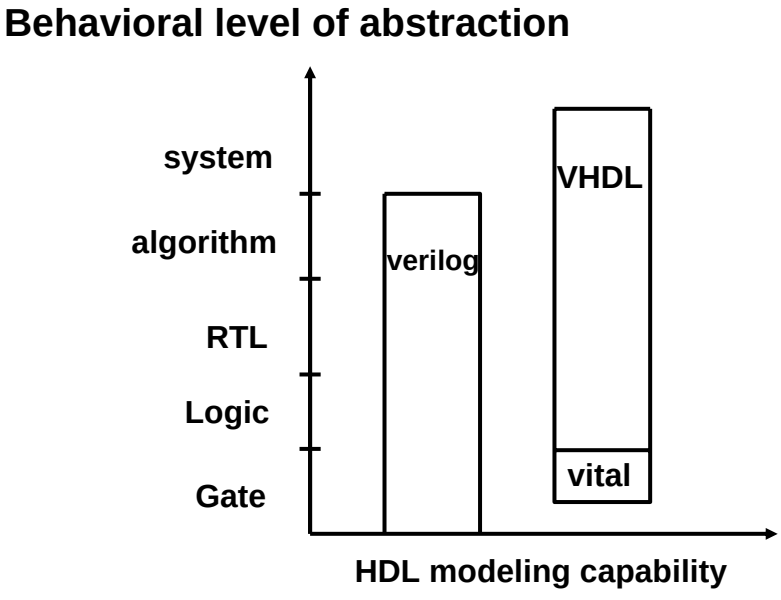
VHDL Advantages (1/2)

- **Clear definition of design requirements**
- **Efficiency in design cycle time**
- **Reuse of designs and packages**
- **Technology independent**
- **Easy analysis of various architecture/implementations**

VHDL Advantages (2/2)

- Design verification
- VHDL is recommended for government contracts
- VHDL commercial models are available for purchase
- VHDL is a documentation language
- VHDL is a simulation language

VHDL vs. Verilog (1/3)



VHDL vs. Verilog (2/3)

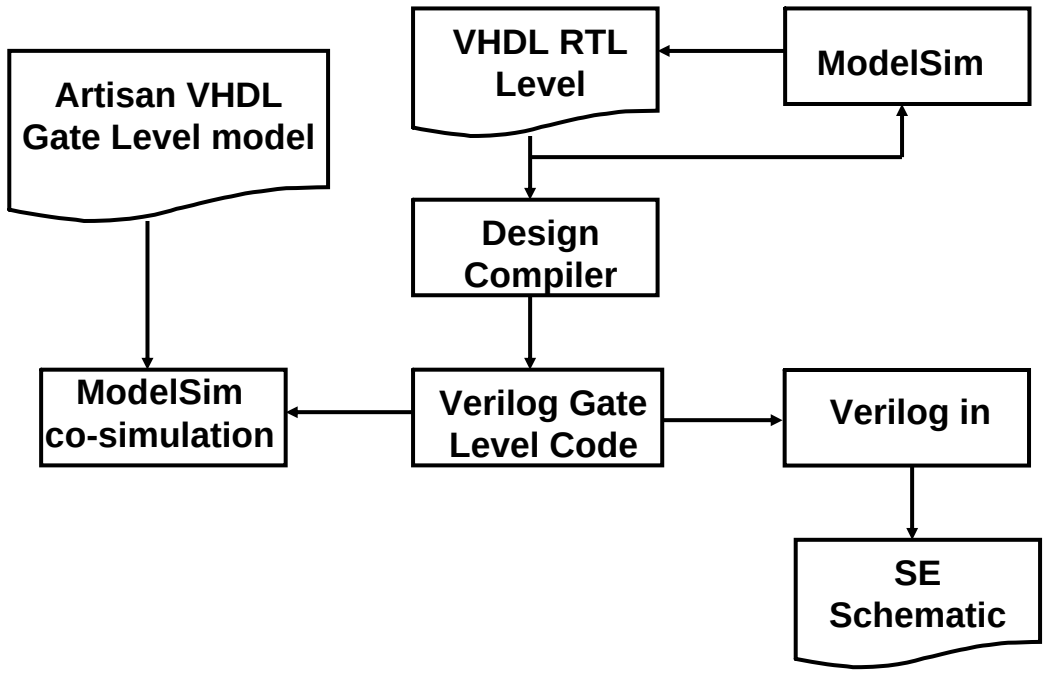
	VHDL	Verilog
Compilation	Compile	interpretative
Libraries	Yes	No
Resuability	Package	Include
Readability	ADA	C & ADA
Easy to Learn	Less intuitive	EASY

VHDL vs. Verilog (3/3)

- **USA - IBM, TI, AT&T, INTEL – VHDL**
- **USA - Silicon Valley – Verilog**
- **Europe – VHDL**
- **Japan – Verilog**
- **Korea – 70-80% VHDL**
- **Taiwan**

CIC Design Flow

CIC VHDL Design Flow



Top Down Design Model

- Partition design into several block.
- Define and verify the function operation for every block.
- Define and verify the timing operation for every block.
- Define and verify the operation for the whole of design.

Model Verification

- Functional verification have the following method:
  - the use of simulator command files.
  - the use of a VHDL testbench
- Timing verification can be verify via adding violation checks in the simulation model.

VHDL Fundamentals

- Structure
- Simulation
- Data Types
- Attribute
- Expression
- Hierarchy
- Generic

Structure

Comments

- **Starts with two adjacent hyphens --.**
  - extends up to the end of the line.

```
-- this line is comments
end;  -- processing of line is complete
------ the first two hyphens start the comment
```

Identifiers (1/2)

- **Identifiers are used as names but can't be reserved words.**
- **An identifier shall be any sequence of letters (a-z, A-Z) ,digits (0-9) and underscore characters (_).**
  - The first character of an identifier must be a letter.
  - The last character of an identifier shall be not a underscore characters.
  - can't have two adjacent underscore characters.

Identifiers (2/2)

- **Identifier shall be case insensitive.**

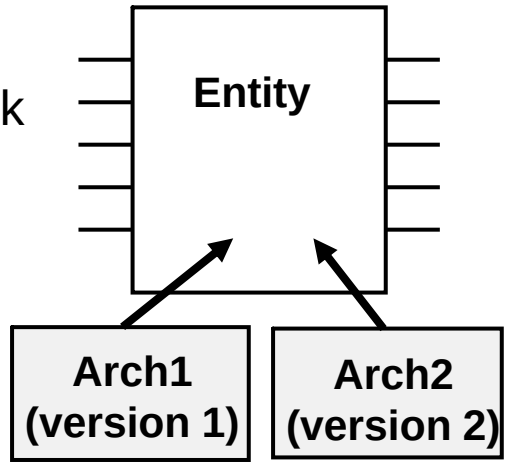
Synopsys Directives

- ***-- pragma translate_off & -- pragma translate_on*** **control the VHDL Compiler translation of VHDL code off & on.**

```
-- pragma translate_off
   Z <= string("don't translate");
-- pragma translate_on
   Z <= A and B;
```

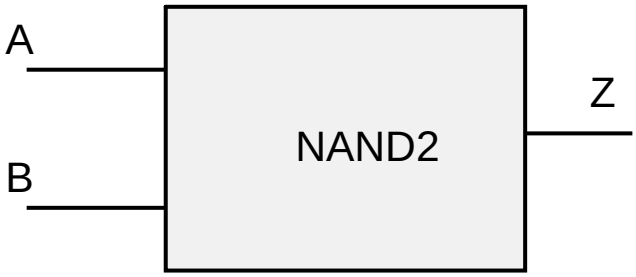

VHDL Five Main Design Unit

- **Entity**
  - I/O port spec. of the block
- **Architecture**
  - function of the block
  - can have more than one
- **Configuration**
  - choose a architecture to run simulation.
- **Package**
- **Package body**



Entity

```
Entity NAND2 is
        port ( A : in std_logic;
               B : in std_logic;
               Z : out std_logic);
end nand2;
```

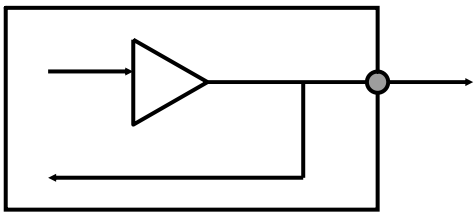


I/O Direction

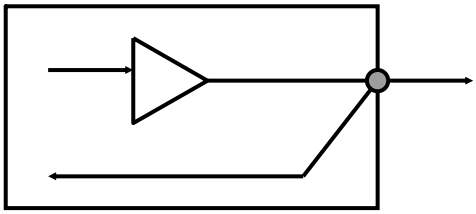
- **IN**
  - Connect with **OUT** or **INOUT**.
- **OUT**
  - Connect with **IN** or **INOUT**.
- **INOUT**
  - Connect with **IN** or **OUT**.
- **BUFFER**
  - Connect with **BUFFER** only.

Buffer vs. INOUT

- Buffer



- INOUT

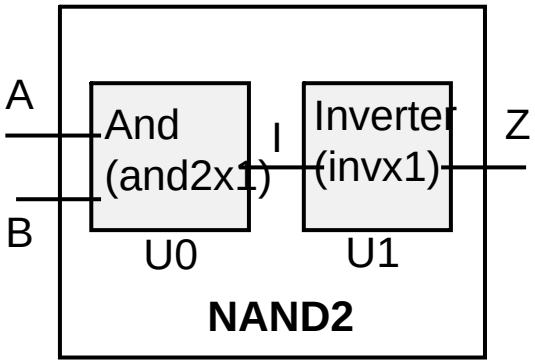


Basic Data Type

- Bit
- BIT_VECTOR (0 TO 7)
- STD_LOGIC
- STD_LOGIC_VECTOR (15 DOWNTO 0)

NAND2 architecture #1

```
Architecture STRUCT of NAND2 is
    signal I : std_logic;
begin
    U0 : and2x1 port map(A, B, I);
    U1 : invx1 port map(I, Z);
end STRUCT;
```



NAND2 architecture #2

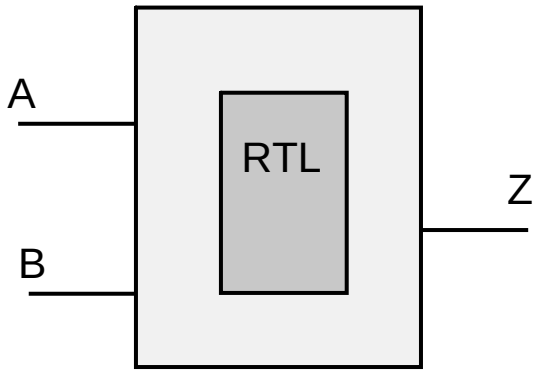
```
Architecture DATAFLOW of NAND2 is
begin

    Z<=A nand B;  -- concurrent statement

end DATAFLOW;
```

NAND2 architecture #3

```
Architecture RTL of NAND2 is
begin
   process(A,B)
   begin
     if (A='1') and (B='1') then  -- sequential statement
        Z<='0';
     else
        Z<='1';
     end if;
   end process;
end RTL;
```


Configuration

```
Configuration cfg_nand2 of NAND2 is
      for RTL
      end for;
end cfg_nand2;
```

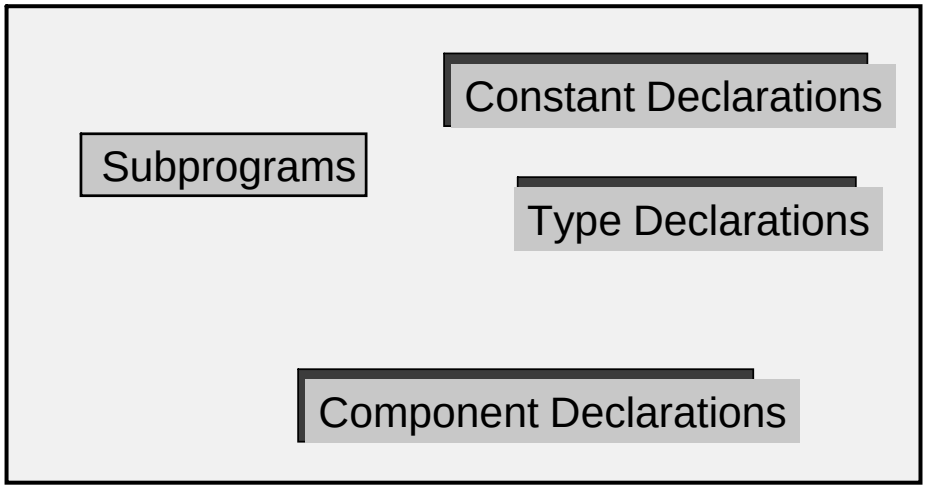


Recall

```
Entity a IS
END a;

ARCHITECTURE b Of a IS
BEGIN
END b;

CONFIGURATION c OF a IS
        FOR b
        END FOR;
END c;
```


Package and Package Bodies

- Package Declarations
- Package Bodies



Package

```
Package EXAMPLE is
      type BYTE is range 0 to 255;
      subtype NIBBLE is BYTE range 0 to 15;
      constant BYTE_FF: BYTE:=255;

      signal ADDEND: NIBBLE;
      component BYTE_ADDER
            port(A,B: in BYTE;
                  C: out BYTE;
                  OVERFLOW: out BOOLEAN);
      end component;
      function MY_FUNCTION(A: in BYTE) return BYTE;
end [EXAMPLE];
```


Recall

PACKAGE a IS
...
END [a];

PACKAGE BODY a IS
...
END [a];

Design Libraries (Content)

- **Primary Units**
  - Entity Declaration
  - Package Declaration
  - Configuration Declaration
- **Secondary Units**
  - Package Body
  - Architecture Body

Design Libraries (Category)

- **Working Library**
  - writable library (WORK)
- **Resource Library**
  - Vendor supply (Synopsys)
  - Standard library (IEEE)
  - Cell library VHDL model (Artisan,Compass)

Using Package

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.numeric_std.ALL;
use IEEE.std_logic_arith.ALL;
use IEEE.std_logic_unsigned.ALL;
use IEEE.std_logic_signed.ALL;

- - Library WORK;
use WORK.EXAMPLE.ALL;

entity...

architecture...
```


Work Library

- **ModelSim**
  - Map work library to other real folder.
- **Altera MaxPlusII and Xilinx**
  - The work library is the folder that put main vhdl program.

Data Type vs. Library

```
signal a: bit;
signal a: bit_vector(3 downto 0);
```

```
library IEEE;
use ieee.std_logic_1164.all;
....
signal a: std_logic;
signal a: std_logic_vector(0 to 3);
```

```
library IEEE;
use ieee.std_logic_arith.all;
....
signal a: signed(7 downto 0);
signal a: unsigned(7 downto 0);
```


Simulation

VHDL Simulation (ModelSim)

modelsim.ini

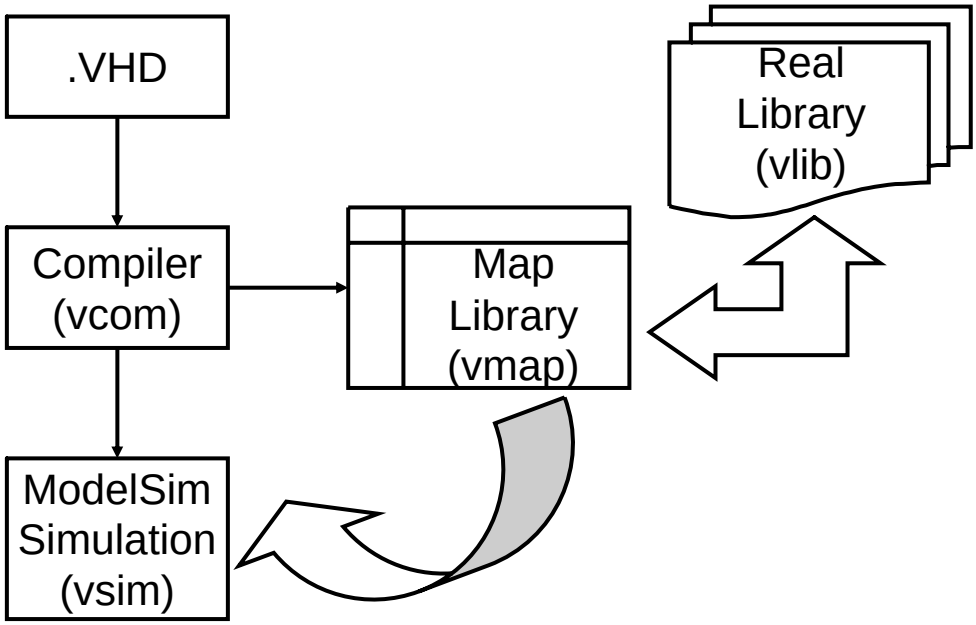
[Library]

others=$Model_TECH/../modelsim.ini

work=work

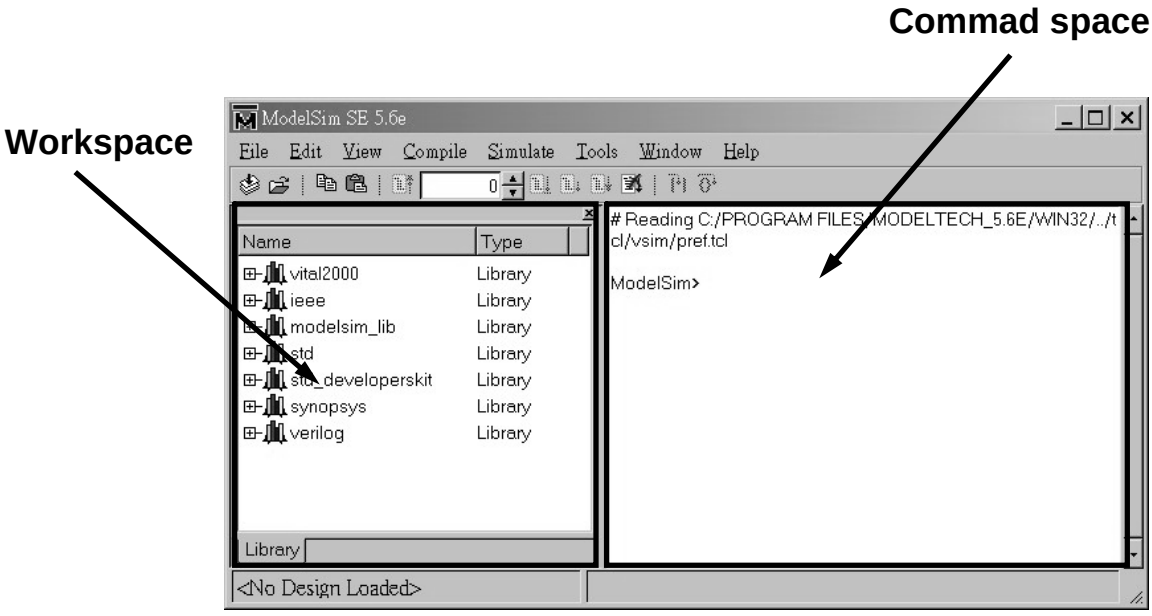
artisan_lib25=/cad2/lab/VHDL/modelsim/lib25

Simulate Flowchart



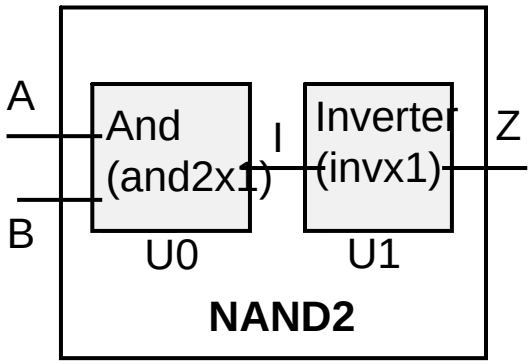
Invoke ModelSim

vsim&



NAND2 architecture #1

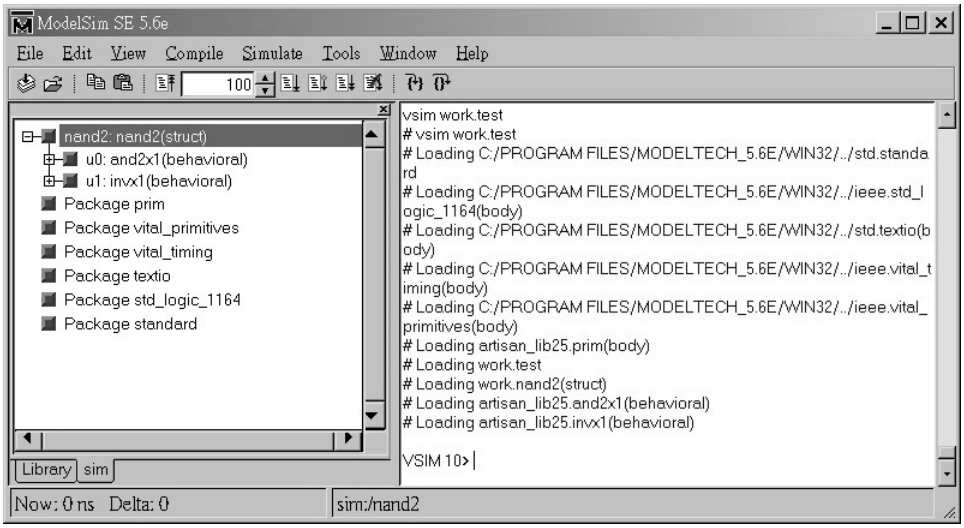
```
Library IEEE;
use ieee.std_logic_1164.all;
library artisan_lib25;
use artisan_lib25.prim.all;
entity nand2 is
    port(a,b: in std_logic;
         z: out std_logic);
end nand2;
architecture STRUCT of NAND2 is
   signal  I :   std_logic;
begin
   U0: and2x1 port map (A, B, I);
   U1: invx1 port map (I, Z);
end STRUCT;
```



Simulate Command

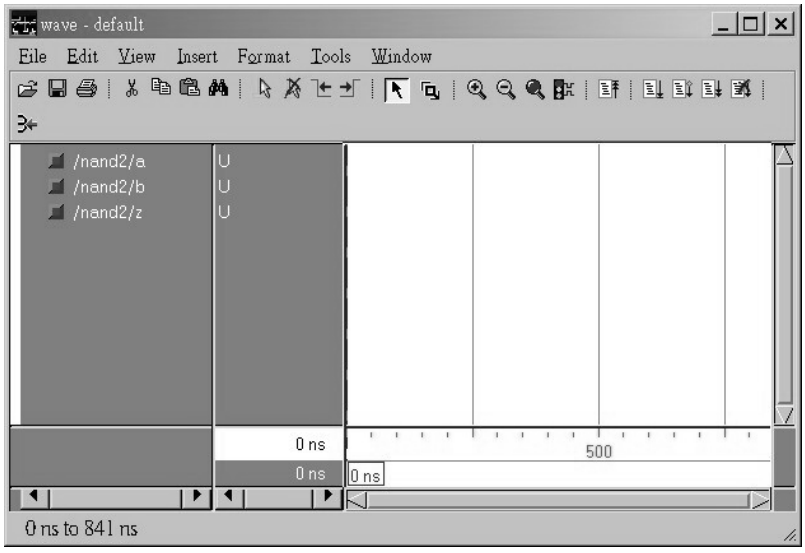
- **Create Library**
  - *vlib work*
- **Library mapping**
  - *vmap work work*
- **VHDL Compilation**
  - *vcom nand2.vhd*
- **Simulation**
  - *vsim test*

Simulation mode



Waveform viewer

add wave a b z

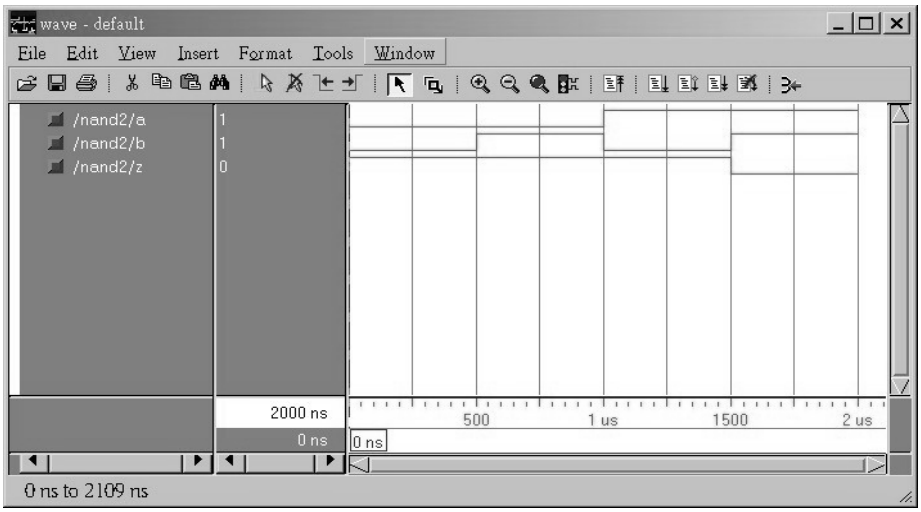


Stimuli

- simulation command
  - *force a 0*
  - *force b 0*
  - *run 500*
  - *force a 0*
  - *force b 1*
  - *run 500*
  - *force a 1*
  - *force b 0*
  - *run 500*
  - *force a 1*
  - *force b 1*
  - *run 500*

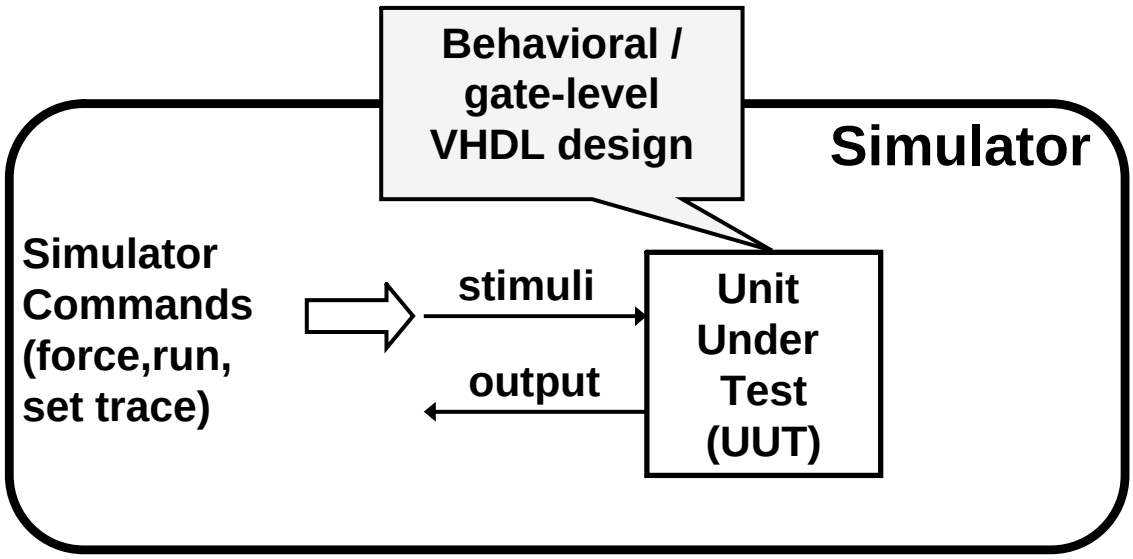
Check Result

- Write all command to ***nand2.do***
  - *do nand2.do*



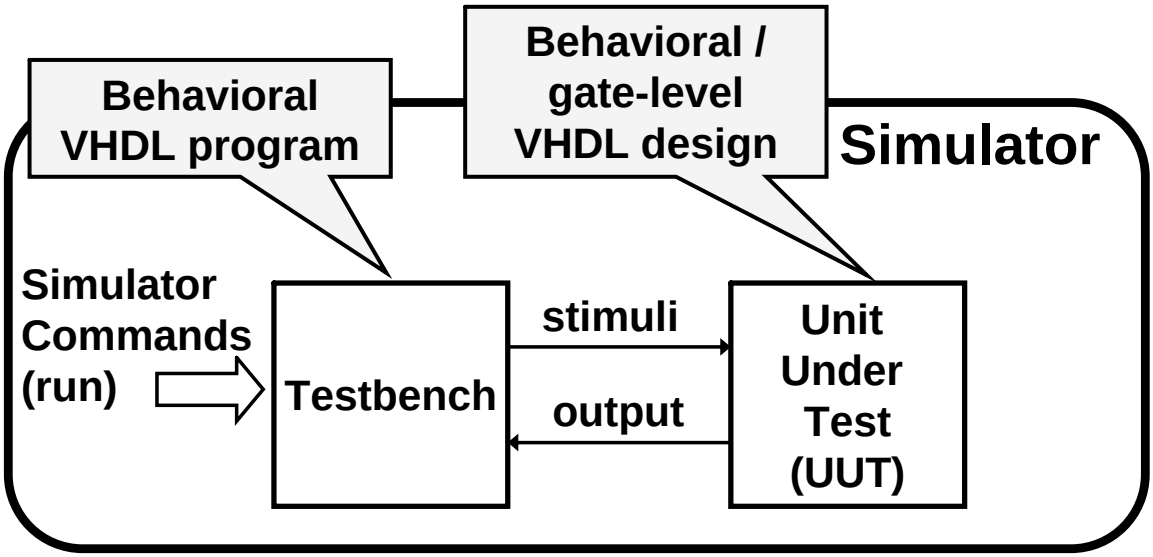
How to Verify Design ? (1/2)

Simulation without testbench



How to Verify Design ? (2/2)

Simulation with testbench



Data Types

Data Objects (1/2)

- **Signal**
  - Describe a real wire in the circuit.
- **Variable**
  - Holds any single value from the values of the specified type.
  - Often used to hold temporary values within a **process** or **subprogram**.
- **Constant**
  - Holds one specific value of the specified type.

Data Objects (2/2)

Signal

Declaration: **signal** data **: std_logic** :='0' **;**

Assignment: data <= '1';

Constant - to enhance readability

Declaration: **constant** bitwidth **: std_logic_vector (7 downto 0)** := "01101110"**;**

Variable - (process, procedure, function)

Declaration: **variable** data **: unsigned (0 to 2)** := "000"**;**

Assignment: data := "101";

Aliases

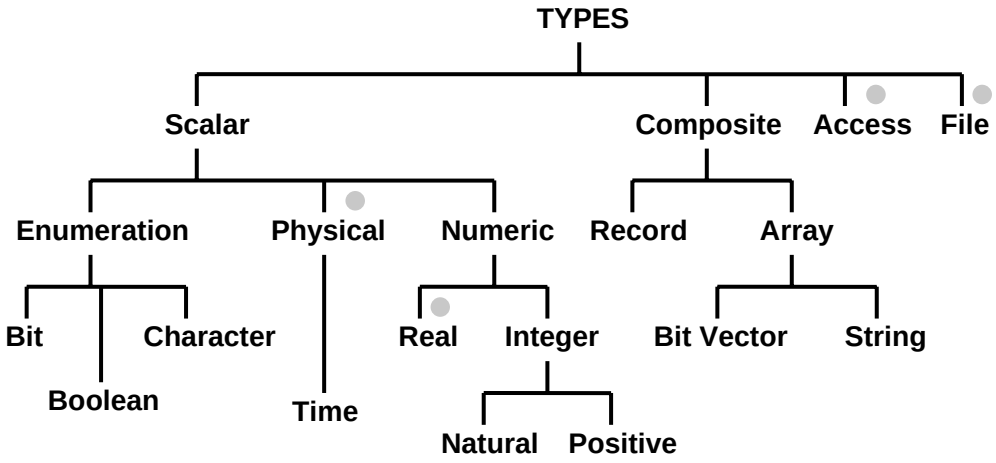
Declares an alternative name for an existing named object or part of an object.

Signal Addr **: bit_vector(31 downto 0);**
Alias Msb **: bit_vector(15 downto 0) is** Addr(31 downto16);
Alias Lsb **: bit_vector(15 downto 0) is** Addr(15 downto 0);

Variable TimeSeconds **: integer range 0 to 59;**
Alias Secs **: integer 0 to 59 is** TimeSeconds;

Data Types

Leaf is the default data type.



TYPE type_name IS ...

Scalar (1/7)

Enumeration

TYPE std_ulogic **IS (**'U','X','0','1','Z','W','L','H','-' **);**

-- only 'X','0','1','Z' can be synthesized.

TYPE Boolean **IS (**FALSE, TRUE**);**

TYPE Bit **IS (**'0','1'**);**

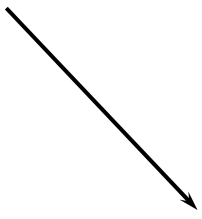
Scalar (2/7)

- **Enumeration-predefined**
  - character
  - bit
  - boolean
  - severity_level

Scalar (3/7)

- **Physical Type - only use in simulation**

```
TYPE Time IS RANGE 0 to 2000000000
UNITS
      fs;
      ps = 1000 fs;
      ns = 1000 ps;
      us = 1000 ns;
END UNITS;
```



integer range

Scalar (4/7)

Physical Type

```
TYPE capacitance IS RANGE 0 to 2000000000
UNITS
        ff;
        pf=1000 ff;
        nf=1000 pf;
        uf=1000 nf;
END UNITS;
```


Scalar (5/7)

Physical-predefined

- Time

Scalar (6/7)

Numeric Type

TYPE Byte **IS RANGE** 0 **TO** 255**;**

TYPE Register **IS RANGE** 10#54.6# **TO** 10#67.3# **;**

<base>#<value>#e<value>

$$10\#3\#E2 = (3)_{10} \times 10^{(2)_{10}} = 300$$

Scalar (7/7)

Numeric Type-Predefined

- integer
- real

Subtypes (1/2)

for Enumeration **use in the FSM's state**

TYPE *Colors* IS (red, yellow, blue, green, black**);**
SUBTYPE Primary **IS *Colors* RANGE *red* TO *blue*;**

for Numeric

TYPE *Integer* IS RANGE -2147483647 **TO** +2147483647**;**

SUBTYPE Natural **IS *Integer* RANGE *0* TO *Integer'HIGH* ;**

Subtypes (2/2)

- **Subtype-predefined**
  - Natural
  - Positive

Composite (1/5)

Record Type

```
Package example Is
   TYPE FloatPointType IS
     RECORD
        Sign: std_logic;
        Exponent: unsigned(6 downto 0);
        Fraction: unsigned(23 downto 0);
   END RECORD;
end;
```

Composite (2/5)

Record Type

```
Use work.example.all;

Architecture arch of xxx is
    signal a: FloatPointType;
begin
    a.sign<='1';
    a.exponent<="1011010";
    a.fraction<=(others=>'0');
end arch;
```

Composite (3/5)

Record vs. Array

```
Architecture arch of xxx is
    signal a: std_logic_vector(31 downto 0);
begin
    a(31)<='1';
    a(30 downto 24)<="1011010";
    a(23 downto 0)<=(others=>'0');
end arch;
```


Composite (4/5)

Array Type (in architecture, process, function, procedure)

- Constrained
  - Declare array type

```
TYPE Data_Bus is ARRAY (0 TO 31) OF Bit;
TYPE Dim_2 is ARRAY (0 TO 7, 0 TO 3) OF Bit;
```

  - Use array type

```
signal a : Data_Bus;
signal b : Dim_2;
```

Composite (5/5)

Array Type (in architecture, process, function, procedure)

- Unconstrained
  - Declare array type

TYPE String **is ARRAY (**Positive **Range <>) OF** Character**;**
TYPE Bit_Vector **is ARRAY (**Natural **Range<>) OF** Bit**;**

  - Use array type

signal c **:** String(3 downto 1)**;**

signal d **:** Bit_Vector(3 downto 0)**;**

Array of 2 - Dimension

Model 1

```
type dim1 is array (7 downto 0) of bit_vector(7 downto 0);
signal a : dim1;
Assignment:
        a(0)(7 downto 0) <= “00000000”;
```

Model 2

```
type dim2 is array (7 downto 0,7 downto 0) of bit;
signal b : dim2;
Assignment:
        b(0,7 downto 0) <= “00000000”;    -- Error
        b(0,0) <= ‘0’;
        b(0,1) <= ‘0’;
```

Access (1/2)

Only simulation

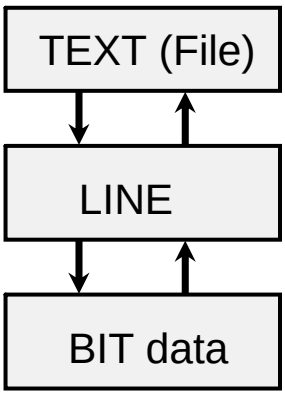
```
Package example Is
    Type data_rec;          -- Incomplete type
    Type data_ptr is Access data_rec;

    TYPE data_rec IS
    RECORD
        data : std_logic_vector(7 downto 0);
        nxt  : data_ptr;
    END RECORD;
end;
```


Access (2/2)

```
PROCESS (CLK)
    VARIABLE head: data_ptr := NULL;
    VARIABLE temp: data_ptr := NULL;
BEGIN
    IF CLK'EVENT AND CLK = '1' THEN
       IF (RW_DIR = '0') THEN    --Write Mode
          temp := NEW data_rec;
          temp.data := D_IN;
          temp.nxt := head;
          head := temp;
       ELSIF (RW_DIR = '1') THEN
          D_OUT <= head.data;
          temp := head;
          head := temp.nxt;
          DEALLOCATE(temp);  -- delete temp
       END IF;
    END IF;
END PROCESS;
```

TEXT I/O

```
USE std.textio.all;
USE ieee.std_logic_textio.all;
FILE input_file : TEXT IS IN "/dsk/vhdl/in.data";
FILE output_file : TEXT IS OUT "/dsk/vhdl/out.data";
VARIABLE l1,l2 : LINE;
VARIABLE test : std_logic_vector(7 downto 0);
```

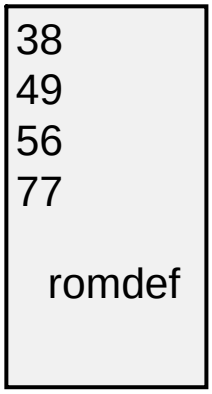


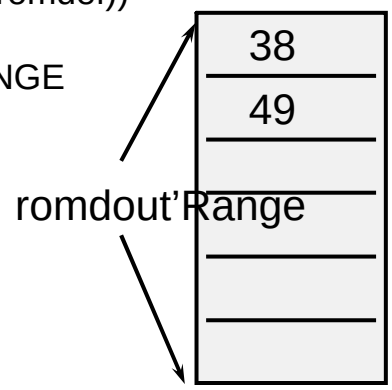

```
WHILE NOT ( ENDFILE(input_file) )
  LOOP
  READLINE(input_file, l1);
  READ(l1, test);
  WRITE(l2, test);
  WRITELINE(output_file, l2);
  END LOOP;
```


TEXT I/O - Memory Init

TYPE mem_data IS ARRAY (integer RANGE <>) OF integer;

```
PROCEDURE rom_init( VARIABLE romdef: IN TEXT;
                    VARIABLE romdout : OUT mem_data) IS
  VARIABLE l: line;
BEGIN
    lp1: WHILE NOT (ENDFILE(romdef))
    LOOP
      lp2: FOR j IN romdout'RANGE
      LOOP
        READLINE(romdef,l);
        READ(l,romdout(j));
      END LOOP lp2;
    END LOOP lp1;
END rom_init;
```





TEXT I/O -- VHDL'93

USE std.textio.ALL;

FILE input_file **: TEXT open Read_Mode IS** "in.data"**;**

FILE output_file **: TEXT open Write_Mode IS** "out.data"**;**

FILE output_file **: TEXT open Append_Mode IS** "out.data"**;**

Attribute

Type Related (1/3)

- **T'HIGH**
  - week'high=SAT, count'high=63, byte'high=7
- **T'LOW**
  - week'low=SUN, count'low=0, byte'low=0

```
TYPE week IS (SUN, MON, TUE, WED, THU, FRI, SAT);
TYPE count IS INTEGER RANGE 0 to 63;
TYPE byte IS ARRAY (7 downto 0) of STD_LOGIC;
```

Type Related (2/3)

- **T'LEFT**
  - week'left=SUN, count'left=0, byte'left=7
- **T'RIGHT**
  - week'right=SAT, count'right=63, byte'right=0

```
TYPE week IS (SUN, MON, TUE, WED, THU, FRI, SAT);
TYPE count IS INTEGER RANGE 0 to 63;
TYPE byte IS ARRAY (7 downto 0) of STD_LOGIC;
```


Type Related (3/3)

```
Signal A : unsigned ( 3 downto 0);
Signal B : unsigned ( 0 to 3);
Signal Y : unsigned (3 downt 0);

Y(0)<=A(A'left) and B(B'left);
  -- Y(0)<=A(3) and B(0);
Y(1)<=A(A'right) or B(B'right);
  -- Y(0)<=A(0) or B(3);
Y(2)<=A(A'high) nand B(B'high);
  -- Y(0)<=A(3) nand B(3);
Y(3)<=A(A'low) nor B(B'low);
  -- Y(0)<=A(0) nor B(0);
```

Array Related (1/3)

- **A'LENGTH**
  - nibble'length=4, address'length=13
- **A'LENGTH(i)**
  - data'length(1)=2, data'length(2)=4

```
TYPE nibble IS ARRAY (3 downto 0) of STD_LOGIC;
TYPE address IS ARRAY (17 downto 5) of BIT;
TYPE data IS ARRAY (1 downto 0 , 3 dwonto 0) of BIT;
```


Array Related (2/3)

- **A'RANGE**
  - nibble'range= 3 downto 0
- **A'REVERSE_RANGE**
  - address'reverse_range = 5 to 17

```
TYPE nibble IS ARRAY (3 downto 0) of STD_LOGIC;
TYPE address IS ARRAY (17 downto 5) of BIT;
```

Array Related (3/3)

Signal A : unsigned (3 downto 0);
Signal B : unsigned (0 to 3);

if (***A'length = B'length***) then

-- if (4 = 4) then

---

for N in ***A'range*** loop

-- for N in 3 downto 0 loop

---

for N in ***A'reverse_range*** loop

-- for N in 0 to 3 loop

Signal Related

- **S'EVENT**
- **S'STABLE**
- **S'ACTIVE**
  - can't be synthesized
- **S'TRANSACTION**
  - can't be synthesized

S'event vs. S'stable (1/2)

- **S'event**
  - Function returning a **Boolean - true** which identifies if signal S has a new value assigned onto this signal.
- **S'stable**
  - Function returning a **Boolean - false** which identifies if signal S has a new value assigned onto this signal.

S'event vs. S'stable (2/2)

```
-- positive-edge-trigger

if (clk'event and clk='1')

          | |

if ( not clk2'stable and clk2='1')
```

S'active

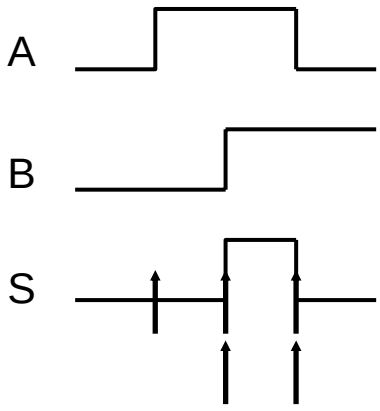
S'active

- Function returning a **Boolean - True** which identifies if signal S had a new assignment made onto it. (whether the value of the assignment is the **SAME** or **DIFFERENT**.)

S'event vs. S'active

S'event vs. *S'active*

S <= A and B;



S'transaction

S'transaction

- **Implicit signal** of type **bit** which is created for signal S when it S'transaction is used in the code.
- This signal toggles in value (between '0' and '1') when signal S had a new assignment made onto it (whether the value of the assignment is the **SAME** or **DIFFERENT.**

User Defined Attribute (1/3)

Declare an "Attribute".

Specify this attribute to the identifiers.

Attribute Delay_time **: time;**
Attribute Delay_time of Clock1, Clock2 **: signal is 2.3 ns;**

User Defined Attribute (2/3)

```
Type Com_loc is
Record
     M, N: integer;
End Record;
Attribute place : Com_loc;
Attribute place of AND_2 : Label is (40, 35);
signal perimeter : integer;
...

AND_2 : And_comp port map (a, b, c);
Perimeter <= 2*(AND_2'place.M + AND_2'place.Y);
```


User Defined Attribute (3/3)

- **Can provide a means of specifying particular enumerated values.**

```
Type Color is (Red, Orange, Yellow, Green);
Attribute Encoding : string;
Attribute Encoding of Color : type is "00 11 10 01";
```

Expression

Operands

- Literal Operands
- Identifier
- Aggregate
- Function call operands
- Qualified Expression Operands
- Type Conversion Operands
- Record & Record Element Operands

Literal Operands

String Literals

"ABC"
B"1010" -- Binary
X"9FDE" -- Hexadecimal
O"576" -- Octal

Numeric Literals

314159
8#57#
16#9FDE#

Enumeration Literals

TRUE
FALSE
'a'
'1'
'0'

Common Question

```
Architecture arch of en is
   Signal k: bit_vector(15 downto 0);
   Signal g: std_logic_vector(15 downto 0);
begin
   k<=X"9FDE";
   g<=X"9FDE";  -- error
   g<=to_stdlogicvector(X"9FDE");
end arch;
```

Identifier Operands

```
port(A,B,C,D: in std_logic;
       Y1,Y2: out std_logic);

function fun_a (A: unsigned(7 downto 0))
                                  return std_logic is
    variable Result: std_logic;
begin
          ...
end fun_a;
```


Aggregate Operands (1/4)

- Aggregate data to an ARRAY.

```
Signal a,b,c,d: std_logic;
signal Arry: std_logic(3 downto 0);
...
Arry <= ( a , b , c , d );
Arry <= ( 3=>a , 2=>b , 1=>c , 0=>d );
```

Aggregate Operands (2/4)

Aggregate data to a RECORD

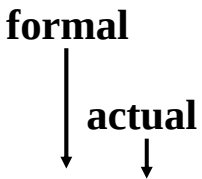
```
type FloatPointType is
   record
      Sign: std_logic;
      Exp: unsigned(6 downto 0);
      Frac: unsigned(23 downto 0);
end record;
Signal Rec : FloatPointType;
```

```
Rec <= ( '1' , "1010" , "1101010" );
Rec <= ( sign=>'1' , exp=>"1010" , frac=>"1101010" );
```


Association Lists

Named Association - *both the formal and actual are listed*.

formal => actual




```
Arry<=(3=>a,2=>b,1=>c,0=>d);
```

Positional Association - *only the actual.*

```
Arry<=(a,b,c,d);
```

Aggregate Operands (3/4)

correct method：

```
(=>,=>,=>)  <=   AR ;

or

AR  <=  (=>,=>,=>);
```


Aggregate Operands (4/4)

wrong method：

```
signal a: std_logic;
signal b: std_logic_vector(3 downto 0);
signal c: std_logic_vector(4 downto 0);
….
C<=(a,b);    --Error
```

Function call Operands

```
Function fn1(f1:std_logic;
             f2:std_logic;
             f3:std_logic;
             f4:std_logic) return std_logic;
```

```
Y1 <= Fn1(a1,a2,a3,a4) or B1 or B2;

Y2 <= B1 or B2 or
      Fn1(F3=>A3,F4=>A4,F1=>A1,F2=>A2);

Y3 <= B1 or Fn1(A1,A2,F4=>A4,F3=>A3) or B2;
```


Index and Slice (ARRAY)

```
y(2 downto 0) <= a(0 to 2);

y(3) <= a(3) and b(3);

y(5 downto 4) <= (a(5) and b(5))&(a(4) or b(4));

y(8 downto 6) <= b(2 downto 0);

y(11 downto 9) <= c;
```

Qualified Expression Operands

- **type_name'(expression)**

```
type R_1 is range 0 to 10;
type R_2 is range 0 to 20;

function FUNC(A: R_1) return BIT;
function FUNC(A: R_2) return BIT;

FUNC(5);  -- ambiguous

FUNC(R_1'(5)); -- unambiguous
```


Type Conversion Operands

- **type_name(expression)**
- **integer types**
- **similar array types**
  - same length
  - convertible or identical element types
- **Enumerated types are** *not convertible*

linteger Types

```
type INT_1 is range 0 to 10;
type INT_2 is range 0 to 20;
signal S_INT:   INT_1;
signal T_INT:   INT_2;
 ...
T_INT <= INT_2(S_INT);  -- Ok
S_INT <= INT_1(T_INT);  -- Error
```


Array Types

```
subtype MY_BIT_VECTOR is BIT_VECTOR (1 to 10);
type BIT_ARRAY_10 is array(11 to 20) of BIT;
signal S_BIT_VEC: MY_BIT_VECTOR;
 ...
BIT_ARRAY_10(S_BIT_VEC);  -- OK
```

Wrong Conversion

type INT_1 is range 0 to 10;
type ARRAY_1 is array (1 to 10) of INT_1;
subtype MY_BIT_VECTOR is BIT_VECTOR (1 to 10);
type BIT_ARRAY_20 is array(0 to 20) of BIT;
signal S_BIT_VEC: MY_BIT_VECTOR;
signal S_BIT: BIT;

wrong conversion：

BOOLEAN(S_BIT); -- enumerated type
INT_1(S_BIT);
BIT_ARRAY_20(S_BIT_VEC); -- length not equal
ARRAY_1(S_BIT_VEC); -- element types not convertible

Record & Record Element Operands

```
signal M,C: R1_Type;
signal Y: R2_Type;
...
if ( C.I /= M.I ) then
   Y.I <= M.I - C.I;
else
   Y.I <= 0;
end if;
```

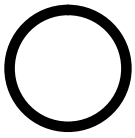
Operators

- **Logical operator(and or nand nor xor xnor not)**
- **relational operator(= /= < <= > >=)**
- **shift operator (sll srl sla sra rol ror)**
  - Only be supported by VHDL'93
- **adding operator (+ - &)**
- **sign (+ -)**
- **multiplying operator (* / mod rem)**
- **miscellaneous operator (** abs)**

Logical Operators

- **nand vs. and**

Z<= a and b and c;



Z<= a nand b nand c;

Z<= (a nand b) nand c;

Z<= a nand (b nand c);

Relational Operators

```
Process(A)
begin
  if((A<=“00000110”) and
    (A>=“00000001”)) then -- Comparators
       Q<=‘1’;
  else
       Q<=‘0’;
  end if;
end process;
```

= /=
vs.
<= >= > <

```
Process(A)  -- Simple Logic
begin
  if((A(7 downto 3)=“00000”) and
    (A(2 downto 0) /= “111”) and
    (A(2 downto 0) /= “000”)) then
       Q<=‘1’;
  else
       Q<=‘0’;
  end if;
end process;
```


Concatenation Operators (1/3)

- concatenation &

```
signal a, b: bit;
signal c: bit_vector (7 downto 0);

c(7 downto 6) <= a & b;
```

Concatenate Operators (2/3)

- **The result of Concatenate Operator is** unconstrained Array

```
Case a&b is       -- ERROR
        when “00” => Q<=C;
        when “01” => Q<=A;
        when others => Q<=‘0’;
end case;
```


Concatenate Operators (3/3)

```
Case bit_vector’(a, b) is
        when “00” => Q<=C;
        when “01” => Q<=A;
        when others => Q<=‘0’;
end case;
```

```
Subtype TTT is bit_vector(1 to 2);
...
Case TTT’(a&b) is
        when “00” => Q<=C;
        when “01” => Q<=A;
        when others => Q<=‘0’;
end case;
```

Multiplying Operators (1/2)

- **for synopsys synthesis**
  - Supports the multiplying operators for all **integer types**.
  - Division, Remainder and Modulus
    - The right-hand operand must be a computable power of 2 and cannot be negative.
- **There are more restrictions of synthesis in the other synthesis tools excepts Synopsys.**

Multiplying Operators (2/2)

- **REM -Sign of rem operation is sign of the “left” operand**
- **MOD -Sign of rem operation is sign of the “right” operand**

J=K*N+(J mod K)

J	K	J REM K	J MOD K
4	5	4	4
-4	5	-4	1
4	-5	4	-1
-4	-5	-4	-4

Miscellaneous Operators

for synopsys synthesis

- Supports the “abs” and “**” operators for all integer types.
- When you're using “**”, the left operand must be the computable value 2.

```
signal A, B: INTEGER range -8 to 7;
signal C: INTEGER range 0 to 15;
signal D: INTEGER range 0 to 3;
...
A <= abs(B);
C <= 2 ** D;
```


Common use package (1/4)

std_logic_1164

```
Library IEEE;
USE IEEE.std_logic_1164.ALL;

ARCHITECTURE ARCH of XXX IS
  signal a: STD_LOGIC;
  signal b: STD_LOGIC_VECTOR(3 downto 0);
begin
     b<=(a,a,a,a);
end arch;
```

Common use package (2/4)

std_logic_arith

```
Library ieee;
Use ieee.std_logic_arith.all;

architecture arch of xxx is
    signal a:unsigned(3 downto 0);
    signal b: signed(3 downto 0);
begin
        a <= “1010”;        --  10
        b <= “1010”;        --   -6
end arch;
```


Common use package (3/4)

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
--use ieee.std_logic_unsigned.all;

Architecture arch of xxx is
  signal a,b,c: unsigned (3 downto 0);
  signal x,y,z: std_logic_vector(3 downto 0);
begin
  c<=a+b;
  z<=x+y;  - - error
end arch;
```

Common use package (4/4)

- IEEE.std_logic.1164_ALL;
- IEEE.numeric_std.ALL;
- IEEE.std_logic_arith.ALL;
- IEEE.std_logic_unsigned.ALL;
- IEEE.std_logic_signed.ALL;
- IEEE.math_real.ALL;
- IEEE.math_complex.ALL;

	std_logic_vector	unsigned
Declaration		
Function		Declaration and Function

Shift Operators (1/2)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_ARITH.ALL;
variable count: unsigned (1 downto 0);
variable u,s,: unsigned(7 downto 0);
variable z: unsigned(7 downto 0);
u:= “01101011”;
s:= “11101011”;
...
count := conv_unsigned(arg=>3, size=>2);
z := shl(u,count);   -- “01011000”
z := shl(s,count);   -- “01011000”
z := shr(u,count);   -- “00001101”
z := shr(s,count);   -- “00011101”
```

1	1

“11”

Shift Operators (2/2)

```
LIBRARY IEEE;
USE IEEE.Numeric_STD.ALL;

variable a,z: unsigned(4 downto 0);
constant count: integer := 2;
...
a := "11100";

z := sll(a,count);    -- "10000"
z := slr(a,count);    -- "00111"
z := rol(a,count);    -- "10011"
z := ror(a,count);    -- "00111"
```


"sla" and "sra" -- VHDL'93

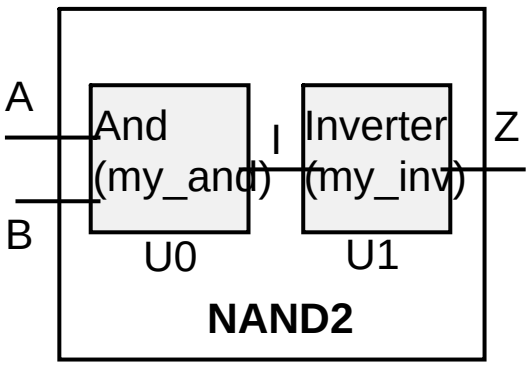
```
variable a : unsigned(4 downto 0);
variable y1,y2 : unsigned(4 downto 0);
constant count: integer := 2;
...
a := "10101";

y1 := sla(a,count);   -- Error
y1 := a sla count;    -- "10111"
y2 := a sra count;    -- "11101"
```

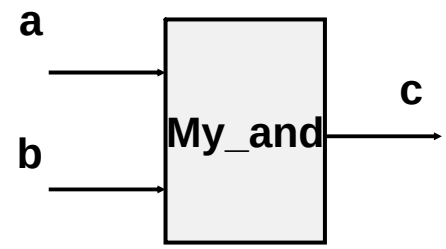
Online Help Manual

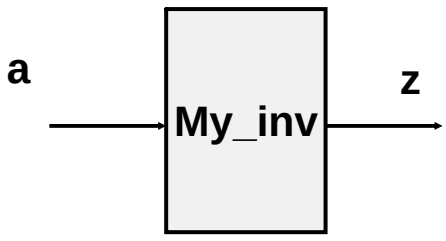
- **Synthesis Reference**
  - %acroread /usr/synopsys/sold/cur/top.pdf
  - Design Compiler
    - HDL Compiler for VHDL Reference Manual
- **Package Reference**
  - /usr/mentor/fpgadv50_ss/Modeltech/vhdl_src
    - ./ieee
    - ./synopsys

Hierarchy

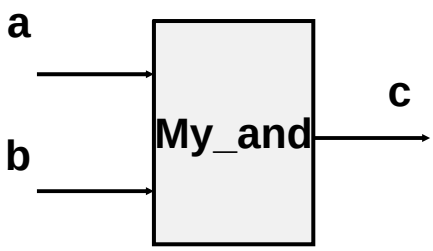
Modeling Nand2





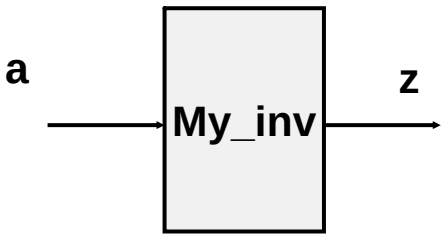


My_and



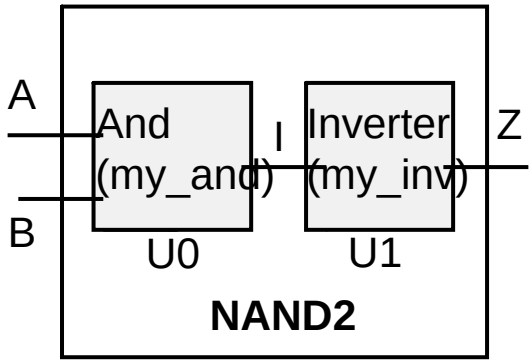

```
entity my_and is
        port(A,B: in std_logic;
                C: out std_logic);
end my_and;
architecture arch of my_and is
begin
   c<= a and b;
end arch;
configuration cfg_and of my_and is
          for arch
          end for;
end cfg_and;
```


My_inv




```
entity my_inv is
     port(a:in std_logic;
             z: out std_logic);
end my_inv;
architecture arch of my_and is
begin
   z<= not a;
end arch;
configuration cfg_inv of my_inv is
          for arch
          end for;
end cfg_inv;
```

NAND2




```
Entity my_nand is
  port(a,b: in std_logic;
       z: out std_logic);
end my_nand;
```

```
architecture arch of my_nand is
   component my_and
      port(A,B: in std_logic;
           C: out std_logic);
   end component;
   component my_inv
      port(a:in std_logic;
           z: out std_logic);
   end component;
   signal I: std_logic;
begin
    U1: my_and port map (a,b,I);
    U2: my_inv port map (I,z);
end arch;
```


Component

- If component and entity have the same name, VHDL simulator will find the corresponding entity automatically.
- If component and entity have the different name, you must designate the corresponding entity oneself.
  - But will have warning during compiling.
    - Can ignore.

Use Entity

```
CONFIGURATION CFG1_NAND OF MY_NAND IS
  FOR ARCH
    FOR U1: MY_AND USE ENTITY WORK.MY_AND(ARCH);
    END FOR;
    FOR ALL: MY_INV USE ENTITY WORK.MY_INV(ARCH);
    END FOR;
  END FOR;
END CFG1_NAND;
```


Use Configuration

```
CONFIGURATION CFG2_NAND OF MY_NAND IS
  FOR ARCH
    FOR U1: MY_AND USE CONFIGURATION WORK.CFG_AND;
    END FOR;
    FOR ALL: MY_INV USE CONFIGURATION WORK.CFG_INV;
    END FOR;
  END FOR;
END CFG2_NAND;
```

Component and Entity are Different name

```
entity yy is
port(x,y: in std_logic;
       z: out std_logic);
end yy;
```

```
ARCHITECTURE xx OF yy IS
    component ND2
      port(A,B: in std_logic;
               C: out std_logic);
    end component;
...
signal r,s: BIT;
...
BEGIN
   U1: ND2 port map(X,Y,r);
   U2: ND2 port map(X,Y,s);
   U3: ND2 port map(r,s,Z);
END xx;
```


Use Entity to Designate

```
Configuration conf of yy is
    for xx
      for u1: ND2 use entity work.my_nand(arch);
      end for;
      for others: nd2 use entity work.my_nand(arch);
      end for;
    end for;
end conf;
```

Use Configuration to Designate

```
Configuration conf2 of yy is
   for xx
      for ALL: ND2 use configuration work.cfg1_nand;
      end for;
   end for;
end conf;
```


Solving the Warning during Compiling

```
ARCHITECTURE xx OF yy IS
   component ND2
      port(A,B: in std_logic;
           C: out std_logic);
   end component;
   for ALL: ND2 use entity work.my_nand(arch);

begin
...
end xx;
```

Connect to VDD and GND?

```
architecture STRUCT of NAND2 is
    signal  I:   std_logic;
    signal GND:std_logic;
    signal VDD: std_logic;
  begin
    GND<='0'; VDD<='1';
    U0: my_and port map (a=>VDD, b=>GND , c=>I);
    U1: my_inv port map (a=>I, z=>Z);
  end STRUCT;
```

port map(a1=>'1', a2=>'0', zn=>I)

VHDL'93 only

Generic

Generic (1/2)

Entity

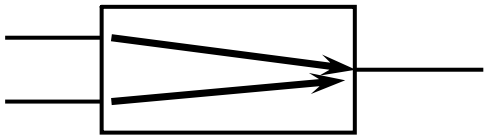
```
ENTITY MY_AND2 IS
    GENERIC( tp : time :=5 ns );
    PORT (q: OUT std_logic;
          a,b:IN std_logic);
END MY_AND2;
```

timing propagation

Generic (2/2)

```
...
ENTITY and2 IS
    GENERIC(tp: time:= 5 ns);
    PORT(q: OUT std_logic;
         a,b: IN std_logic);
END and2;
...
PROCESS(a,b)
   VARIABLE newstate: std_logic;
BEGIN
   newstate :=  a AND b;
   q <= newstate after tp;
END PROCESS;
```

timing propagation



After waiting 5 ns, newstate was assigned to q.

Overriding the defaults of Generic

- **in architecture**
  - component declaration
  - component instantiations
- **in configuration**
  - parents
  - children

Generic (component declaration)

Component Declaration

```
COMPONENT my_and
    GENERIC(tp:time:=3 ns);
    PORT (q: OUT std_logic;
          a,b:IN std_logic);
END COMPONENT;
```

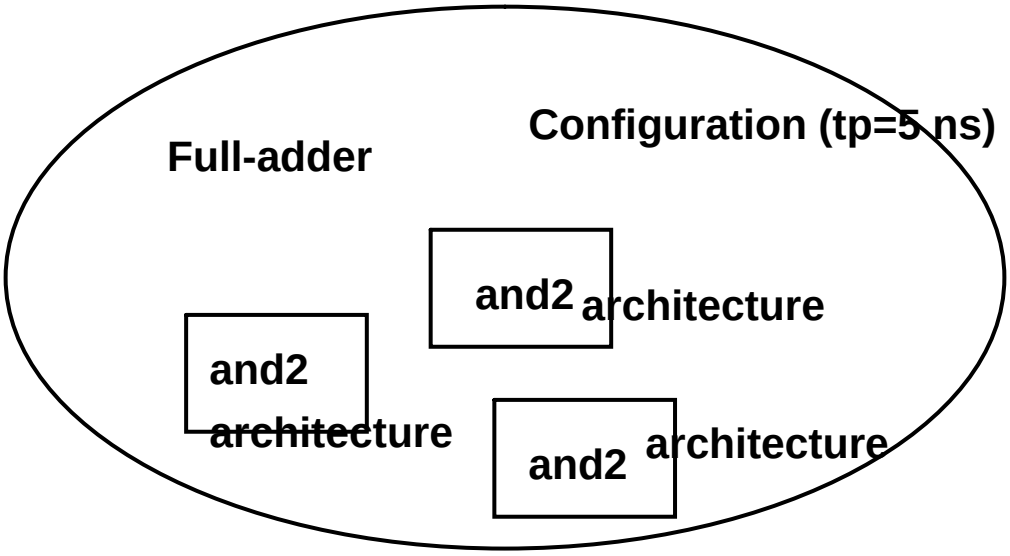
timing propagation

Generic (component instantiations)

Component Instantiation

```
U4: my_not GENERIC MAP(7 ns) PORT MAP(IN,Y);

U5: my_and  GENERIC MAP(tp=>7 ns)
                PORT MAP(a=>IN1, q=>IN3, b=>CIN);
```

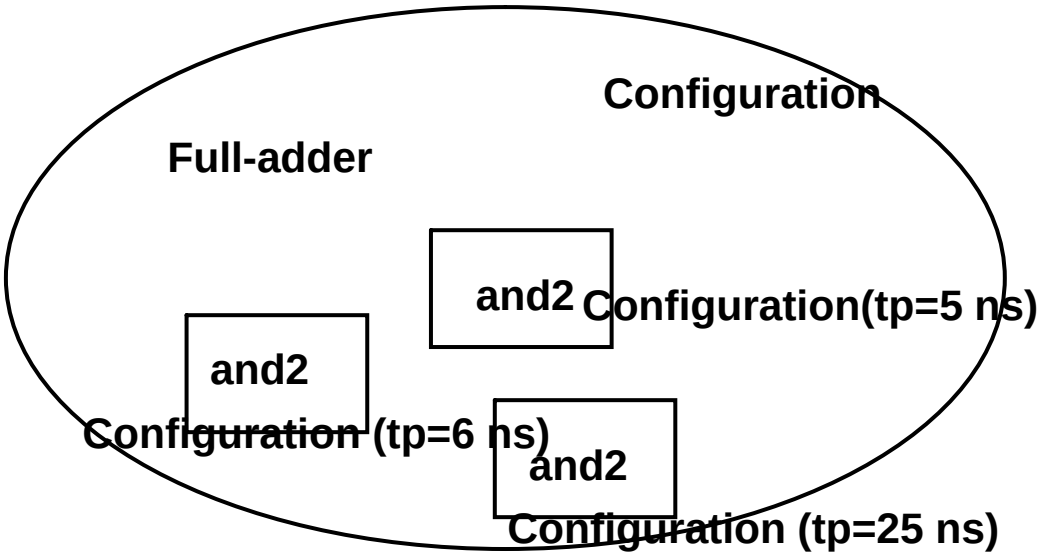
Generic (children) (1/2)



Generic (children) (2/2)

```
CONFIGURATION fulladd_config OF fulladd IS
   FOR structure
      ...
      FOR u0: my_and use entity work.my_and(arch)
            GENERIC MAP (tp=>3 ns);
      END FOR;
      ...
   END FOR;
END fulladd_config4;
```

Generic (parents) (1/3)



Generic (parents) (2/3)

Configuration of and2

```
CONFIGURATION 74F08_cfg OF my_and IS
    FOR arch
      GENERIC MAP(tp=> 2.5 ns);
    END FOR;
END c_74F08_cfg;

CONFIGURATION  74LS08_cfg OF my_and IS
    FOR arch
      GENERIC MAP(tp=> 10 ns);
    END FOR;
END c_74LS08_cfg;
```

Generic (parents) (3/3)

Configuration of fulladd

```
CONFIGURATION fulladd_config OF fulladd IS
   FOR structure
      ...
      FOR u4: my_and use configuration work.74f08_cfg;
      END FOR;
      FOR u5: my_and use configuration work.74ls08_cfg;
      END FOR;
      ...
   END FOR;
END fulladd_config4;
```


Application of Generic (I)

7bits Shift left logic

```
ENTITY shl7 IS
    GENERIC ( Size: INTEGER := 8);
    PORT ( D : IN unsigned( Size - 1 downto 0 );
           Q : OUT unsigned( Size - 1 downto 0 ));
END shl7;
...

  constant count : unsigned(1 downto 0) := “11”;
...

  Q <= shl(D,count);
```

Application of Generic (II)

15bits Shift left logic

```
COMPONENT shl7
    GENERIC ( Size: INTEGER := 4);
    PORT (D :IN unsigned( Size - 1 DOWNTO 0 );
          Q :OUT unsigned( Size - 1 DOWNTO 0 ));
END COMPONENT;
Signal D,Q : unsigned(15 dwonto 0);
  ...

u0 : shl7 GENERIC MAP (16) PORT MAP ( D,Q);
```

❖VHDL Syntax

- ➢Sequential Statements
- ➢Concurrent Statements

Sequential Statements

Architecture Code Structure (1/3)

```
Architecture arch of en is
    {Declaration Statement}
begin
    {Concurrent Statement}
    {Sequential Statement}
end arch;
```

Architecture Code Structure (2/3)

Declaration statements

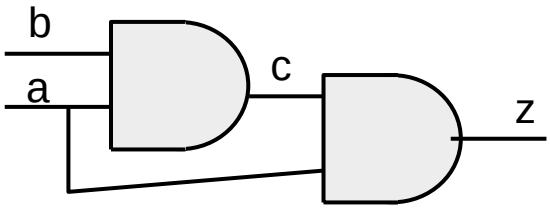
- Declares the internal signals that is similar to the statement in the packages.
- Be placed in the front of “begin”.
  - architecture
  - block
  - process
  - procedure
  - function

Architecture Code Structure (3/3)

Sequential Statement

- can use “variable”.
- three types:
  - process
  - function
  - procedure

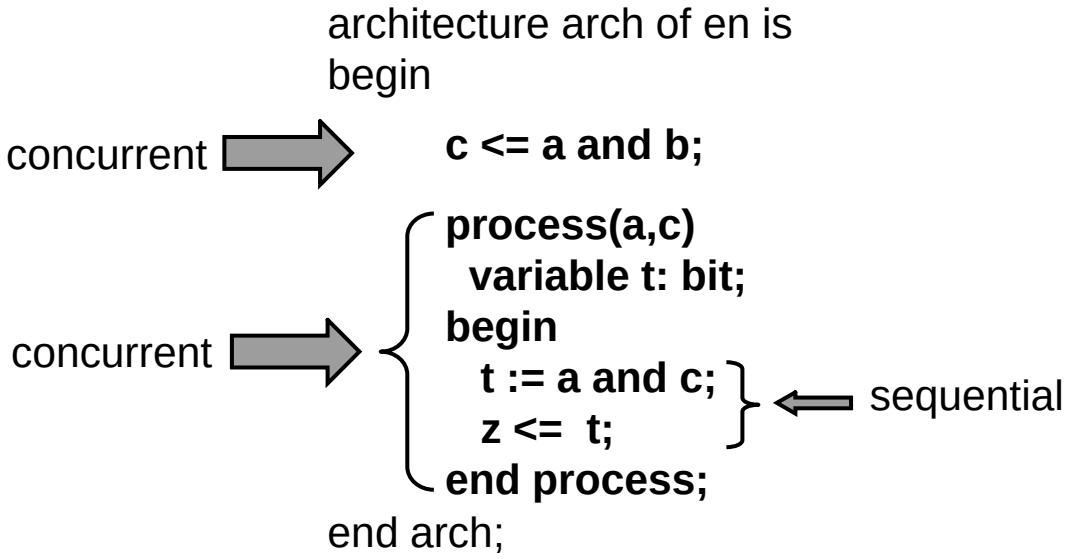
c<=a and b;
z<=a and c;



Concurrent Statement

- Each statement can be taken as a small circuit.
- Each statement will operate simultaneously.

Statements



Process

Process **(a,b)**
[process declaration]
begin
...

end Process

Process
[process declaration]
begin
...
wait on a,b
end Process

a,b -> sensitivity list

Sequential Statements

- Assignment
- IF
- CASE
- LOOP
- EXIT
- NEXT
- WAIT
- NULL
- ASSERT

```
[label:]process (...)
begin
     .............
end process  [label];
 -- a small circuit
```


Assignment

- Assignment

```
architecture arch of en is
   signal a: std_logic:='0';
begin
   process
      variable b: std_logic:='1';
   begin
      a <= '1';       -- signal
      b := '0'        -- variable;
   end;
end arch;
```

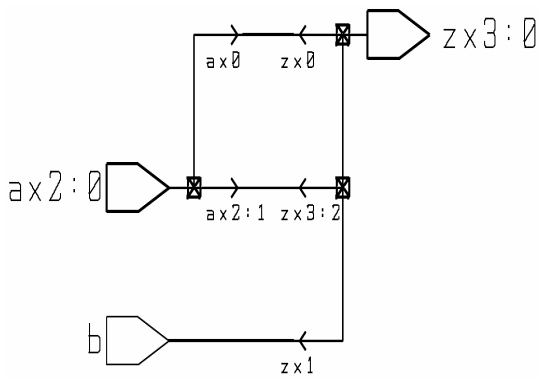
Assign (1/2)

Five kinds of Target

- ***data*** <= ‘1’; -- simple name
- ***databus(3)*** <=‘0’; -- indexed named
- ***databus(2 to 4)*** <=“101”; -- slices
- ***A.NUM_FIELD*** := 12; -- field names
- ***(data1,data2)*** <=“10”; -- Aggregates

Assign (2/2)

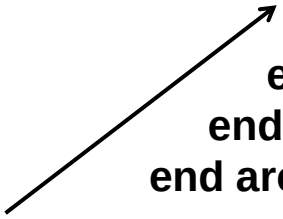
```
entity en is
  port (a : in bit_vector(2 downto 0);
        b : in bit;
        z : out bit_vector(3 downto 0));
end en;
```




```
architecture arch of en is
begin
  process(a,b)
  begin
    z(1 downto 0)<= (b , a(0) );
    z(3 downto 2)<=a(2 downto 1);
  end process;
end arch;
```

Signals in the Process (1/2)

```
Architecture arch of yy is
  signal a,b: std_logic:='1';
  signal c,d: std_logic:='0';
begin
    process(clk,a,b,c)
    begin
        if(clk'event and clk = '1') then
           c <= a and b;
           d <= c;
        end if;
    end process;
end arch;
```



Side effect error

Signals in the Process(2/2)

```
process(clk,a,b,c)
begin
    if(clk'event and clk = '1') then
       c <= a and b;
       d <= c;
    end if;
end process;
```

$C_{(t)} <= A \text{ and } B$
$D <= C_{(t-1)}$

Variables in the Process (1/2)

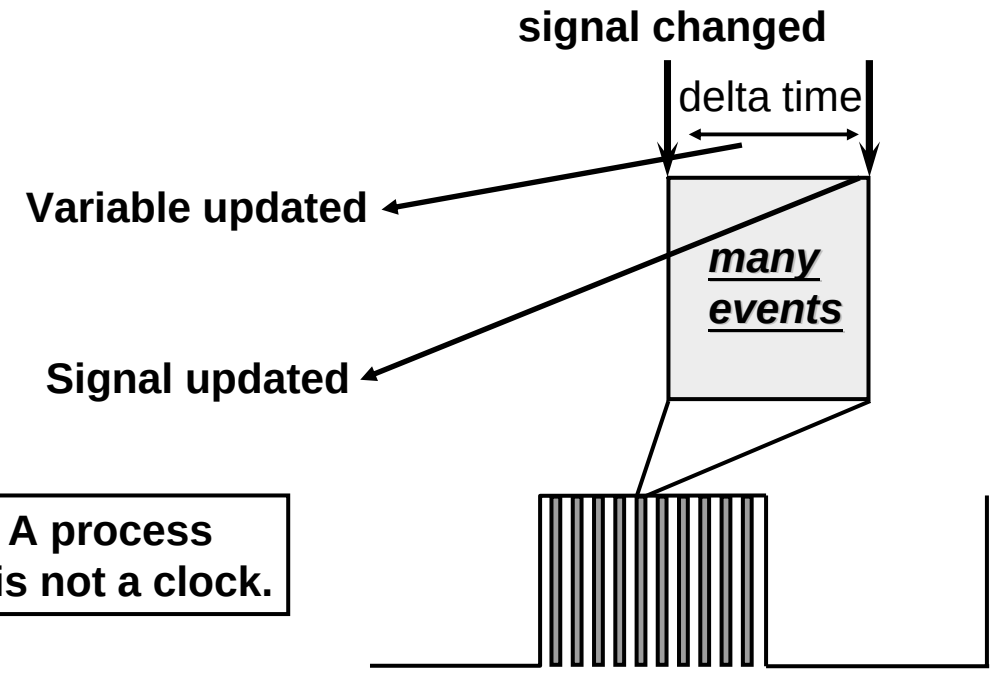
```
Architecture arch of yy is
  signal a,b: std_logic:='1';
  signal d: std_logic:='0';
begin
    process(clk,a,b)
      variable c : std_logic;
    begin
      if(clk'event and clk = '1') then
         c := a and b;
         d <= c;
      end if;
    end process;
end arch;
```


Variables in the Process(2/2)

```
process (clk,a,b)
    variable c : std_logic;
begin
     if(clk'event and clk = '1') then
        c := a and b;
        d <= c;
     end if;
end process;
```

$C_{(t)}$:= A and B
D <= $C_{(t)}$

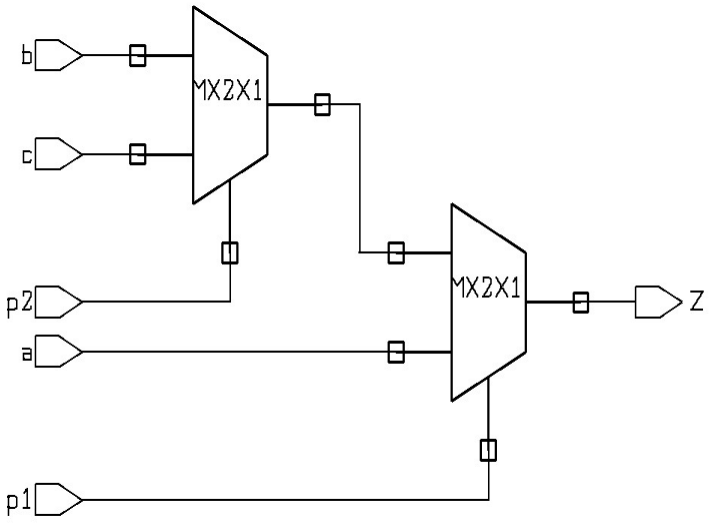
Delta time



Note: A process is not a clock.

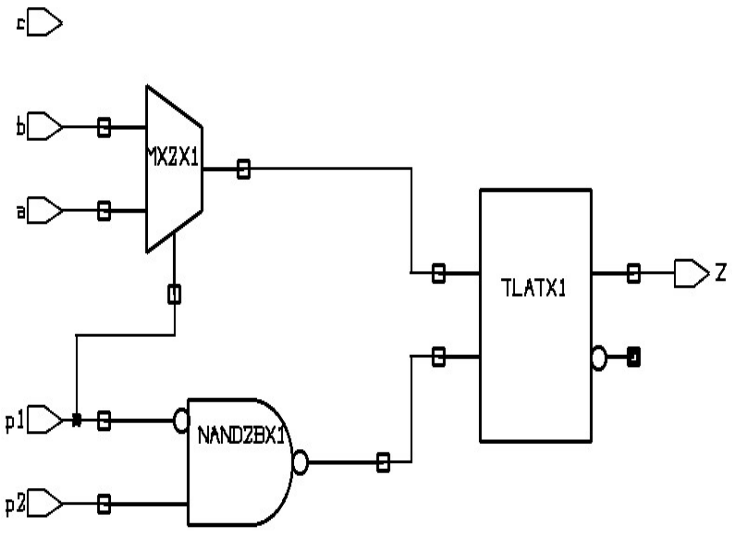
IF statement (1/2)

```
process( a,b,c,p1,p2 )
begin
   if (P1='1') then
      Z<=A;
   elsif (P2='0') then
      Z<=B;
   else
      Z<=C;
   end if;
end process;
```



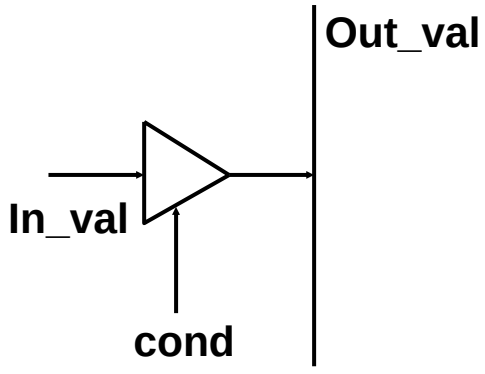
IF statement (2/2)

```
process( a,b,c,p1,p2 )
begin
   if (P1='1') then
      Z<=A;
   elsif (P2='0') then
      Z<=B;
   end if;
end process;
```



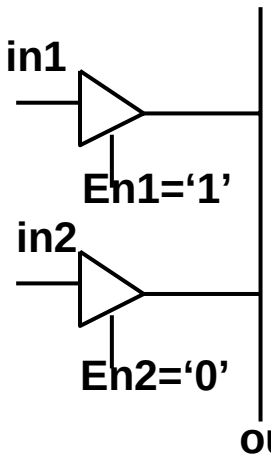
Tri – state (1/2)

```
signal out_val, in_val : std_logic;
signal cond : boolean;
...
if (cond) then
    out_val <= in_val;
else
    out_val <= 'Z';
end if;
```



Tri – state (2/2)

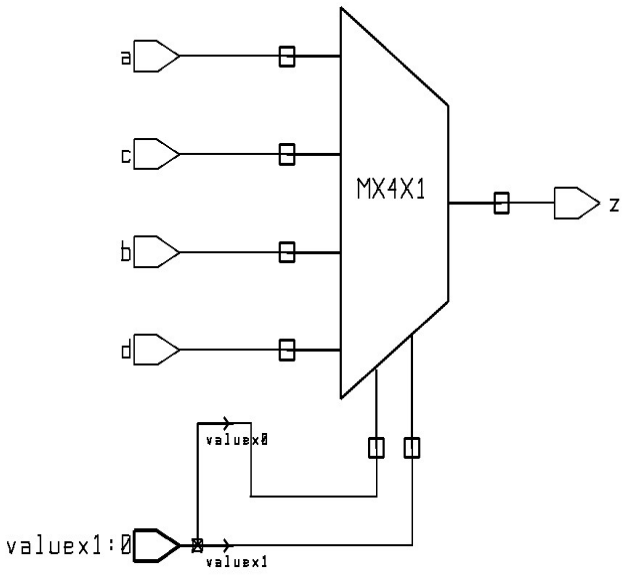
```
Process(en1, en2, in1, in2)
begin
  If (En1='1') then
     out<=in1;
  else
     out<='Z';
  end if;
  if(En2='1') then
     out<=in2;
  else
     out<='Z';
  end if;
end process; -- error
```




```
Process(en1, en2, in1, in2)
begin
   If (En1='1') then
      out<=in1;
   elsif (En2='1') then
      out<=in2;
   else
      out<='Z';
   end if;
end process;
```


Case Statement (1/2)

```
process(a,b,c,d,value)
begin
  case VALUE is
     when PICK_A  =>Z <= A ;
     when PICK_B => Z <= B;
     when PICK_C => Z <= C;
     when PICK_D => Z <= D;
  end case;

end process;
```



Case Statement (2/2)

```
package test is
  type ENUM is (PICK_A, PICK_B, PICK_C, PICK_D);
end test;

use work.test.all;

entity en is
     port (value: in ENUM;
           a,b,c,d: in bit;
           z: out bit);
end en;
```

PICK_A=00

PICK_B=01

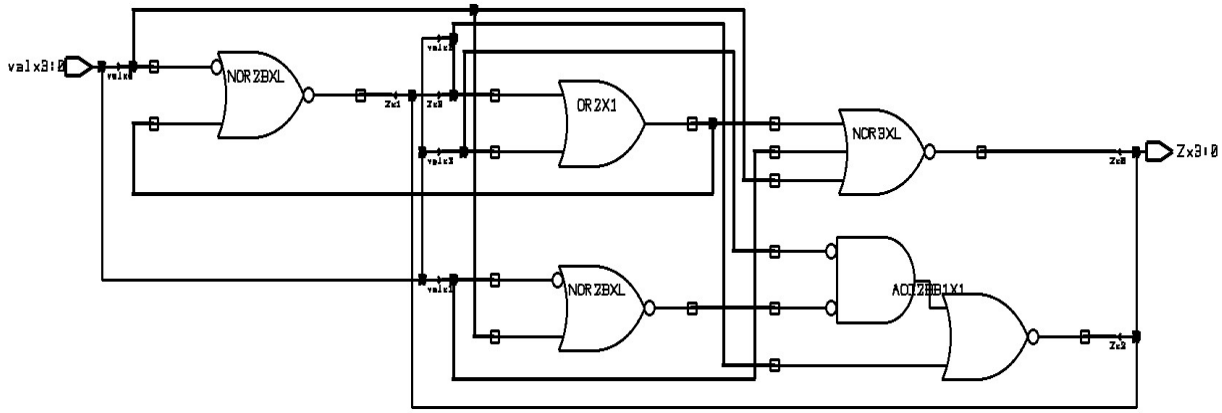
PICK_C=10

PICK_D=11

Case Statement – Encode (1/2)

```
signal VALUE :INTEGER range 0 to 15;
signal Z :BIT_VECTOR(3 downto 0);
    .............
Process(value)
begin
   case VALUE is
      when 0 =>              Z <= "0001";
      when 1|3 =>            Z <= "0010";
      when 4 to 7 | 2 =>     Z <= "0100";
      when others =>         Z <= "1000";
   end case;
end process;
```

Case statement – Encode (2/2)



Loop statement

- **loop**
  - can't be synthesized
- **while…loop**
  - can't be synthesized
- **for...loop**
  - used for synthesis
    - don't use wait statement
    - fixed range

Note：Loop and clock have no relational.
A loop is not a clock.

loop statement

PROCESS
BEGIN

Because wait exists, don't write sensitivity list.

[loop1:] LOOP

WAIT FOR 25 ns;
clk1<= NOT clk1;

END LOOP [loop1];

END PROCESS;

while...loop statement

PROCESS
VARIABLE a:integer :=1;
BEGIN

[loop1:] WHILE (a<=10)
LOOP
WAIT FOR 25 ns;
clk1<= NOT clk1;
a:=a+1;
END LOOP [loop1];

END PROCESS;

For...loop statement

```
PROCESS
BEGIN
   [loop1:] FOR a  IN  1 TO 10
       LOOP

        WAIT FOR 25 ns;
        clk1 <= NOT clk1;

   END LOOP [loop1];
END PROCESS;
```


While Loop vs. For Loop

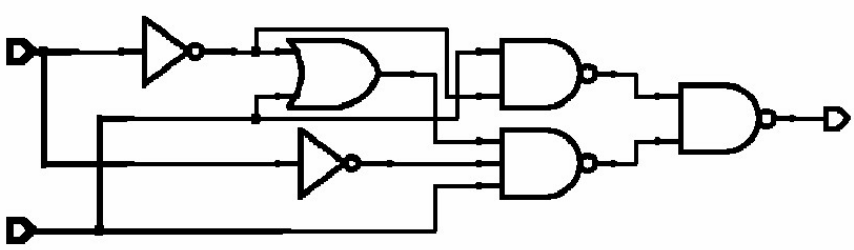
```
process
   VARIABLE a:integer :=1;  -- permanent existence
begin
  loop1: While(a<=10)
     LOOP
     WAIT FOR 25 ns;
     clk<= NOT clk ;
     a:=a+1;
  END LOOP loop1;

  loop2: FOR a IN 1 TO 10
     LOOP
     WAIT for 26 ns ;
     clk1 <= NOT clk1;
  END LOOP loop2;
end process;
```

Exit Statement

```
signal A,B: BIT_VECTOR(7 downto 0);
signal A_LESS_THAN_B: BOOLEAN;
...
for I in 7 downto 0 loop
   if(A(I)='1' and B(I)='0') then
      A_LESS_THAN_B<=FALSE;
      exit;
   elsif (A(I)='0' and B(I)='1') then
      A_LESS_THAN_B<=TRUE;
      exit;
   else
      null;
   end if;
end loop;
```

A>B

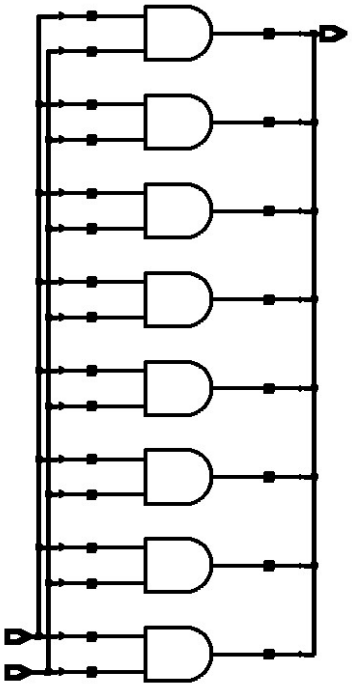
A<B



Next Statement

```
process(B,copy_enable)
begin
   A<="00000000";
   for I in 1 to 8 loop
      next when COPY_ENABLE(I)='0';
      A(I)<=B(I);
   end loop;
end process;
```

```
If(copy_enable(I)='0') then
        next;
else
        a(I)<=b(I);
end if;
```



Wait Statement

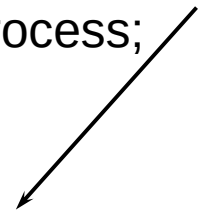
- **wait on sensitivity clause**
  - can't be synthesized
- **wait for timeout clause**
  - can't be synthesized
- **wait until condition clause**

Note：wait and sensitivity list in the process are not able to coexist.

Wait on

```
process (a,b)
begin
   z<=a and b;
end process;
```

```
process
begin
        z<=a and b;
        wait on a,b;
end process;
```



only signal

Wait until

```
process
begin
   i<=i+1;
   wait on i;
   count<=i;
   -- negative-edge trigger
   wait until clk='0' and clk'event;
end process;
```


Wait for

```
process
begin
   if pwring then
      wait for 150 ns;
      reset<='1';
      pwring <= false;
   end if;
end process;
```

```
process
begin
   wait on dtack until dtack='1'
                       for 150 ms;
   -- halt until dtack is '1',
   -- only wait for 150ms.
   if dtack='0' then
      buserr<='1';
   end if;
end process;
```

Variable vs. Signal

```
signal I, count: integer:=0;
.....
process
   begin
      i<=i+1;
      wait on i;
      count<= i;
      wait on clk;
end process;
```

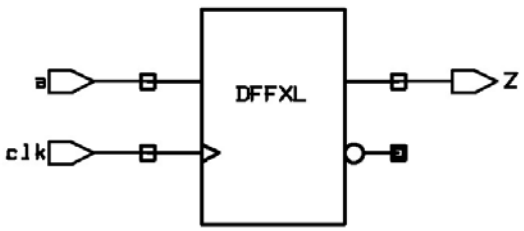
I	1 2 3 4 5	wait on
count	1 2 3 4 5	
I	1 2 3 4 5	no wait on
count	0 1 2 3 4	

```
signal count: integer:=0;
.....
process
   variable I: integer:=0;
   begin
      I:=i+1;
--    wait on i;
      count<= i;
      wait on clk;
end process;
```

I	1 2 3 4 5
count	1 2 3 4 5

Register - wait until vs. if (1/2)

```
process(a,clk)
begin
   if (clk='1' and clk'event) then
      Z<=A;
   end if;
end process;
```




```
process
begin
   wait until (clk='1' and clk'event);
      Z<=A;
end process;
```

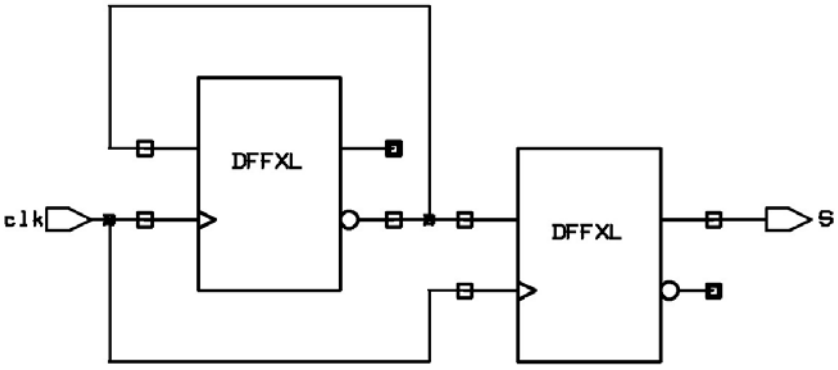
Register - wait until vs. if (2/2)

```
process(a,clk,reset)
begin
   if(reset ='1') then
     Z<='0';
   elsif(clk='1' and clk'event) then
      Z<=A;
   end if;      -- OK
end process;
```

```
if(a='1') then
    wait until (clk='1' and clk'event)
      Z<=A;
end if;   -- bad for synthesis
```

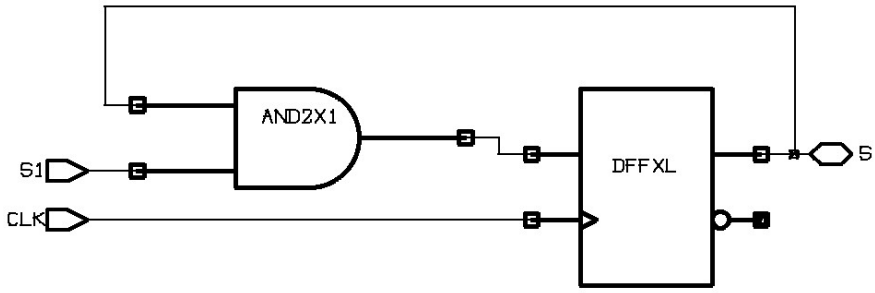

Variable vs. Register (1/2)

```
Process
  variable VAR: bit;
begin
  wait until clk'event and clk='1';
  VAR := not Var; -- register
  S <= Var;
end process;
```



Variable vs. Register (2/2)

```
Process
  variable VAR: bit;
begin
  wait until clk'event and clk='1';
  VAR := S and S1; -- combinational
  S <= Var;
end process;
```



Inside a clocked if statement

```
Process(CLK,S2)
  variable Var: bit;
begin
        if CLK'event and CLK='1' then
                Var:=not Var;
                S<=Var;
        end if;
        Var := S3;     -- X
        S <= S3;       -- X
        S2 <= Var;     -- X
        S4 <= S;       -- O
end process;
```

Latch

- **The signal or variable is in an equivalent process where not all the alternatives of a conditional expression are considered.**
- **The VHDL attribute 'event is not present in the conditional expression.**
- **The signal or variable is read in any path prior to being assigned a value.**

Latch or Combinational?

```
Process (S1)
begin
  if S1='1' then
    Var:=not Var; -- latch
    S<=Var;  -- latch
  end if;
end process;
```

```
Process (S1)
begin
   Var:='0';
   S<='0';
   if S1='1' then
     Var:= not Var;  -- Comb
     S<=Var;  -- Comb
   end if;
end process;
```

Register vs. Latch

- **wait statement**
  - wait until (clk'event and clk='1'); -- Flip-flops
  - wait until (clk='1'); -- Flip-flops
- **if statement without else**
  - if (clk'event and clk='1') -- Flip-flops
  - if(clk='1') -- latch

Assert statement (1/2)

- **ASSERT condition**
  REPORT string
  SEVERITY serverity_level
- **serverity_level**
  - **note**
  - **warning**
  - **error**
  - **failure**

Assert statement (2/2)

```
PROCESS
BEGIN
   WAIT UNTIL (nmi='0' OR int0='0')
         FOR 5 us;
   IF (nmi='1' AND int0='1') THEN
      err<='1';
   END IF;
   ASSERT (nmi='0' OR int0='0')
   REPORT "NO INTERRUPT OCCURRED IN 5 us"
   SEVERITY ERROR;
END PROCESS;
```

Note：When the condition of ASSERT is false, it will report message.

Subprograms

Procedure

procedure pro_name **(**A: in Bit; Z: out Bit**);**

Function

function fun_name **(**A,B: in Bit **) return** Integer**;**

return

- return; -- procedure
- return expression; -- Functions

Subprogram Declarations

```
package test is
  type BYTE is array (7 downto 0) of BIT;
  type NIBBLE is array (3 downto 0) of BIT;

  function IS_EVEN(NUM: in INTEGER) return BOOLEAN;
    -- Returns TRUE if NUM is even.

  procedure BYTE_TO_NIBBLES(B: in BYTE;
                            UPPER, LOWER: out NIBBLE);
    -- Splits a BYTE into UPPER and LOWER halves.
end test;
```


Subprogram Bodies

```
package body test is
  function IS_EVEN(NUM: in INTEGER) return BOOLEAN is
  begin
        return ((NUM rem 2) =0);
  end IS_EVEN;

  procedure BYTE_TO_NIBBLES(B: in BYTE;
                UPPER, LOWER: out NIBBLE) is
  begin
        UPPER:=NIBBLE(B(7 downto 4));
        LOWER:=NIBBLE(B(3 downto 0));
  end BYTE_TO_NIBBLES;
end test;
```

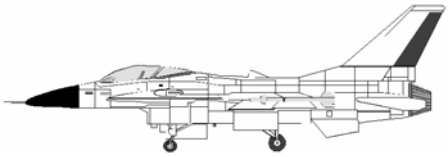
Subprogram Calls

```
process (….)
   signal INT: INTEGER;
   variable EVEN: BOOLEAN;
begin
   INT<=7;
   EVEN := IS_EVEN(INT);
end process;

process(….)
   variable TOP, BOT: NIBBLE;
begin
   BYTE_TO_NIBBLES("00101101", TOP, BOT);
end process;
```


Subprogram Overloading

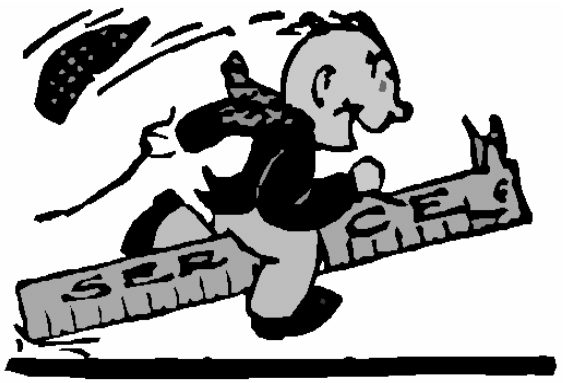
```
type SMALL is range 0 to 100;
type LARGE is range 0 to 10000;
```

```
function IS_ODD(NUM: SMALL) return BOOLEAN;
function IS_ODD(NUMBER: LARGE) return BOOLEAN;

signal A_NUMBER : SMALL;
signal B: BOOLEAN;
...
B <= IS_ODD(A_NUMBER);
```

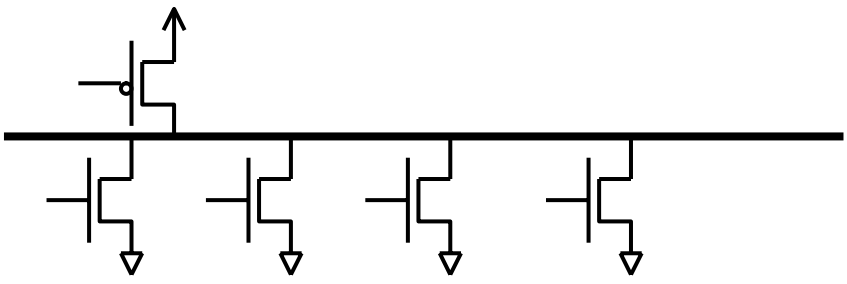


Operator Overloading

```
type NEW_BIT is ('0', '1', 'X');

function "and"(I1, I2: in NEW_BIT) return NEW_BIT;
function "or"(I1,I2:in NEW_BIT) return NEW_BIT;
...
signal A,B,C:NEW_BIT;
...
C<=(A and B) or C;
```



Resolution functions (1/5)



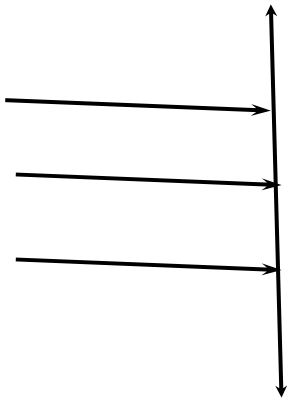
Only Simulation

Resolution functions (2/5)

- **Four steps**
  - type new_type is ...
    - Declares a new type.
  - function res_f(...) return new_type is
    - Declares a function returns the new type.
  - subtype res_t is res_f new_type;
    - Utilizes the prior declarations to declare a subtype.
  - signal bus : res_t;
    - Utilizes the subtype to allocate signals.

Resolution functions (3/5)

- **wired_and**
- **wired_or**



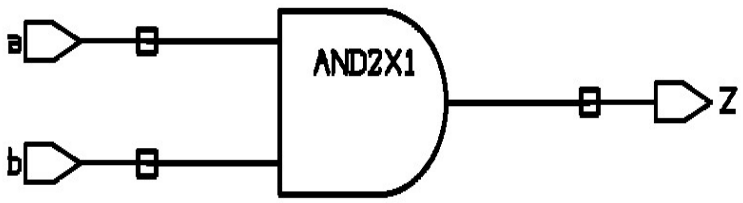
```
package RES_PACK is
   function RES_FUNC(DATA: in BIT_VECTOR) return BIT;
   subtype RESOLVED_BIT is RES_FUNC BIT;
end;
```

Resolution functions (4/5)

```
package body RES_PACK is
   function RES_FUNC(DATA: in BIT_VECTOR) return BIT is
      --pragma resolution_method wired_and
   begin
      -- The code in this function is ignored by VHDL Compiler
      -- but parsed for correct VHDL syntax
         for I in DATA'range loop
            if DATA(I)='0' then
                  return '0';
            end if;
         end loop;
         return '1';
   end;
end;
```


Resolution functions (5/5)

```
use work.RES_PACK.all;

entity WAND_VHDL is
   port(X,Y : in BIT;
        Z : out RESOLVED_BIT);
end WAND_VHDL;

architecture WAND_VHDL of WAND_VHDL is
begin
   Z<=X;
   Z<=Y;
end WAND_VHDL;
```



Concurrent Statements

Concurrent Statements

- Signal Assignment
- Process
- Block & Guarded Block
- Concurrent Assert
- Concurrent Procedure Call
- Conditional Signal Assignment
- Selected Signal Assignment
- Generate Statement

Signal Assignment

```
ARCHITECTURE BEHAVIOR OF NOR2 IS

BEGIN
        q <= a NOR b;
END BEHAVIOR;
```


Process Statement

```
architecture EXAMPLE of COUNTER is
begin
        process(IN_COUNT,CLEAR)
        begin
            if(CLEAR='1' or IN_COUNT=9) then
                OUT_COUNT<=0;
            else
                OUT_COUNT<=IN_COUNT+1;
            end if;
        end process;
end EXAMPLE;
```

Block Statement (1/2)

[label:] Block

{Declaration Statement}

begin

{Concurrent Statement}

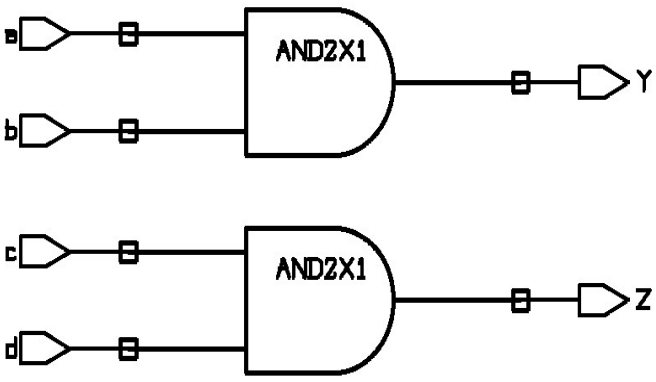
{Sequential Statement}

end block [label];

Block Statement (2/2)

```
B1: block
      signal S: BIT;
begin
      S<=A and B;
      B2: block
          signal S:BIT;
      begin
          S<=C and D;
          B3: block
          begin
              Z<=S;
          end block B3;
      end block B2;
      Y<=S;
end block B1;
```

- **concurrent statements hierarchically**



the Different of End

***label*: process()**
begin
end process *[label]*;

***label*: block**
begin
end block *[label]*;

component *and2*
end component;

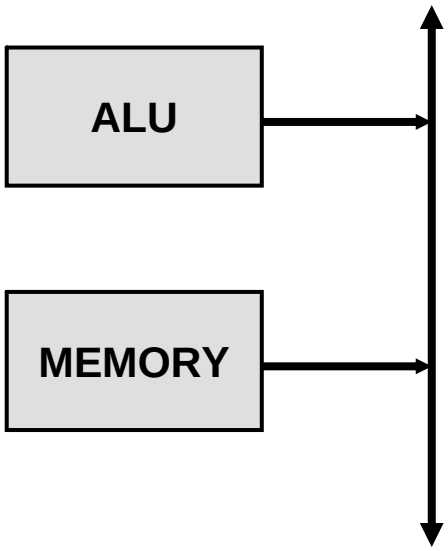
entity *name* is
end *name;*

architecture *arch* of *name* is
end *arch*;

configuration *test* of *name* is
end *test*;

package *mypack* is
end *[mypack];*

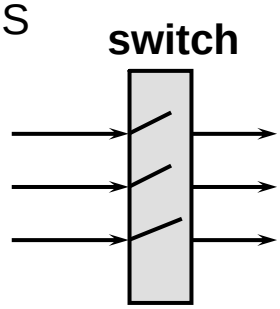
Guarded Block (1/2)



Guarded Block (2/2)

- **Enable and disable drivers contained in the block.**
  - not be supported by synthesis

```
ARCHITECTURE guarded_latch OF latch IS
BEGIN
    B1: BLOCK (en='1')
    BEGIN
       q <= guarded d after 5 ns;
       qbar <= guarded not (d) after 6 ns;
    END BLOCK B1;
END guarded_latch;
```



Guarded Signal

- **SIGNAL KIND**
  - REGISTER
    - signal to retain its last value.
  - BUS
    - signal to take on the default value sepcified by the bus resoluation function.

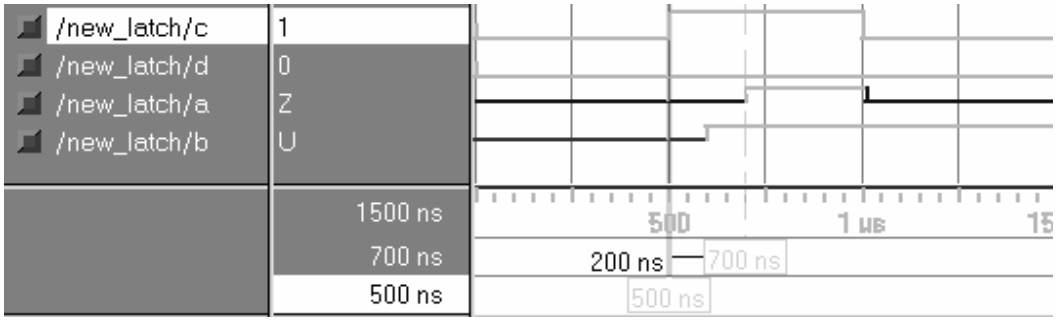
Disconnect specification (1/2)

- **model the time delay of tri-state signals**
- **used for SIGNAL KIND signals**

```
ENTITY new_latch IS
   PORT(c,d:IN std_logic;
            a: OUT std_logic BUS;
            b: OUT std_logic);
   -- no register here, but port will hold the original value.
   DISCONNECT a: std_logic AFTER 10 ns;
END new_latch;
```


Disconnect specification (2/2)

```
B1: BLOCK(c='1')
BEGIN
     b<= guarded not d after 100 ns;
     a<= guarded not d after 200 ns;
END BLOCK B1;
```



Concurrent Procedure Calls (1/3)

```
procedure ADD(signal A, B: in BIT;
              signal SUM: out BIT);
…
architecture arch of en is
begin
   ADD(A,B,SUM);       -- Concurrent procedure call
   ...
   process(A,B)        -- The equivalent process
   begin
      ADD(A,B,SUM);       -- Sequential procedure call
   end process;
end arch;
```


Concurrent Procedure Calls (2/3)

```
Architecture arch of en is
        signal S:boolean;

Procedure P(A:boolean) is
begin
    assert S
    report "S is false -- procedure P"
    severity NOTE;
end P;

Procedure P2(A_s:boolean) is
begin
    assert A_s
    report "S is false -- procedure P2"
    severity NOTE;
end P2;

Begin
  P(False);
  P2(S);

  S<=  True,
       False after 10 ns,
       True after 20 ns,
       False after 30 ns;
end arch;
```

Concurrent Procedure Calls (3/3) [Results]

0 NS
Assertion NOTE at 0 NS in design unit CPROC(CPROC_A) from process /CPROC/_P0:
"S is False -- Procedure P”
Assertion NOTE at 0 NS in design unit CPROC(CPROC_A) from process /CPROC/_P1:
"S is False -- Procedure P2"10.00 NSAssertion NOTE at
10.00 NS
in design unit CPROC(CPROC_A) from process /CPROC/_P1:
"S is False -- Procedure P2"30.00 NSAssertion NOTE at
30.00 NS
in design unit CPROC(CPROC_A) from process /CPROC/_P1:
"S is False -- Procedure P2"

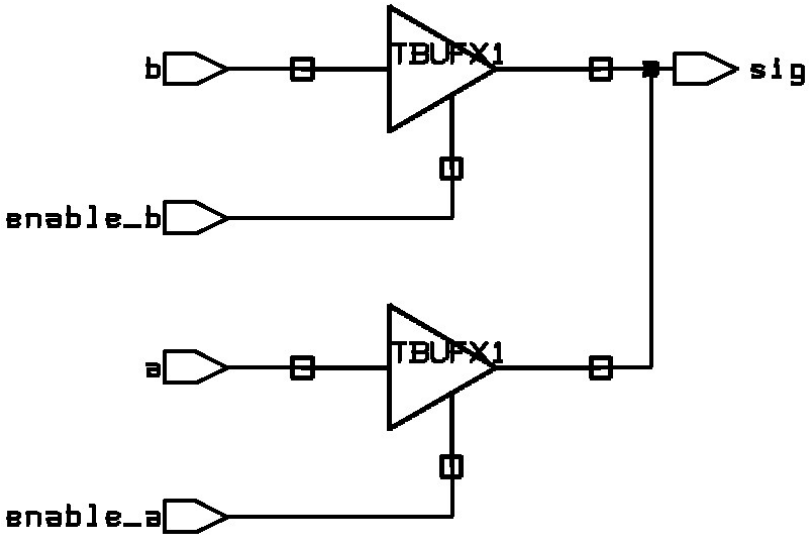
Conditional Signal Assignment

```
Z <= A when ASSIGN_B = '1' else
     B when ASSIGN_A = '1' else
     C ;
```

```
process(A, ASSIGN_A, B, ASSIGN_B, C)
begin
    if ASSIGN_A = '1' then
        Z<=A;
    elsif ASSIGN_B = '1' then
        Z<=B;
    else
        Z<=C;
    end if;
end process;
```

Process statement (tri-state)

```
A_OUT<=A when ENABLE_A else 'Z';
B_OUT<=B when ENABLE_B else 'Z';

process(A_OUT)
begin
      SIG<=A_OUT;
end process;
process(B_OUT)
begin
      SIG<=B_OUT;
end process;
```

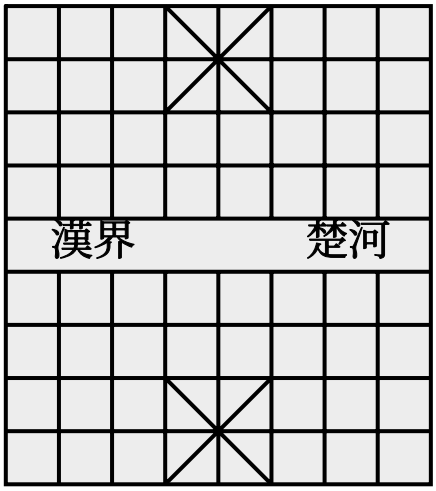


Selected Signal Assignment

```
signal A, B, C, D, Z: BIT;
signal CONTROL: bit_vector(1 downto 0 );
...

with CONTROL select
    Z <= A when "00",
         B when "01",
         C when "10",
         D when "11";
```

```
process(CONTROL, A, B, C, D)
begin
      case CONTROL is
         when 0=>
                Z<=A;
         when 1=>
                Z<=B;
         when 2=>
                Z<=C;
         when 3=>
                Z<=D;
      end case;
end process;
```

Generate

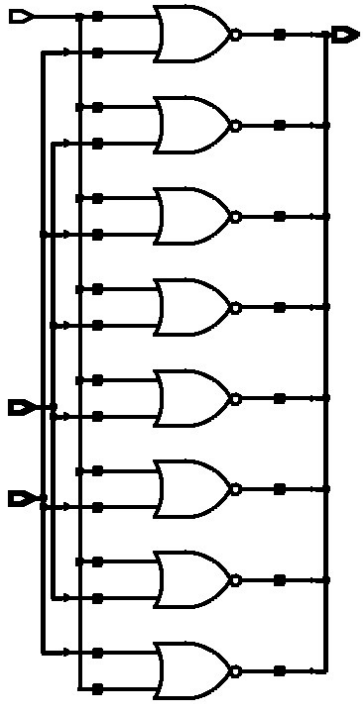


Generate Statement – FOR (1/4)

```
signal A, B: bit_vector(3 downto 0);
signal C: bit_vector(7 downto 0);
signal X: bit;
...
LABEL: for I in 3 downto 0 generate
    C(2*I+1)<=A(I) nor X;
    C(2*I)<=B(I) nor X;
end generate [LABEL];
```

```
C(7) <= A(3) nor X;
C(6) <= B(3) nor X;
C(5) <= A(2) nor X;
C(4) <= B(2) nor X;
C(3) <= A(1) nor X;
C(2) <= B(1) nor X;
C(1) <= A(0) nor X;
C(0) <= B(0) nor X;
```

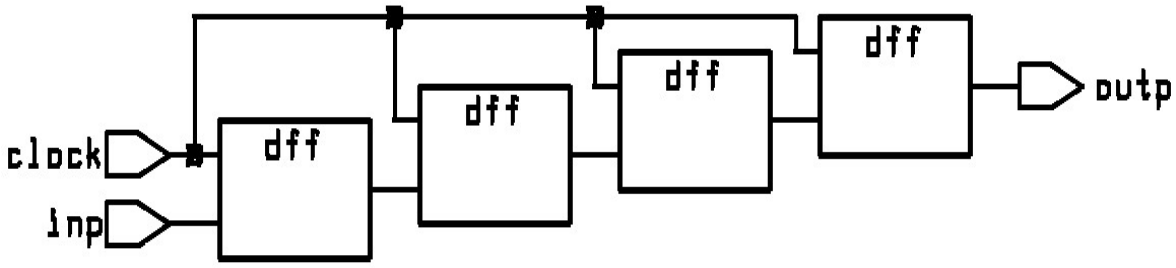
Generate Statement – FOR (2/4)



Generate Statement – FOR (3/4)

```
component dff
      port(sig_in,clock:in std_logic;
            q: out std_logic);
end component;
signal a: std_logic_vector(0 to 4);
...
a(0)<=inp;
g1: FOR i IN 0 TO 3 GENERATE
       u0 : dff port map(a(i),clock,a(i+1));
END GENERATE;                    -- concurrent
outp<=a(4);
```

Generate Statement – FOR (4/4)



Generate Statement – IF (1/5)

CONVERTER

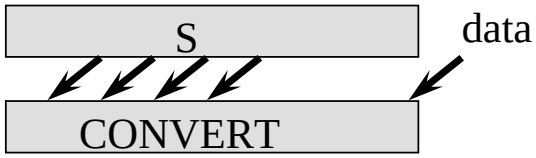
```
entity CONVERTER is
        generic(N: INTEGER:=8);
        port(CLK,DATA: in BIT;
                Convert: buffer BIT_VECTOR(N-1 downto 0));
end CONVERTER;
...
signal S : bit_vector(N-2 downto 0);
...
```

Generate Statement – IF (2/5)

CONVERTER

G: for I in convert'range generate

```
-- shift (N-1) data bit into high-order bit
G1: if (I=CONVERT'left) generate
      process begin
          wait until (CLK'event and CLK='1');
          CONVERT(I)<=S(I-1);
      end process;
end generate G1;
```



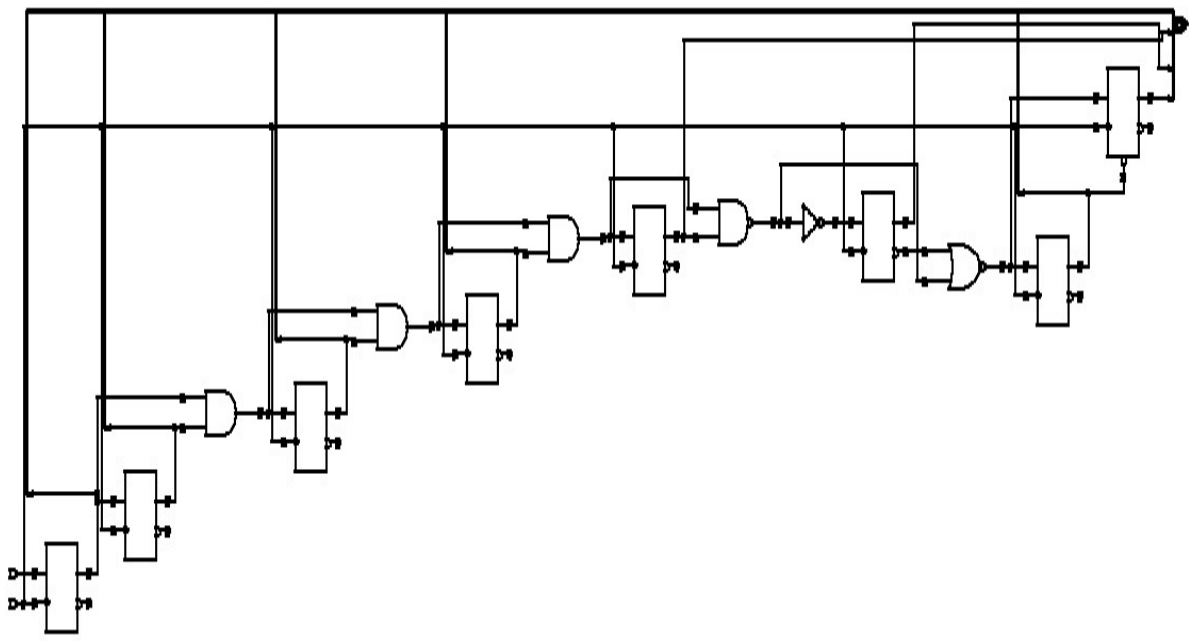
Generate Statement – IF (3/5)

```
-- shift middle bits up
G2:if(I>CONVERT'right and I<CONVERT'left) generate
      S(I)<=S(I-1) and CONVERT(I);
      process begin
          wait until (CLK'event and CLK='1');
          CONVERT(I)<=S(I-1);
      end process;
end generate G2;
```

Generate Statement – IF (4/5)

```
        -- Move DATA into low-order bit
        G3:if(I=CONVERT'right) generate
            process begin
                wait until (CLK'event and CLK='1');
                CONVERT(I)<=DATA;
            end process;
            S(I)<=CONVERT(I);
        end generate G3;
***end generate G;***
```


Generate Statement – IF (5/5)



Concurrent Assertion

```
ARCHITECTURE behv OF ckints IS
BEGIN
   err<='1' WHEN (nmi='1' AND int0 ='1')
      ELSE '0';
   ASSERT (nmi='0' OR int0='0')
   REPORT "NO INTERRUPT OCCURRED"
   SEVERITY ERROR;
END behv;
```


❖Modeling Logic Circuits

- ➢Combinational Logic
- ➢Sequential Logic
- ➢Finite State Machine

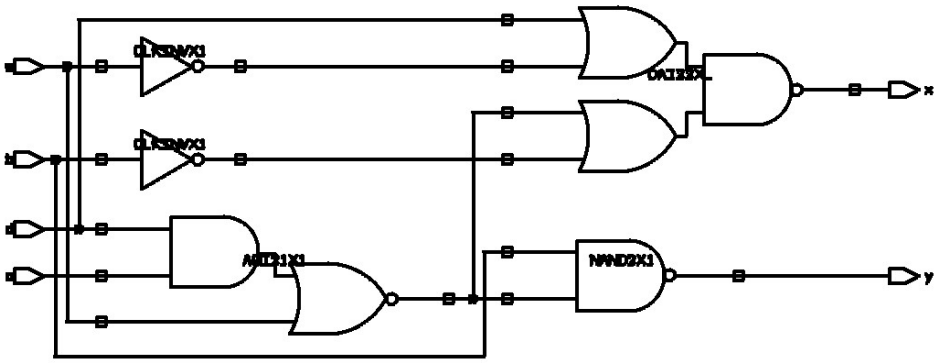
Combinational Logic

Boolean function (1/2)

$$x = A \bullet B \bullet C' + A \bullet D' + B \bullet C \bullet D;$$
$$y = A + B' + C \bullet D;$$

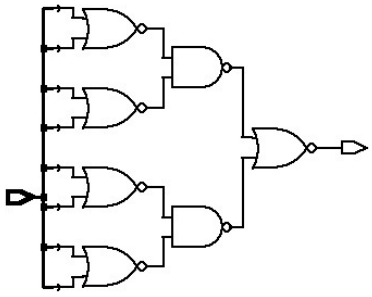
```
    …
    signal A,B,C,D : std_logic;
    signal X,Y : std_logic;
ARCHITECTURE a OF BL_EQ IS
BEGIN
   X <= (A and B and not(C)) or (A and not(D))
                             or (B and C and D);
   Y <= A or not(B) or (C and D);
END a;
```

Boolean function (2/2)

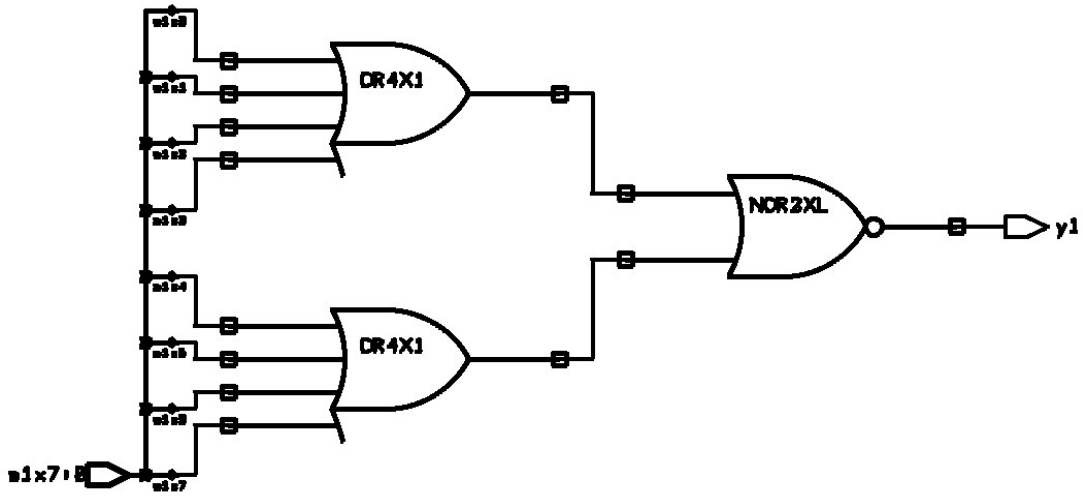


Logical equations (1/2)

```
p1:process(a1)
  variable v: unsigned(1 downto 0);
begin
  v:=((a1(0) nor a1(1)), (a1(2) nor a1(3)));
  s1<=v(0) nand v(1);
end process p1;

p2:process (a1)
  variable v: unsigned(1 downto 0);
begin
  v:=(a1(4) nor a1(5))&(a1(6) nor a1(7));
  s2<=v(0) nand v(1);
end process p2;

y1<=s1 nor s2;
```

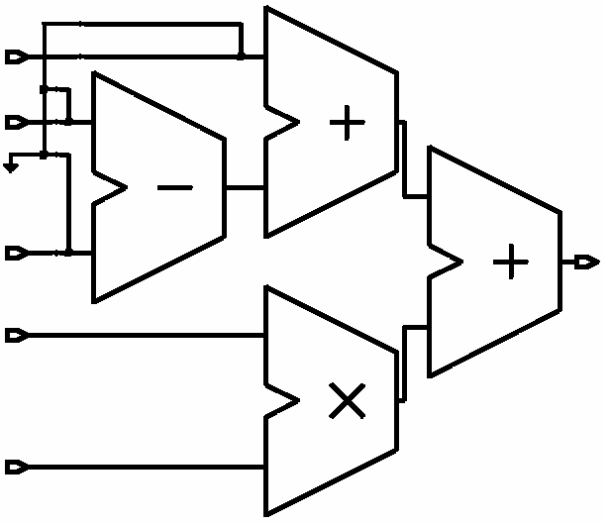


Logical equations (2/2)



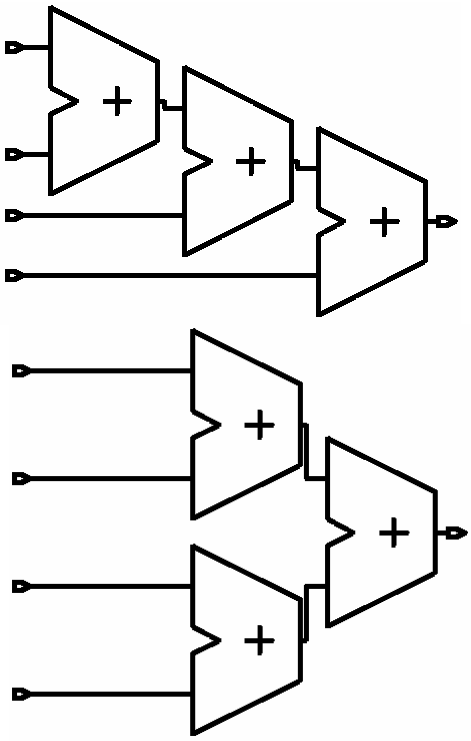
Arithmetic equations

```
process(a2,b2,c2,d2,e2)
begin
   y2<=a2+(b2-c2)+(d2*e2);
end process;
```



Logical structure control

```
process(a1,b1,c1,d1,a2,b2,c2,d2)
begin
    y1<=a1+b1+c1+d1;
    y2<=(a2+b2)+(c2+d2);
end process;
```

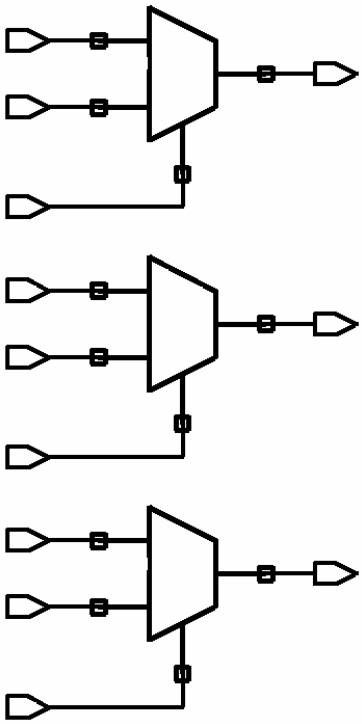


Combinational Logic

- **process**
  - if
  - case
- **concurrent**
  - conditional assign
  - with...select

Multiplexers

```
y1<=a1 when sel1='1' else
       b1 ;

process(sel2,a2,b2,sel3,a3,b3)
begin
   y2<=b2;
   if(sel2='1') then
      y2<=a2;
   end if;

   if(sel3='1') then
      y3<=a3;
   else
      y3<=b3;
   end if;
end process;
```



MUX4-1 (1/2)

```
process(sel,a,b,c,d)
begin
  if(sel="00") then
      y<=a;
  elsif(sel="01") then
      y<=b;
  elsif(sel="10") then
      y<=c;
  else
      y<=d;
  end if;
end process;
```

```
y<=a when sel="00" else
     b when sel="01" else
     c when sel="10" else
     d;
```

```
if(sel(1)='0') then
   if(sel(0)='0') then
       y<=a;
   else
       Y<=b;
   end if;
else
    if(sel(0)='0') then
       y<=c;
    else
       y<=d;
     end if;
end if;
```

MUX4-1 (2/2)

```
process(sel,a,b,c,d)
begin
  case sel is
    when "00" => y<=a;
    when "01" => Y<=b;
    when "10" => y<=c;
    when "11" => y<=d;
    when others => y<=a;
  end case;
end process;
```

```
with sel select
    y<=a when "00",
       b when "01",
       c when "10",
       d when "11",
       a when others;
```


8-3 Encoder (1/5)

```
process(a)
begin
   if (a="00000001“) then y<="000";
   elsif (a="00000010") then y<="001";
   elsif (a="00000100") then y<="010";
   elsif (a="00001000") then y<="011";
   elsif (a="00010000") then y<="100";
   elsif (a="00100000") then y<="101";
   elsif (a="01000000") then y<="110";
   elsif (a="10000000") then y<="111";
   else y<="XXX";
   end if;
end process;
```

8-3 Encoder (2/5)

```
process(a)
begin
    case a is
      when "00000001" => y<="000";
      when "00000010" => y<="001";
      when "00000100" => y<="010";
      when "00001000" => y<="011";
      when "00010000" => y<="100";
      when "00100000" => y<="101";
      when "01000000" => y<="110";
      when "10000000" => y<="111";
      when others => y<="XXX";
    end case;
end process;
```


8-3 Encoder (3/5)

```
y <= "000" when a="00000001" else
     "001" when a="00000010" else
     "010" when a="00000100" else
     "011" when a="00001000" else
     "100" when a="00010000" else
     "101" when a="00100000" else
     "110" when a="01000000" else
     "111" when a="10000000" else
     "XXX";
```

8-3 Encoder (4/5)

```
with a select
    y<= "000" when "00000001",
        "001" when "00000010",
        "010" when "00000100",
        "011" when "00001000",
        "100" when "00010000",
        "101" when "00100000",
        "110" when "01000000",
        "111" when "10000000",
        "XXX" when others;
```


8-3 Encoder (5/5)

```
process(a)
  variable n: integer range 0 to 7 ;
  variable test: unsigned (7 downto 0);
 begin
  test:="00000001";
  y<="XXX";
  for n in 0 to 7 loop
     if(A=test) then
       y<=conv_unsigned(n,3);
       exit;
     end if;
     test:=shl(test,"1");
  end loop;
 end process;
```

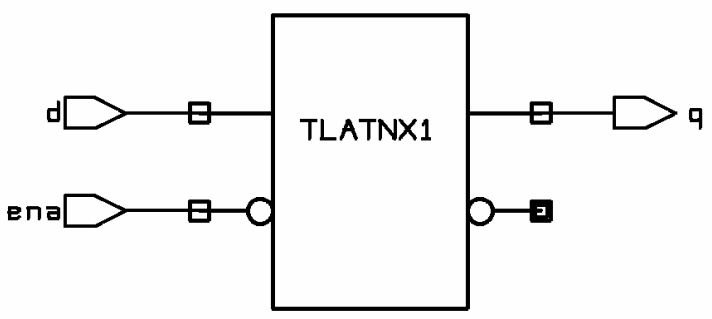
Comparators

```
compare: process(a1,b1,a2,b2,a3,b3)
  begin
   y1<='1';
   for n in 0 to 5 loop
       if(a1(n) /= b1(n)) then
           y1<='0';
           exit;
       else
           null;
       end if;
   end loop;
   if(a3=b3) then
       y3<='1';
   else
       y3<='0';
   end if;

   y2<='0';
   if(a2=b2) then
       y2<='1';
   end if;

end process;
```


Sequential Logic

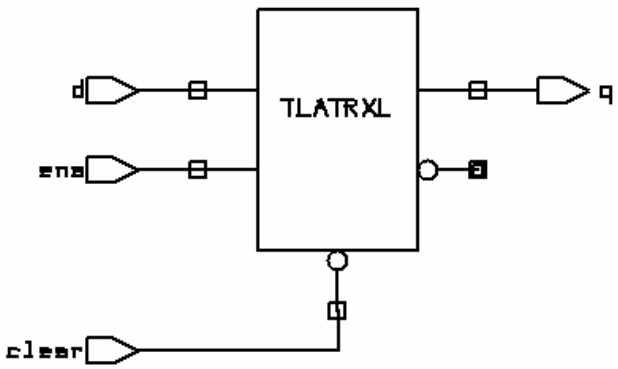
Latch

```
architecture a of latch is
begin
   process(ena,d)
   begin
      if ena = '0' then
         q <= d;
      end if;
   end process;
end a;
```



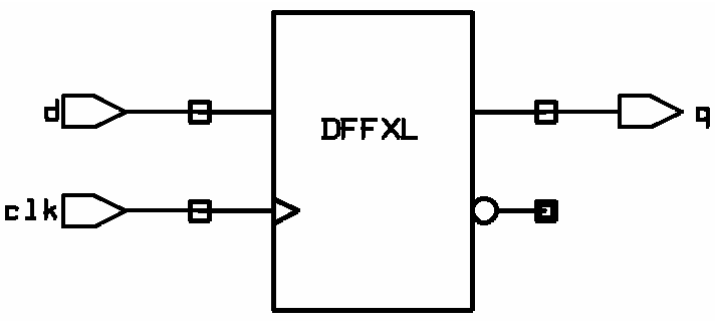
Latch with Clear Inputs

```
architecture a of latch is
begin
   process(ena,d,clear)
   begin
      if clear = ‘0’ then
         q <= ‘0’;
      elsif ena = ‘1' then
         q <= d;
      end if;
   end process;
end a;
```



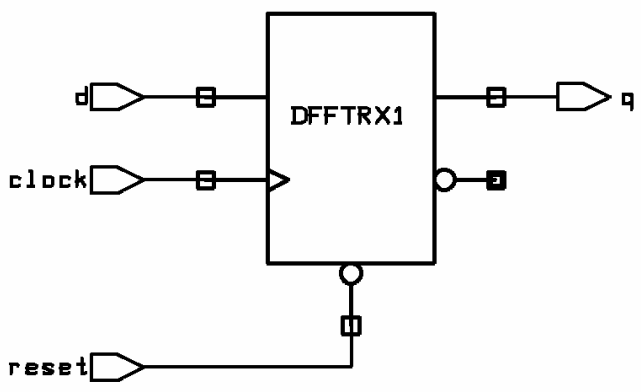
Flip-Flop

```
architecture a of d_ff is
begin
   process(clk,d)
   begin
        if clk‘event and clk = '1' then
           q <= d;
        end if;
   end process;
end a;
```



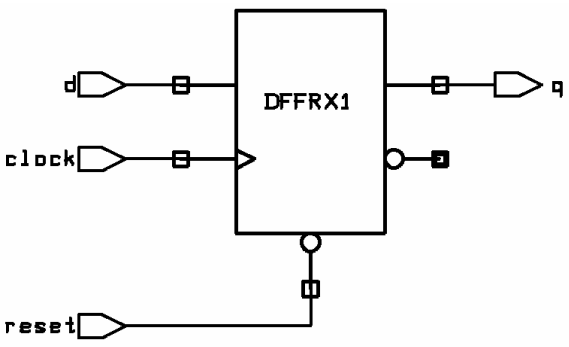
Flip-Flop (with Synchronous Reset)

```
architecture a of dff_sr is
begin
   process(reset,clk,d)
   begin
      if clk'event and clk = '1' then
         if reset = '0' then
            q <= '0';
         else
            q <= d;
         end if;
      end if;
   end process;
end a;
```



Flip-Flop (with Asynchronous Reset)

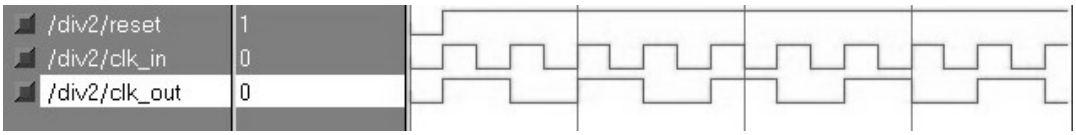
```
architecture a of dff_ar is
begin
   process(reset,clk,d)
   begin
     if reset = '0' then
        q <= '0';
     elsif clk'event and clk = '1' then
        q <= d;
     end if;
   end process;
end a;
```

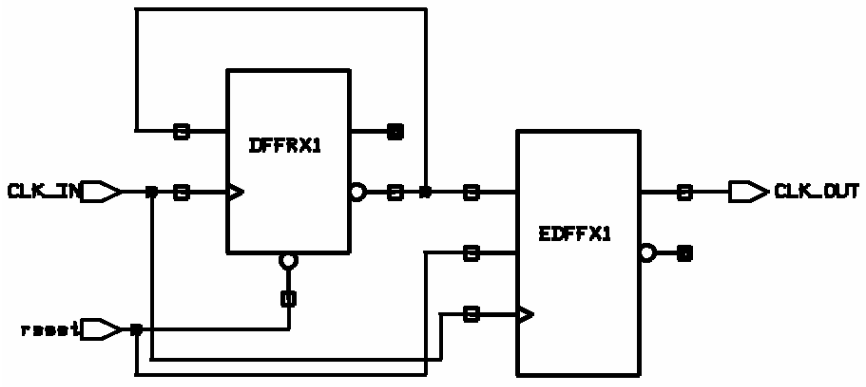


Frequency Divider (1/2)

```
architecture a of div2 is
begin
   process(clk_in,reset)
      variable clk_2: std_logic;
   begin
      if reset = '0' then
         clk_2 := '0';                -- set initial value
      elsif clk_in'event and clk_in = '1' then
         clk_2 := not clk_2;
         clk_out <= clk_2;
      end if;
   end process;
end a;
```


Frequency Divider (2/2)

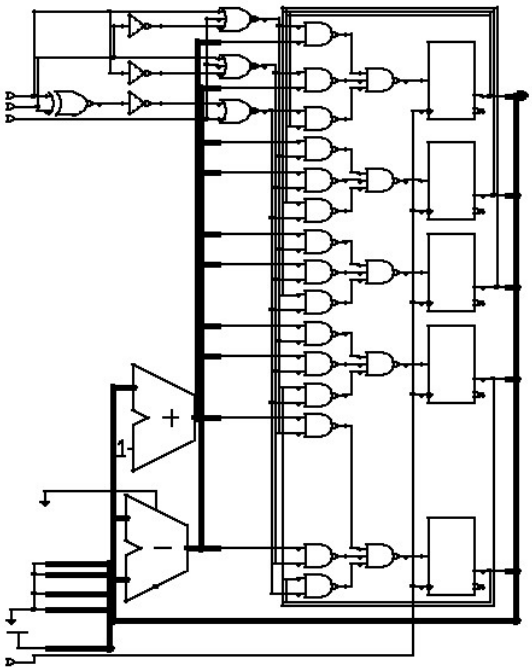




up-by-one down-by-two Counter (1/2)

```
process(clock,reset,up,down)
    variable updown : unsigned(1 downto 0);
begin
   updown:=up&down;
   if (reset='1') then
         count<="00000";
   elsif clock'event and clock='1' then
      case updown is
         when "00"=> count<=count;
         when "10"=> count<=count+1;
         when "01"=> count<=count-2;
         when others=> count<=count;
       end case;
   end if;
end process;
```

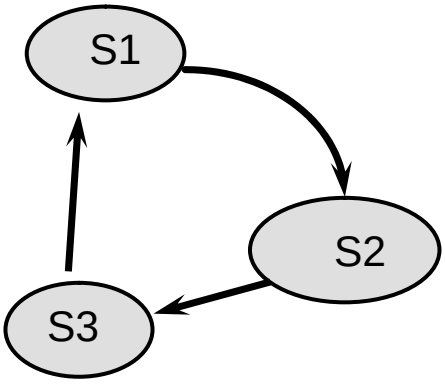

up-by-one down-by-two Counter (2/2)

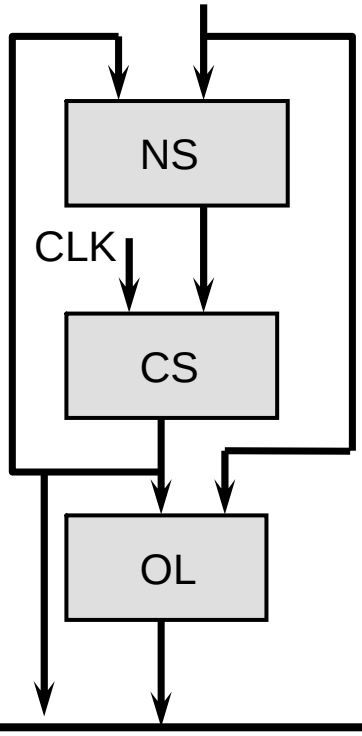


Finite State Machine

Finite State Machine (1/2)

- Current state
- Next state
- Output logic





Finite State Machine (2/2)

- **three ways**
  - CS, NS, OL
  - CS + NS, OL
  - NS + OL, CS
- **OL and CS don't put into the same process.**
  - Maybe the Output side will connect with a latch after being synthesized.

Current State (CS)

```
seq: process(clock,reset)
  begin
     if (reset='1') then
        currentstate<=st0;
     elsif clock'event and clock='1' then
         currentstate<=nextstate;
     end if;
  end process seq;
```

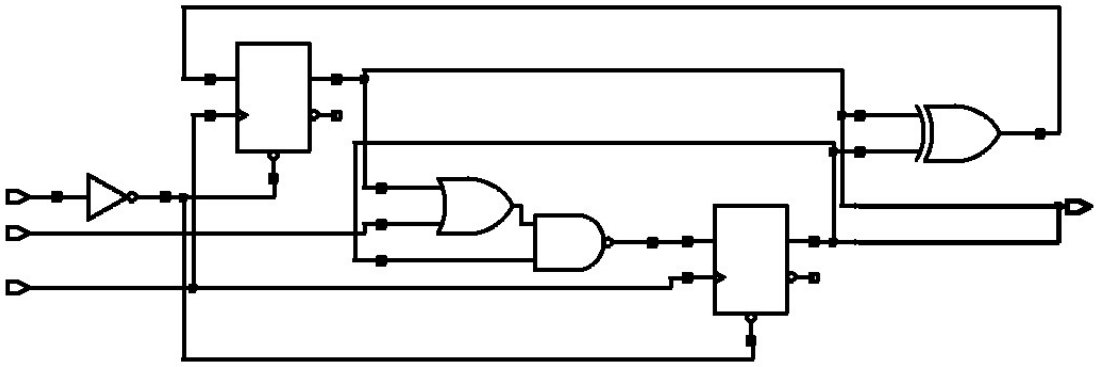
Next state (NS)

```
comb: process (control,currentstate)
  begin
    case currentstate is
      when st0=> nextstate<=st1;
      when st1=> if (control='1') then
                   nextstate<=st2;
                 else
                   nextstate<=st3;
                 end if;
      when st2=> nextstate<=st3;
      when st3=> nextstate<=st0;
      when others=> nextstate<=st0;
    end case;
end process comb;
```

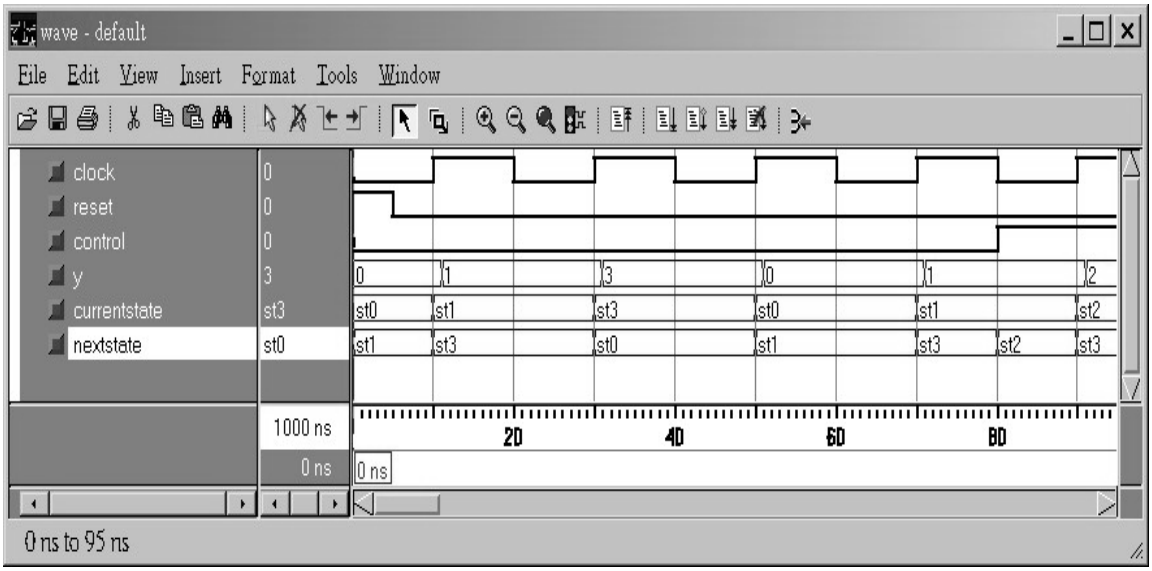

Output logic (OL)

```
with currentstate select
      y <=  0 when st0,
            1 when st1,
            2 when st2,
            3 when st3,
            0 when others;
```

FSM Circuits



Simulation Results



CS+NS

```
next_cur: process(clock,reset)
begin
   if(reset='1') then
        state<=st0;
   elsif clock'event and clock='1' then
        case state is
            when st0=> state<=st1;
            when st1=> if(control='1') then
                            state<=st2;
                       else
                            state<=st3;
                        end if;
            when st2=> state<=st3;
            when st3=> state<=st0;
            when others=> state<=st0;
        end case;
   end if;
end process next_cur;
```

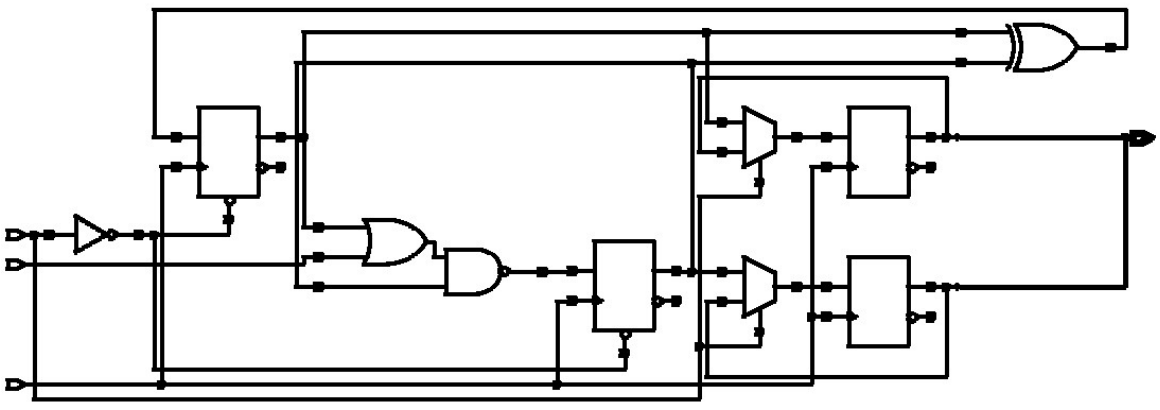

NS+OL

```
comb: process (control,currentstate)
begin
      case currentstate is
          when st0=> nextstate<=st1;
                     y<=0;
           when st1=> if(control='1') then
                          nextstate<=st2;
                      else
                          nextstate<=st3;
                      end if;
                     y<=1;
           when st2=> nextstate<=st3;
                     y<=2;
           when st3=> nextstate<=st0;
                     y<=3;
           when others=> nextstate<=st0;
                          y<=0;
      end case;
end process comb;
```

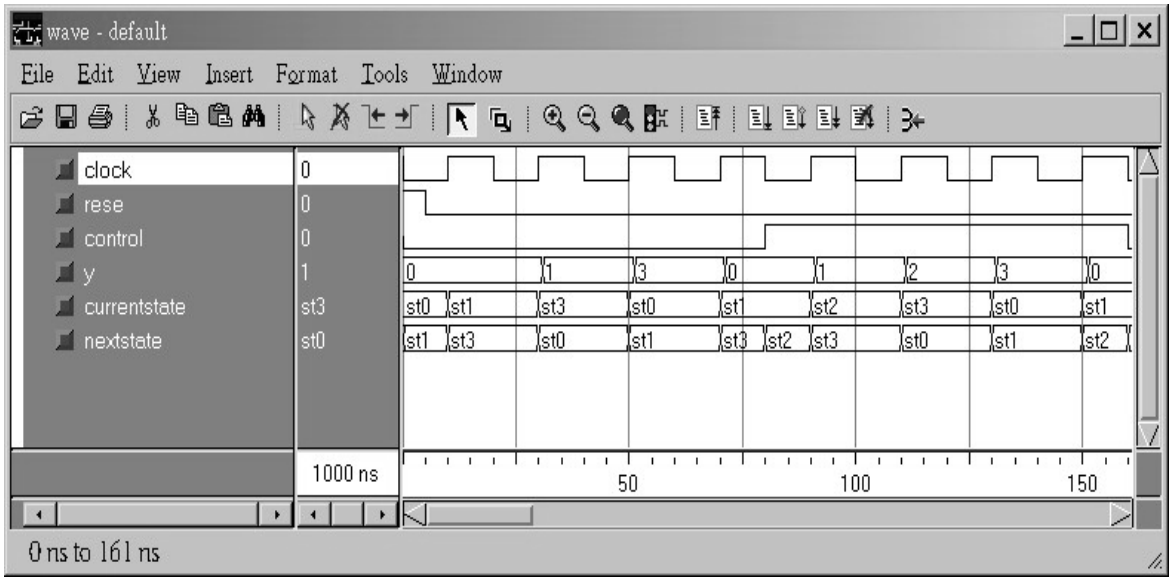
CS+OL (I)

```
seq: process(clock,reset,currentstate)
begin
      if(reset='1') then
           currentstate<=st0;
      elsif clock'event and clock='1' then
           currentstate<=nextstate;
           case currentstate is
                when st0=>y<=0;
                when st1=>y<=1;
                when st2=>y<=2;
                when st3=>y<=3;
           end case;
      end if;
end process seq;
```


CS+OL (I) : Wrong Circuit

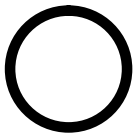


CS+OL (I) : Wrong Results



CS+OL (II)

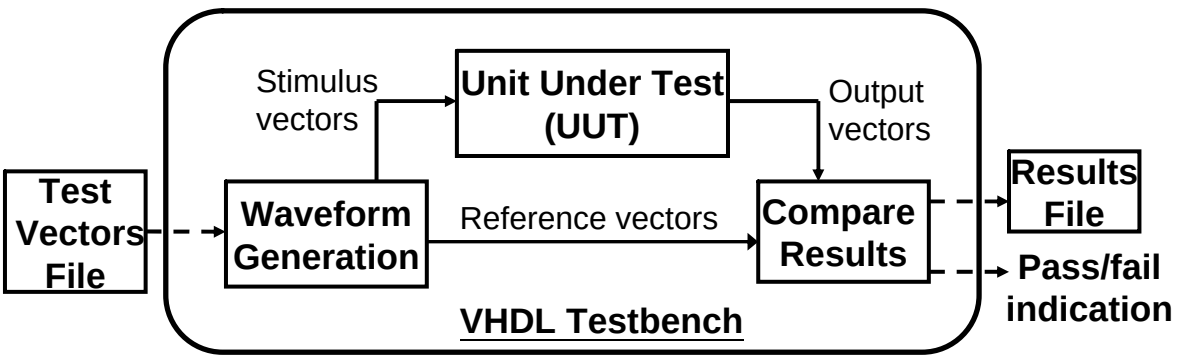
```
seq: process(clock,reset,currentstate)
begin
      if(reset='1') then
            currentstate<=st0;
      elsif clock'event and clock='1' then
            currentstate<=nextstate;
      end if;
      case currentstate is
            when st0=>y<=0;
            when st1=>y<=1;
            when st2=>y<=2;
            when st3=>y<=3;
       end case;
end process seq;
```



Testbench

Testbench

- A testbench is normally written in the same HDL language.
- There is wide variety of ways in which testbench may be coded.



Objectives

- Instantiate the hardware model under test.
- Generate stimulus waveforms
- Generate expected waveforms
- Provide a pass or fail indication automatically.

Advantage of Testbench in VHDL

- No need to learn other simulation tool or language.
- Can transportable across different design tools.
- Can be exploited to the full in a testbench.

Vector Generation

- Generate them “on-the-fly” form within a testbench.
- Read vectors stored as constants in an array.
- Read vectors stored in a separate system file.

Vectors Generated “on-the-fly”

- Use continuous loops for repetitive signals, such as clocks.
- Use simple assignments for signals with few transitions, such as resets.
- Use relative or absolute time generated signals.
- Use loop constructs for repetitive signal patterns

Entity of Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_textio.all;
USE std.textio.all;

ENTITY testbench IS
END testbench;
```


Unit under Test

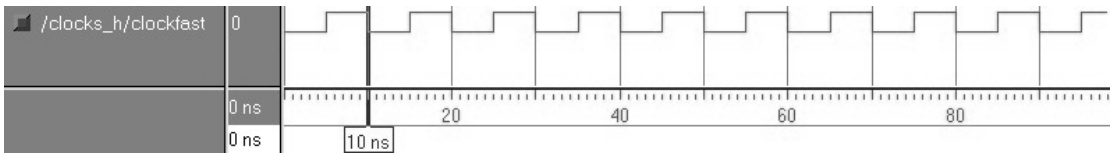
```
component f_adder
        port (a,b,c: in std_logic;
                sum,carry: out std_logic);
end component;
signal a_i,b_i,c_i: std_logic;
signal sum_o,carry_o: std_logic;

...

u0: f_adder port map (a_i,b_i,c_i,sum_o,carry_o);
```

Generating Fast Clock Signals

```
constant FastClock  : integer := 100; -- 100MHz
constant FastClockPeriod : TIME := 1 us/FastClock;
signal ClockFast : std_logic := '0';
```

...

ClockFast <= not ClockFast **after FastClockPeriod / 2 ;**

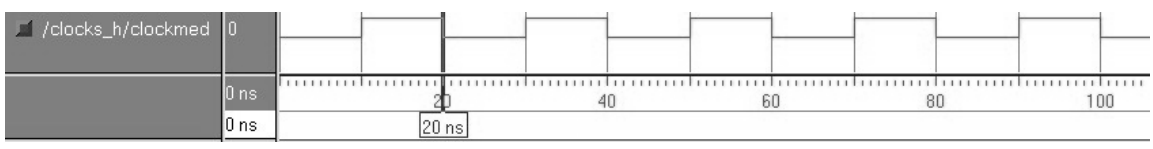


Generating Medium Clock Signals

```
constant MedClock  : integer :=  50; -- 50MHz
constant MedClockPeriod  : TIME := 1 us/MedClock;
signal ClockMed : std_logic := '0';
```

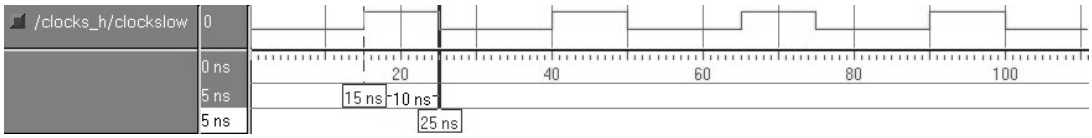
...

ClockMed <= '1' **after (MedClockPeriod * 0.5)**
when ClockMed = '0' else
'0' **after (MedClockPeriod * 0.5)**;



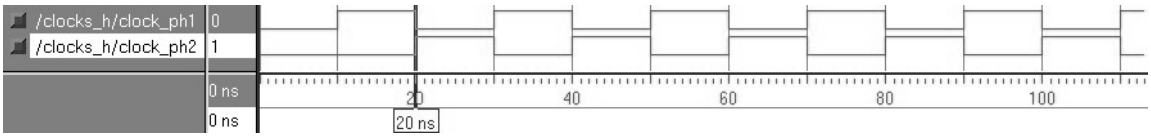
Generating Slow Clock Signals

```
constant SlowtClock : integer :=  40; -- 40MHz
constant SlowClockPeriod : TIME := 1 us/SlowtClock;
signal ClockSlow : std_logic := '0';
...
process
begin
   wait for (SlowClockPeriod * 0.60);
      ClockSlow <= '1';
   wait for (SlowClockPeriod * 0.40);
      ClockSlow <= '0';
end process;
```



Generating Phase Clock Signals

```
constant MedClock : integer :=  50; -- 50MHz
constant MedClockPeriod  : TIME := 1 us/MedClock;
signal Clock_ph1,Clock_ph2 : std_logic := '0';

...

Clock_ph1 <= not Clock_ph1 after MedClockPeriod /2;
Clock_ph2 <= Clock_ph1'delayed(MedClockPeriod/2);
```

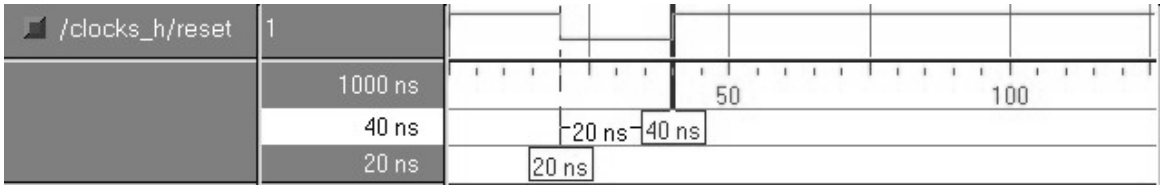


Generating Clock Signals with Wait

```
constant RunClock  : integer := 100; -- 100MHz
constant runtime : TIME := 1 us/RunClock;
signal clk,stop   : std_logic := '0';
...
process
begin
   loop
      wait for runtime/2;
      clk <= not clk;
      exit when stop = '1';
   end loop;
   wait;
end process;
```


Generating Signals with Few Transitions

```
signal reset : std_logic;

...

reset <= '1', '0' after 20 ns, '1' after 40 ns;
```



Reset with Wait

```
constant RunClock  : integer := 100; -- 100MHz
constant runtime : TIME := 1 us/RunClock;
signal reset  : std_logic := '1';
...
process
begin
     wait for (runtime/4);
     reset <= '0';
     wait;
end process;
```


Relative Time Generated Signals

```
   signal A,B : std_logic;
   constant runtime : TIME := 50 ns;
   ...
 process
 begin
   A <= '0';
   B <= '0';
   wait for runtime;
   B <= '1';
   wait for runtime;
   A <= '1';
   B <= '0';
   wait for runtime;
   B <= '1';
   wait for runtime;
 End process;
```

Absolute Time Generated Signals

```
signal A,B : std_logic;
constant runtime : TIME := 50 ns;
 …
A <= '0',
     '1'  after (runtime*2);

B <= '0',
     '1' after (runtime*1),
     '0' after (runtime*2),
     '1' after (runtime*3);
```


Repetitive Stimulus Using Loops

```
signal a,b,c,sum,carry,stop : std_logic := '0';
signal cs : std_logic_vector(1 downto 0);
...
process
Begin
   cs <= conv_std_logic_vector((conv_integer(a) +
         conv_integer (b) + conv_integer (c)),2);
   for i in 0 to 7 loop
      (c, a, b) <= conv_std_logic_vector (i,3);
      wait for 10 ns;
      assert carry&sum = cs
      report "error result"
      severity error;
   end loop;
   stop <= '1';
   wait;
end process;
```

Testbench of Full_adder

Vectors Generated

```
use std.textio.all;
use ieee.std_logic_textio.all;
...
file output_file : TEXT IS OUT “out.data";
...
process
  variable BufLine : LINE;
begin
  wait for 15 ns;
  write(BufLine,string’(“(A,B)=”));
  write(BufLine,a);
  write(BufLine,b);
  writeline(output_file, BufLine);
end process;
```

```
(A,B)=00
(A,B)=01
(A,B)=10
(A,B)=11
```


Vectors Stored in an Array

```
type MemVecArr is array (0 to 2) of bit_vector(7 downto 0);
Signal OperateData : bit_vector(7 downto 0);
constant MemArr : MemVecArr :=(0 => "00000000",
                               1 => "00000001",
                               2 => "00000011");
...
process
begin
    for N in MemVecArr’range loop
       OperateData <= MemArr(N);
       wait for 10 ns;
    end loop;
end process;
```

Reading Test Vector System Files

```
file input_file : TEXT IS IN “in.data";
...
process
  variable BufLine : LINE;
  variable a,b,c : bit;
begin
  while not(ENDFILE(input_file)) loop
     readline(input_file, BufLine);
     read(BufLine,a);
     read(BufLine,b);
     read(BufLine,c);
     wait for 10 ns;
  end loop;
end process;
```

a	b	c
000		
001		
010		
011		
100		
101		
110		
111		

Report Message

- **report “string”;**
- **report T’IMAGE(X);**
- **report “string” & T’IMAGE(X);**

```
process
    Variable error_num  : integer;
begin
    error_num := 10;
    report “Error number is :”;
    report  integer’IMAGE(error_num);
    report  “Error number is :” & integer’IMAGE(error_num);
    wait for 10 ns;
end process;
```

Report Hexadecimal String (1/2)

```
process
    Variable hex_value  : std_logic_vector(15 downto 0);
begin
    hex_value := X”abcd”;
    report   “Hex Value is : ” & to_hex(hex_value) & “;”;
    wait for 10 ns;
end process;
```

report message:

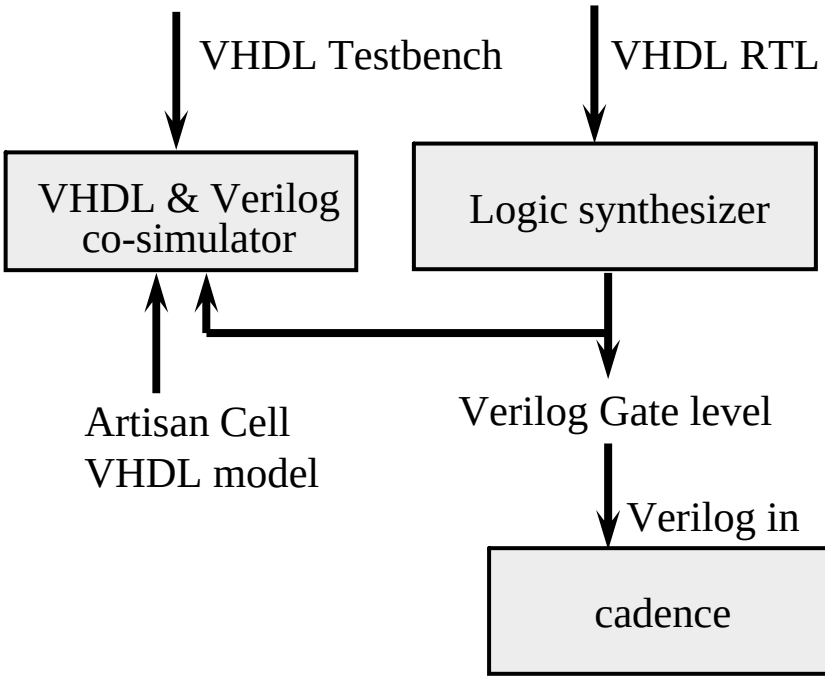
Hex Value is : abcd;

Report Hexadecimal String (2/2)

```
function to_hex(S : std_logic_vector) return string is
  constant hex_index : string(1 to 16) :=
                        ('0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f');
  variable result : string(S’length/4  downto 1);
  variable temp : std_logic_vector(3 downto 0);
begin
  for i in 0 to S’length-1 loop
    for j in i*4 to (i*4)+3 loop
      temp(j-(i*4)) := S(j);
    end loop;
    if temp >= "0000" and temp <= "1111" then
      result(i+1) := hex_index(to_integer(unsigned(temp))+1);
    else
      result := (others => 'X');
      return result;
    end if;
  end loop;
  return result;
end to_hex;
```

Gate-Level Simulation

Verilog Out Design Flow



Gate-Level Simulation for Verilog

- Apply for Cell-base Design Kit
- Prepare VHDL Cell Library
- Compile VHDL Cell Library
- Start on gate-level co-simulation

Apply for Cell-base Design Kit

- http://www.cic.edu.tw/chip_design/design_petition.html
  - CIC 0.25um Cell-Based Design kit V1.0 Release
  - CIC 0.35um Cell-Based Design Kit (TSMC/TSMC) Release

Prepare VHDL Cell Library

- **0.25 um technology**
  - CIC_CBDK25_V1/CIC/VHDL/CORE/tsmc25.vhd
- **0.35 um technology**
  - CBDK035_TSMC_TSMC/CIC/VHDL/*.vhd

Compile VHDL Cell Library .25um

- **Invoke ModelSim**
  - *vsim&*
- **Create a library – artisan_lib25**
  - *vlib **artisan_lib25***
  - *vmap artisan_lib25 **artisan_lib25***

Compile VHDL Cell Library .25um

- **Compile tsmc25.vhd to artisan_lib25 library**
  - Select **Compile -> Compile...** open “Compile HDL Source Files window”.
  - Select “artisan_lib25” in the **Library form.**
  - Compile tsmc25.vhd

Compile VHDL Cell Library .35um

- **Invoke ModelSim**
  - *vsim&*
- **Create a library – vital**
  - *vlib **vital***
  - *vmap vital **vital***
- **Compile** Vtable.vhd **to** vital **library**
  - Select **Compile -> Compile...**
  - Select “vital” in the **Library form**
  - Compile Vtable.vhd

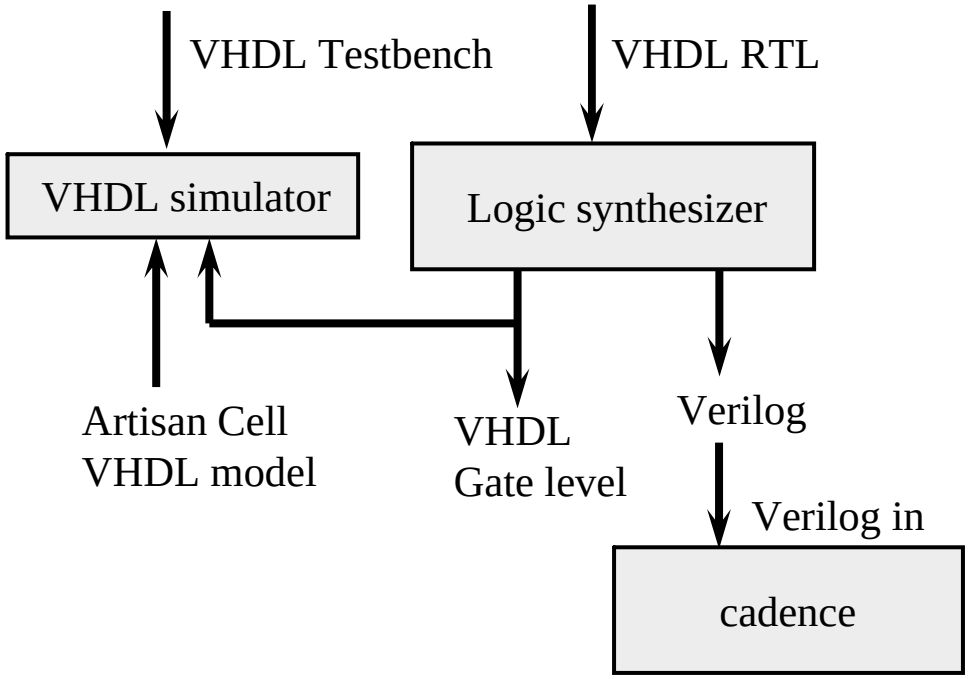
Compile VHDL Cell Library .35um

- **Create a library – tsmc_lib35**
  - *vlib **tsmc_lib35***
  - *vmap tsmc_lib35 **tsmc_lib35***
- **Compile *.vhd to** tsmc_lib35 **library**
  - Select **Compile -> Compile...**
  - Select “tsmc_lib35” in the **Library form**
  - Compile *.vhd
    (can ignore the warnings.)

Start on Gate-Level co-Simulation

- **Load Design (I)**
  - Select Simulate -> Simulate... open “Simulate window”.
  - Select “Libraries”label, add the path of artisan_lib25 in the Search Libraries[-L].
  - Select “Design” label and load design.
- **Load Design (II)**
  - vsim –L artisan_lib25 work.design
  - vsim –L compass_lib work.design

VHDL Out Design Flow



Gate-Level Simulation for VHDL

- **Modify *.synopsys_dc.setup* of Synopsys**
- **Apply for Cell-base Design Kit**
- **Prepare VHDL Cell Library**
- **Compile VHDL Cell Library**
- **Start on VHDL gate-level simulation**

Modify *.synopsys_dc.setup*

- **0.25 um technology**
  - vhdlout_use_packages={"IEEE.std_logic_1164", "**artisan_lib25.prim**" }
- **0.35 um technology**
  - vhdlout_use_packages={"IEEE.std_logic_1164",“**tsmc_lib35.vcomponents**" }
- **In the VHDL gate-level code, you can see the following description:**
  - use artisan_lib25.prim.all
  - use tsmc_lib35.vcomponents.all

Apply for Cell-base Design Kit

- **http://www.cic.edu.tw/chip_design/design_petition.html**
  - CIC 0.25um Cell-Based Design kit V1.0 Release
  - CIC 0.35um Cell-Based Design Kit (TSMC/TSMC) Release

Prepare VHDL Cell Library

- **0.25 um technology**
  - CIC_CBDK25_V1/CIC/VHDL/CORE/tsmc25.vhd
- **0.35 um technology**
  - CBDK035_TSMC_TSMC/CIC/VHDL/*.vhd

Compile VHDL Cell Library .25um

- **Invoke ModelSim**
  - *vsim&*
- **Create a library – artisan_lib25**
  - *vlib **artisan_lib25***
  - *vmap artisan_lib25 **artisan_lib25***

Compile VHDL Cell Library .25um

- **Compile tsmc25.vhd to artisan_lib25 library**
  - Select **Compile -> Compile...** open “Compile HDL Source Files window”.
  - Select “artisan_lib25” in the **Library form.**
  - Compile tsmc25.vhd

Compile VHDL Cell Library .35um

- **Invoke ModelSim**
  - *vsim&*
- **Create a library – vital**
  - *vlib **vital***
  - *vmap vital **vital***
- **Compile** Vtable.vhd **to** vital **library**
  - Select **Compile -> Compile...**
  - Select “vital” in the **Library form**
  - Compile Vtable.vhd

Compile VHDL Cell Library .35um

- **Create a library – tsmc_lib35**
  - *vlib **tsmc_lib35***
  - *vmap tsmc_lib35 **tsmc_lib35***
- **Compile *.vhd to** tsmc_lib35 **library**
  - Select **Compile -> Compile...**
  - Select “tsmc_lib35” in the **Library form**
  - Compile *.vhd
    (can ignore the warnings.)

Start on Gate-Level co-Simulation

- **Load Design (I)**
  - Select Simulate -> Simulate... open “Simulate window”.
  - Select “Design” label and load design.
- **Load Design (II)**
  - vsim work.design

How to do gate-level simulation

VHDL 版本

1.Synopsys 合成前：

- 編輯 .synopsys_dc.setup ,如果是.25 製程，則修改或加入一行如下：
  vhdlout_use_packages={"IEEE.std_logic_1164", "**artisan_lib25.prim**"}
- 如果是.35 製程，則修改或加入一行如下：
  vhdlout_use_packages={"IEEE.std_logic_1164","**tsmc_lib35.vcomponents**"}
- 合成出的 vhdl gate-level 電路，可看到以下描述：
  use artisan_lib25.prim.all; 或 **use tsmc_lib35.vcomponents.all;**

2.申請 CIC Cell-Base Design Kit。

- http://www.cic.edu.tw/chip_design/design_petition.html

3.得到所需 VHDL 版本的 Cell Library：

- .25 製程
  路徑：CIC_CBDK25_V1/CIC/VHDL/CORE/tsmc25.vhd
- .35 製程
  路徑：CBDK035_TSMC_TSMC/CIC/VHDL/*.vhd

4.Compile .25 製程 VHDL Library：

- Invoke modelsim
  % vsim&
- Create a library – **artisan_lib25**
  Modelsim> vlib **artisan_lib25**
  Modelsim> vmap artisan_lib25 **artisan_lib25**
- Compile tsmc25.vhd to **artisan_lib25** library
  選擇 Compile -> Compile...開啓 Compile HDL Source Files window.
  在 Library form 中選擇 **artisan_lib25**.
  開始 compile tsmc25.vhd.

5.Compile .35 製程 VHDL Library：

- Invoke modelsim
  % vsim&
- Create a library – **vital**
  Modelsim> vlib **vital**
  Modelsim> vmap vital **vital**
- Compile Vtable.vhd to **vital** library
  選擇 Compile -> Compile...開啓 Compile HDL Source Files window.
  在 Library form 中選擇 **vital**
  開始 compile Vtable.vhd.，
- Create a library – **tsmc_lib35**
  Modelsim> vlib **tsmc_lib35**
  Modelsim> vmap tsmc_lib35 **tsmc_lib35**

- Compile *.vhd to **tsmc_lib35** library
  選擇 Compile -> Compile...開啓 Compile HDL Source Files window.
  在 Library form 中選擇 **tsmc_lib35**.
  開始 compile *.vhd.
  (有 warning 可以忽略)

6.開始 gate-level simulation for modelsim。

Verilog 版本

1.申請 CIC Cell-Base Design Kit。

- http://www.cic.edu.tw/chip_design/design_petition.html

2.得到所需 Verilog 版本的 Cell Library：

- .25 製程
  路徑：CIC_CBDK25_V1/CIC/Verilog/tsmc25.v
- .35 製程
  路徑：CBDK035_TSMC_TSMC/CIC/Verilog/tcb773p.v

3.Compile .25 製程 Verilog Library：

- Invoke modelsim
  % vism&
- Create a library – **artisan_lib25**
  Modelsim> vlib **artisan_lib25**
  Modelsim> vmap artisan_lib25 **artisan_lib25**
- Compile tsmc25.v to **artisan_lib25** library
  選擇 Compile -> Compile...開啓 Compile HDL Source Files window.
  在 Library form 中選擇 **artisan_lib25**.
  開始 compile tsmc25.v.

4.Compile .35 製程 Verilog Library：

- Invoke modelsim
  % vism&
- Create a library –**tsmc_lib35**
  Modelsim> vlib **tsmc_lib35**
  Modelsim> vmap tsmc_lib35 **tsmc_lib35**
- Compile tcb773p.v to **tsmc_lib35** library
  選擇 Compile -> Compile...開啓 Compile HDL Source Files window.
  在 Library form 中選擇 **tsmc_lib35**.
  開始 compile tcb773p.v

5.Load Design

Simulate -> Simulate…開啓 Simulate window.
選擇 Libraries 標籤，在 Search Libraries[-L]中加入 **artisan_lib25** 或 **tsmc_lib35** 的路徑.
選擇 Design 標籤，選取欲模擬的電路，再按 Load.

6.開始 gate-level simulation for modelsim。

How to dump VCD file

1.The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes.

2.First, load design you want to simulate.

- Simulate -> Simulate…

3.Assign VCD command on command field.

- Open a VCD file
  VSIM> ***vcd file*** *dump.vcd*
- Add all signal to VCD file
  VSIM>***vcd add*** *–r **

4.Assgin simulate command on command field.

- For example: **force**、**run**…
- But **add wave** is unnecessary.

5.When simulation is completion, end simulation is necessary.

- Simulate -> End Simulation…

6.When you complete above steps, you can get the **dump.vcd**.

7.Translate VCD file to WLF file

*%**vcd2wlf** ???.vcd ???.wlf*

8.View WLF file.

*%**vsim** –view ???.wlf &*
VSIM>***add wave*** ***

9.Other VCD command reference.

- /usr/mentor/fpgadv50_ss/Modeltech/docs/pdf/se_cmds.pdf

How to dump FSDB file

1.Debussy supports a new and open FSDB file format that has the following advantages over the standard VCD file format：

- An FSDB file is much more compact than the standard VCD file. Typically, an FSDB file is about 5 to 50 times smaller than a VCD file.
- The simulation run time for an FSDB file is faster than that for a VCD file. With FSDB files, Debussy displays waveform and back-annotated signal values faster.

2.Prepare data file

- **novas.vhd** that provided by Debussy.
  /usr/debussy/share/PLI/modelsim_fli/SOLARIS2/**novas.vhd**

3.Compile novas.vhd.

%vlib novas

*%vcom –work novas /usr/debussy/share/PLI/modelsim_fli/SOLARIS2/**novas.vhd***

4.Set the path of **novas_fli.so** to your library path.

%setenv LD_LIBRARY_PATH

/usr/debussy/share/PLI/modelsim_fli/SOLARIS2/:$LD_LIBRARY_PATH

5.Start your simulation.

- Invoke ModelSim
  %vsim&
- Compile your design
  Modelsim>*vcom userdesign.vhd*
- Load your design to simulate (“**userdesign**” is the name of top entity or comfiguration)
  Modelsim>*vsim userdesign **novas.novas***

6.Assign FSDB command on command field.

- Open a FSDB file
  VSIM>*fsdbDumpfile dump.fsdb*
- Set the depth of design you want to view signal
  VSIM>*fsdbDumpvars 2 entity_name*
  (“2” is the depth. “entity_name” is the name of top entity.)

7.Run your simulation.

VSIM>*run 100000ns*

8.End simulation to save the **dump.fsdb.**

- Simulate -> End Simulation…

9.Use nWave of Dubussy to view FSDB file.

- Invoke nWave

%nWave&

- Please refer the Debussy's training course to get more information.

Other method：

10. Reference the package *pkg* defined in novas.vhd. Add the following line to your VHDL design：

    library novas;

    use novas.pkg.all;

11. Compile **novas.vhd** to the **novas** library.

    Modelsim>*vlib novas*

    Modelsim>*vmap novas novas*

    Modelsim>*vcom –work novas novas.vhd*

12. Add the routine in the package *pkg* to your design. For example：

    - Open a FSDB file

      fsdbDumpfile("dump.fsdb");

    - Set the depth of design you want to view signal

      fsdbDumpvars(2,"entity_name");

      ("2" is the depth. "entity_name" is the name of top entity.)

13. When your simulation complete and **end simulaton**, you can get **dump.fsdb**.

*If your design is Verilog code, you also get dump.fsdb file from step 2 to step 8.

VHDL LAB –ModelSim

Setup your environment

1. Log in WS with username/password shown in the KB
2. %/bin/rm –rf *
3. %cicsetup
4. Log out and then log in again
5. Copy setup files for VHD simulator (Mentor ModelSim) and logic synthesyzer (Synopsys's Design compiler):

   %cp –r /cad2/lab/VHDL/lab-modelsim ~/VHDL

6. ***%source /usr/mentor/cic_setup/modelsim.csh***

LAB1

1. %cd ~/VHDL/lab1
2. Set up working directory for modelsim

   %cp ~/VHDL/modelsim.ini .
3. Invoke ModelSim

   %vsim&
4. Creat a work library.

   <File> -> <New> -> <Library…>

   a. Enable "a new library and a logical mapping to it" in the Create New Library window.
   b. Fill in "work" in Library Name form.
   c. Click <OK>.
5. Compile VHDL code

   <Compile> -> <Compile…>

   a. Use mouse to select source file "multi.vhd" in UP window.
   b. Click <Compile> to compile the VHDL code.
   c. Click <Done> to close Compile window.
6. Execute simulation

   <Simulate> -> <Simulate…>

   a. Select the "Design" slice.
   b. Select the "cfg1_multiplier" in work library of middle window.
   c. Click "Load" to load cfg1_multiplier
7. Assign command on command field

   >add wave /multiplier/a /multiplier/b /multiplier/z

   >force /multiplier/a 1011

   >force /multiplier/b 0011

   >run 500

   >force /multiplier/b 1010

   >run 500
8. Write step 7 to test.do. Using command script include test.do

   >do test.do
9. End Simulation to save results

   <Simulate> -> <End Simulation>
10. Review waveform

    %vsim –view vsim.wlf

    >add wave *

LAB2

1. %cd ~/VHDL/lab2
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Invoke Modelsim
   %vsim&
4. Creat a new library
   >vlib work
5. map the new library to the work library
   >vmap work work
6. Revise f_adder.vhd and run simulation
7. Compile full adder
   >vcom f_adder.vhd testfixture_fa.vhd
8. Execute simulatioin
   >vsim -wlf fadder.wlf work.cfg_testfa
9. Add all signal to wave form viewer.
   >add wave *
10. Run simulation
    >run -all
11. End Simulation to save result.
    <Simulate> -> <End Simulation>
12. View the f_adder.do and study.
    %vi f_adder.do
13. Revise add4.vhd and run simulation
    >vcom add4.vhd testfixture_add4.vhd
    >vsim -wlf add4.wlf work.cfg_testadd4
    >add wave *
    >run -all

LAB3 Co-simulation：VHDL call Verilog

1. %cd ~/VHDL/lab3
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Prview adder4.vhd and fa.v
4. Revise adder4.vhd and run simulation
5. Simulate 4bit-adder
   %vsim&
   >vlib work
   >vmap work work
   >vlog fa.v
   >vcom adder4.vhd testfixture_add4.vhd
   >vsim –wlf add4.wlf cfg_testadd4
   >add wave *
   >run -all

LAB4 Co-simulation：Verilog call VHDL

1. %cd ~/VHDL/lab4
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Prview adder4.v and fa.vhd
4. Simulate 4bit-adder
   %vsim&

>vlib work
>vmap work work
>vcom fa.vhd
>vlog adder4.v
>vcom testfixture_add4.vhd
>vsim –wlf add4.wlf cfg_testadd4
>add wave *
>run -all

LAB5

1. %cd ~/VHDL/lab5
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Revise alu.vhd and test_alu.vhd
   % vi alu.vhd
   % vi test_alu.vhd
4. Simulate ALU
   %vsim&
   >vlib work
   >vmap work work
   >vcom alu.vhd test_alu.vhd
   >vsim –wlf alu.wlf work.cfg_testALU
   >add wave *
   >run -all
5. Revise ALU.vhd to concurrent version

LAB6 FSM

1. %cd ~/VHDL/lab6
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Revise fsm.vhd
   % vi fsm.vhd
4. Simulate FSM
   %vsim&
   >vlib work
   >vmap work work
   >vcom fsm.vhd test_fsm.vhd
   >vsim –wlf fsm.wlf work.cfg_testFSM
   >add wave *
   >run -all

LAB7 Gate-level implementation and simulation for VHDL

1. %cd ~/VHDL/lab7
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Set up environment for Synopsys Design Compiler
   %cp ~/VHDL/.synopsys_dc.setup .
   %source /usr/synopsys/CICuser_setup/synthesis.csh
4. Run Synopsys Design Compiler
   %da&
   <Setup>-><command Window>

5. Read VHDL RTL to DC
   >Read –f vhdl alu.vhd
6. Choose Design
   >current_design alu
7. Perform logic synthesis
   >compile
8. Create schematic
   >create_schematic -herarchy
9. Change naming rule vhdl
   >change_names –rule vhdl
10. Output vhd gate-level netlist and SDF files
    >write –f vhdl –h –o alu_gate.vhd
    >write_sdf –version 2.1 alu.sdf
    !!!You may write step 5-10 to a dc file.
    Then type "***include ???.dc***" in the command field.to run all commands.
11. Simulate ALU in GATE level
    %vsim&
    >vlib work
    >vmap work work
    >vcom alu_gate.vhd test_alu.vhd
    >vsim –wlf alu.wlf work.cfg_testALU
    >add wave *
    >run -all
    Have Error Message!!!.
    Because to "ALU_test.vhd" use the RTL architecture in "ALU.vhd".
12. Revise "test_alu.vhd" to let it use the gate-level **architecture** in "alu_gate.vhd"
13. Re-compile the "test_alu.vhd" and re-run the simulation
    ALU error?
    Because we forget to annotate the timing information into the design in "vsim" phase.
14. Add net delay to simulation
    >vsim **–sdftyp /u=alu.sdf** –wlf alu.wlf work.cfg_testALU
    >add wave *
    >run -all

LAB8 Gate-level implementation and simulation for Verilog

1. %cd ~/VHDL/lab8
2. Set up working directory for modelsim
   %cp ~/VHDL/modelsim.ini .
3. Set up environment for Synopsys Design Compiler
   %cp ~/VHDL/.synopsys_dc.setup .
   %source /usr/synopsys/CICuser_setup/synthesis.csh
4. Run Synopsys Design Compiler
   %da&
   <Setup>-><command Window>
5. Read VHDL RTL to DC
   >Read –f vhdl alu.vhd
6. Choose Design
   >current_design alu

7. Perform logic synthesis
   >set_fix_multiple_port_nets -all -buffer_constants
   > compile -map_effort medium
8. Change naming rule verilog
   > change_names -rule verilog -hierarchy
9. Output verilog gate-level netlist and SDF files
   > write -f verilog -h -o alu_gate.v
   > write_sdf -version 2.1 alu.sdf
   !!!You may write step 5-10 to a dc file.
   Then type "***include ???.dc***" in the command field.to run all commands..
10. Simulate ALU in GATE level
   %vsim&
   >vlib work
   >vmap work work
   >vlog alu_gate.v
   >vcom test_alu.vhd
   >vsim –L ../lab7/artisan_lib25 -sdftyp /U=alu.sdf -wlf alu.wlf \
   work.cfg_testalu
   >add wave *
   >run -all