VHDL For Logic Synthesis (2)
(Synthesizable Code)

Introduction to CAD

Naehyuck Chang

naehyuck@snu.ac.kr

Seoul National University
Dept. of Computer Engineering

Introduction

- Hardware implementation of VHDL code depends on
  - Coding conventions
  - Fitter technology
  - Optimization option
  - Nature of application
- Not all design can be synthsizable.

Introduction (contd.)

- Not synthesizable VHDL code
  - High-level performance model
  - Test benches
  - Hardware/software mixed model
- Design for synthesis does require additional constraints.
  - Gated clock, timing characteristics, unbounded conditions, enumerated types, *etc*.

Introducton (contd.)

- Optimization
  - Different target architecture: PLD versus FPGA
- Using constant or simple one-bit data types whenever possible.

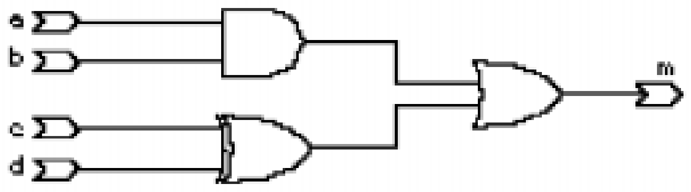
Combinational Logic

- Logical operators
  - `and`, `or`, `nand`, `not`, `xor` and `not`
  - Defined for bit and Boolean types.
  - Also defined for **std_logic**, **std_ulogic** and **std_logic_vector** when **ieee** library and **std_logic_1164** package are included.

Combinational Logic (contd.)

- Example 1

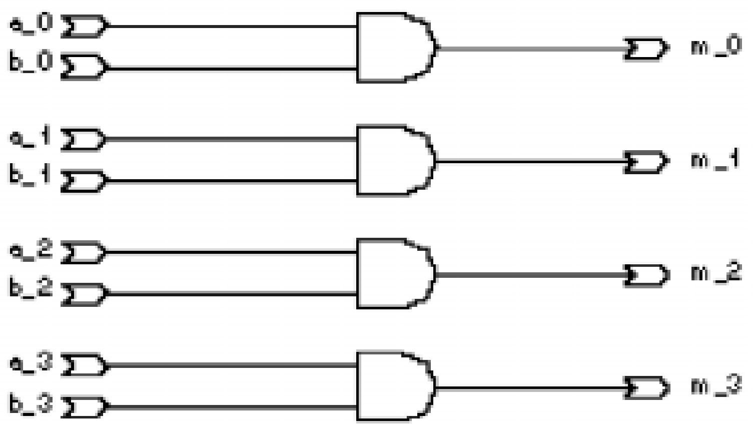
```
entity logical_ops_1 is
    port (a, b, c, d: in bit;  m: out bit);
end logical_ops_1;

architecture example of logical_ops_1 is
    signal e: bit;
begin
    m <= (a and b) or e; --concurrent signal assignments
    e <= c xor d;
end example;
```



Combinational Logic (contd.)

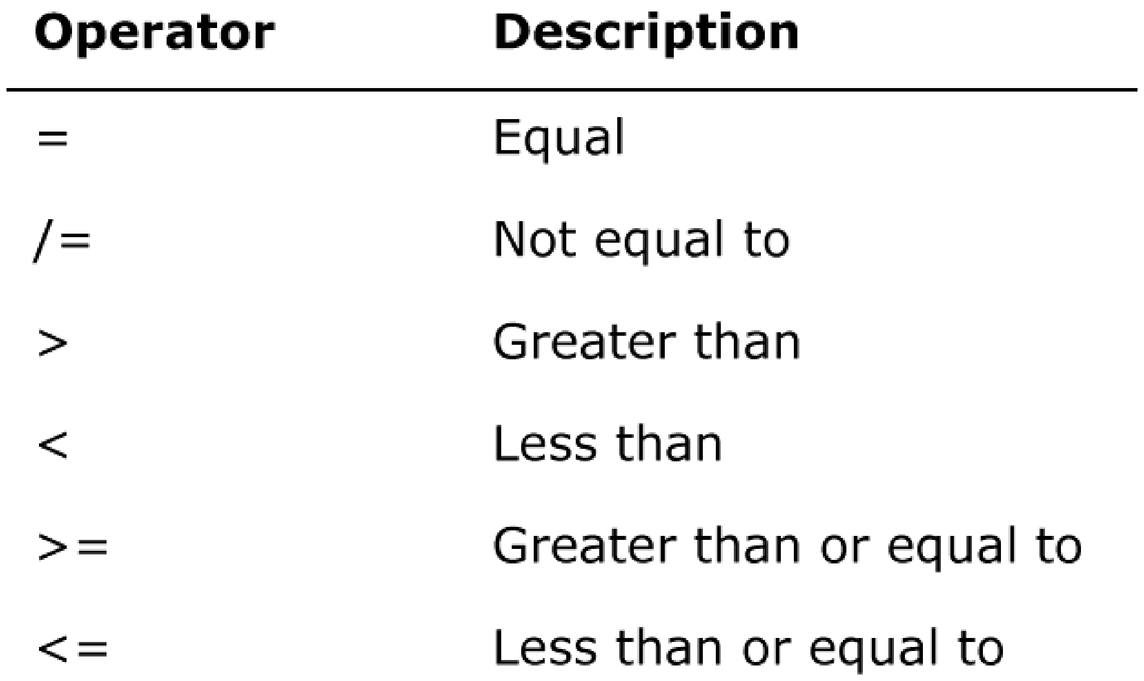
- Example 2

```
entity logical_ops_2 is
    port (a, b: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end logical_ops_2;

architecture example of logical_ops_2 is
begin
    m <= a and b;
end example;
```



Combinational Logic (contd.)

- Relational operators



Operator	Description
=	Equal
/=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

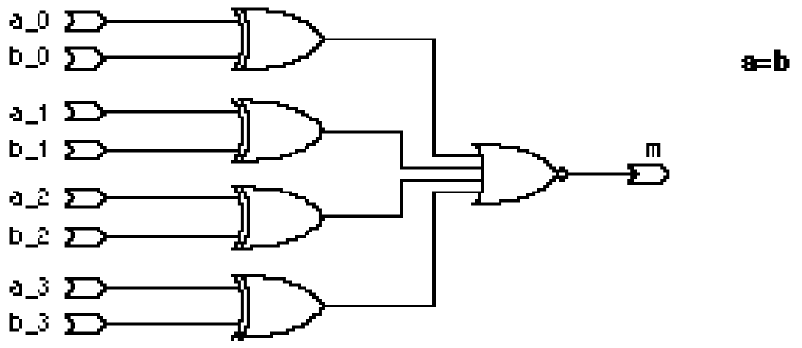
Combinational Logic (contd.)

- Resulting type is Boolean.
- Equal operators are defined for all VHDL data types.
- Magnitude operators are defined for numeric types, enumerated types and some array.
- Overloaded version is defined for **bit_vector** and **std_logic_vector** types.

Combinational Logic (contd.)

- Example 1

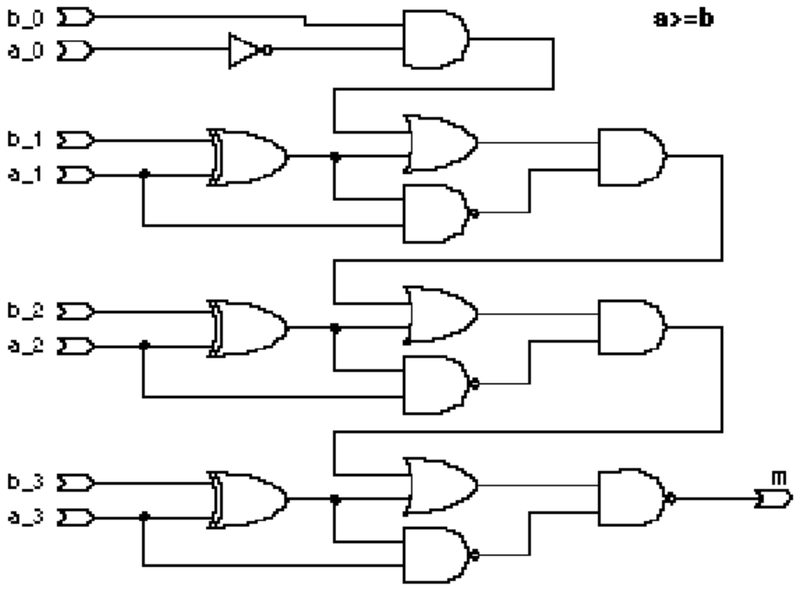
```
entity relational_ops_1 is
    port (a, b: in bit_vector (0 to 3); m: out Boolean);
end relational_ops_1;
architecture example of relational_ops_1 is
begin
    m <= a = b;
end example;
```



Combinational Logic (contd.)

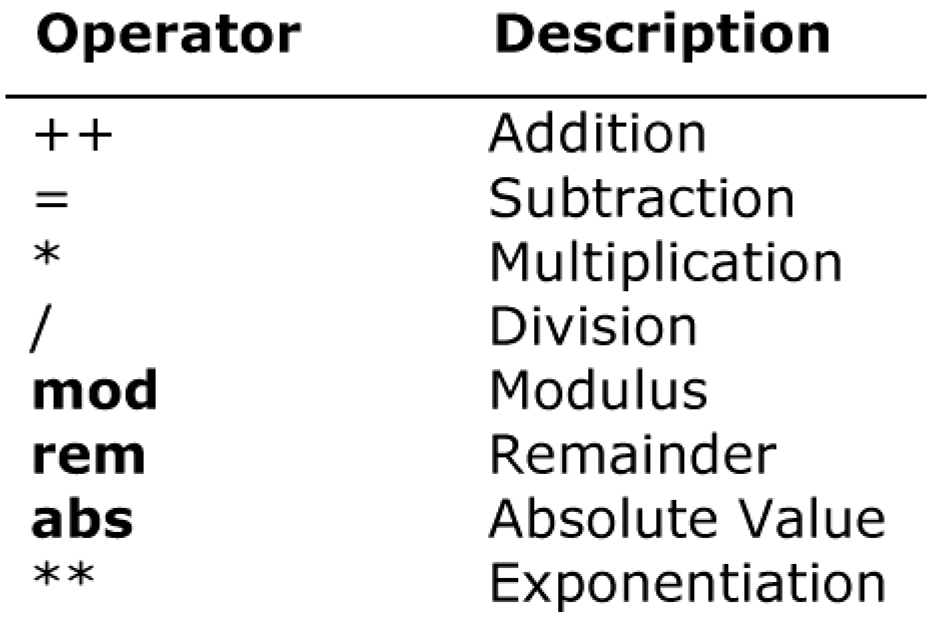
- Example 2

```
entity relational_ops_2 is
    port (a, b: in integer range 0 to 3; m: out Boolean);
end relational_ops_2;
architecture example of relational_ops_2 is
begin
    m <= a >= b;
end example;
```



Combinational Logic (contd.)

- Arithmetic operators



Operator	Description
++	Addition
=	Subtraction
*	Multiplication
/	Division
mod	Modulus
rem	Remainder
abs	Absolute Value
**	Exponentiation

Combinational Logic (contd.)

- Operators are defined for numeric types.
- Overloaded version defines (`+`,`-`) operators for **bit_vector** and **std_logic_vector** in the package **bit_ops** and **std_logic_ops**.
- (`+`, `-`) operators are somewhat expensive.
- (`*`, `/`, `mod`, `rem`) operators are **extremely** expensive.
  - Special optimization for a constant and an even power of 2.

Combinational Logic (contd.)

– (`abs`) operator is cheap.

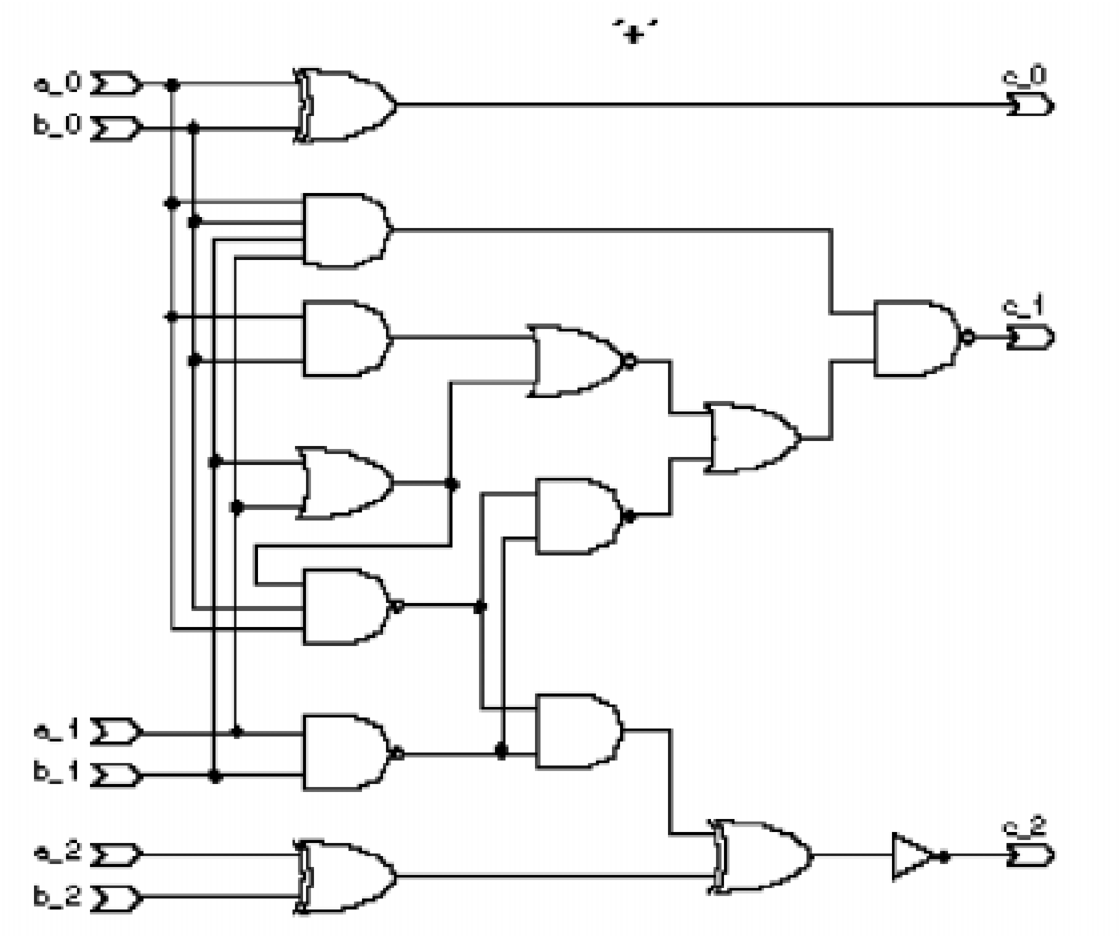
– (`**`) operator only supports constant arguments.

Combinational Logic (contd.)

- Example

```
package example_arithmetic is
    type small_int is range 0 to 7;
end example_arithmetic;

use work.example_arithmetic.all;

entity arithmetic is
    port (a, b: in small_int; c: out small_int);
end arithmetic;
architecture example of arithmetic is
begin
    c <= a + b;
end example;
```

Combinational Logic (contd.)



Combinational Logic (contd.)

- Shift operators

Operator	**Description**
sll	Shift Left Logical
srl	Shift Right Logical
sla	Shift Left Arithmetic
sra	Shift Right Arithmetic
rol	Rotate Left Logical
ror	Rotate Right Logical

Combinational Logic (contd.)

– Operators are defined for bit and Boolean types.

– Overloaded version defines shift operators for **bit_vector** and **std_logic_vector** in the package **std_logic_ops**.

– Operators are not expensive the right operand is constant.

– Operators are quite expensive when the right operand depends on a signal.

Conditional Logic

- Conditional logic is combinational logic that implements a multiplexer-like function.
- Two forms
  - Conditional signal assignment: if statement
  - Selected signal assignment: case statement

Conditional Logic (contd.)

- Concurrent statement
  - Conditional signal assignment

```
entity control_stmts is
    port (a, b, c: in Boolean;  m: out Boolean);
end control_stmts;

architecture example of control_stmts is
begin
    m <= b when a else  c;
end example;
```

Conditional Logic (contd.)

– IEEE standard 1076-1993 describes that **else** clause is optional
    - Without **else** clause, the resulting circuit mostly introduces a latch, which many not be the desired circuit.

Conditional Logic (contd.)

– Selected signal assignment

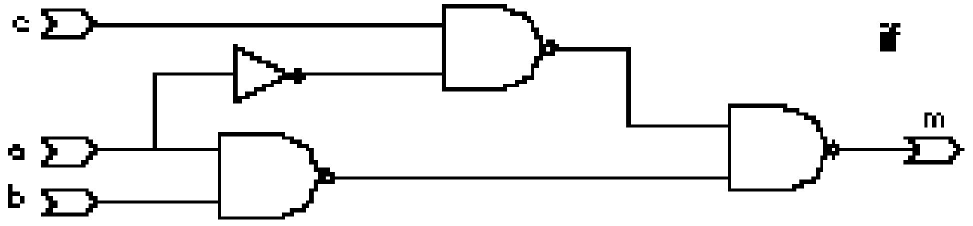
```
entity control_stmts is
    port (sel: bit_vector (0 to 1); a,b,c,d: bit;
          m: out bit);
end control_stmts;

architecture example of control_stmts is
begin
    with sel select
        m <= c    when b"00",
        m <= d    when b"01",
        m <= a    when b"10",
        m <= b    when others;
end example;
```

Conditional Logic (contd.)

- Sequential statement
  - If statement
    - Must evaluate a Boolean type.

Conditional Logic (contd.)

```
entity control_stmts is
    port (a, b, c: in Boolean;  m: out Boolean);
end control_stmts;
architecture example of control_stmts is
begin
    process (a, b, c)
        variable n: Boolean;
    begin
        if a then
            n := b;
        else
            n := c;
        end if;
        m <= n;
     end process;
 end example;
```

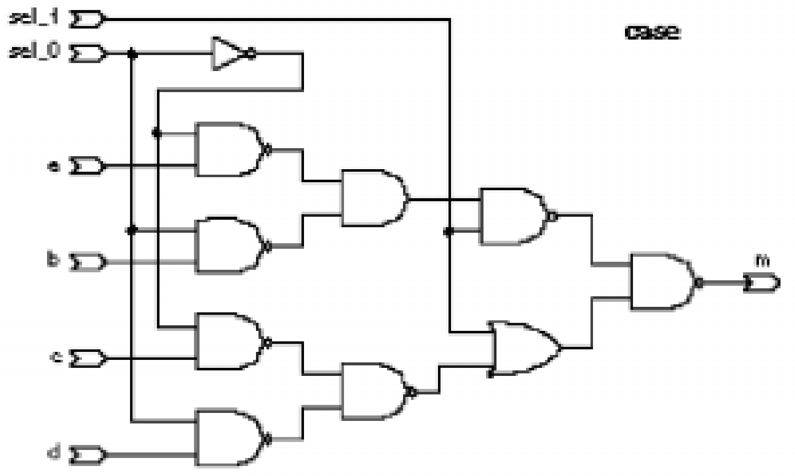


Conditional Logic (contd.)

- Case statement
  - VHDL requires that all the possible conditions be represented in the condition of a **case** statement.
  - Use the **others** clause at the end of a case statement to cover any unspecified conditions.
  - Since **std_ulogic** and **std_logic types** have nine possible values (instead of two possible values for bit types), you should always include an **others** clause when using these types.

Conditional Logic (contd.)

```
entity control_stmts is
    port (sel: in bit_vector (0 to 1); a,b,c,d: in bit;
          m: out bit);
end control_stmts;
architecture example of control_stmts is
begin
    process (sel,a,b,c,d)
    begin
        case sel is
            when b"00"  => m <= c;
            when b"01"  => m <= d;
            when b"10"  => m <= a;
            when others => m <= b;
        end case;
    end process;
end example;
```



Replicated Logic

- Function
- Procedure
- Loop statement
- Generate statement
  - Concurrent loop statement

Replicated Logic (contd.)

- Functions
  - always terminated by **return** statement.
  - For both functions and procedures, the VHDL synthesizer will generate a block of logic for each instance (unique reference to) the function or procedure.

Replicated Logic (contd.)

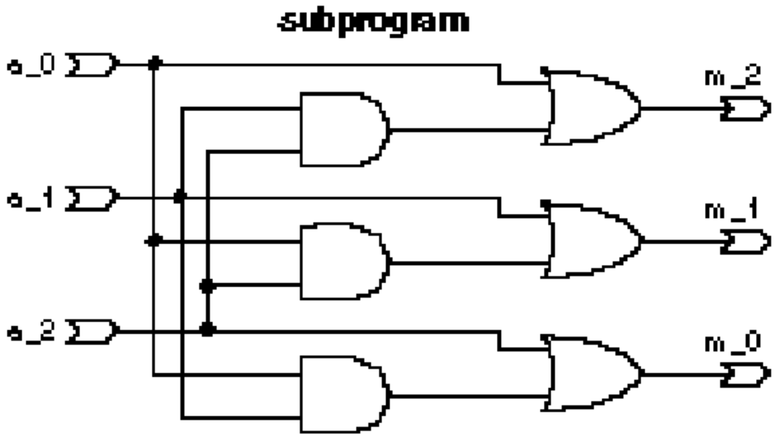
```
entity func is
    port (a: in bit_vector (0 to 2);
          m: out bit_vector (0 to 2));
end func;
architecture example of func is
    function simple (w, x, y: bit) return bit is
    begin
        return (w and x) or y;
    end;
begin
    process (a)
    begin
        m(0) <= simple(a(0), a(1), a(2));
        m(1) <= simple(a(2), a(0), a(1));
        m(2) <= simple(a(1), a(2), a(0));
    end process;
end example;
```

Replicated Logic (contd.)

- Procedures
  - do not have a return value.
  - A return statement may also be used in a procedure, where it never returns a value.

Replicated Logic (contd.)

```
entity proc is
    port (a: in bit_vector (0 to 2);
          m: out bit_vector (0 to 2));
end proc;
architecture example of subprograms is
    procedure simple (w, x, y: in bit; z: out bit) is
    begin
        z <= (w and x) or y;
    end;
begin
    process (a)
    begin
        simple(a(0), a(1), a(2), m(0));
        simple(a(2), a(0), a(1), m(1));
        simple(a(1), a(2), a(0), m(2));
    end process;
end example;
```



Replicated Logic (contd.)

- Loop statement
  - If possible, loop ranges should be expressed as constants. Otherwise, the logic inside the loop may be replicated for all the possible values of the loop ranges.
  - Loop statements may be terminated with an **exit** statement

Replicated Logic (contd.)

```
entity loop_stmt is
    port (a: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end loop_stmt;

architecture example of loop_stmt is
begin
    process (a)
        variable b:bit;
    begin
        b := 1;
        for i in 0 to 3 loop     -- no need to declare i
            b := a(3-i) and b;
            m(i) <= b;
        end loop;
    end process;
end example;
```

Replicated Logic (contd.)

– Specific iterations of the loop statement may be terminated with the **next** statement.
  - Terminates the current iteration of a loop, then continues with the first statement in the loop.

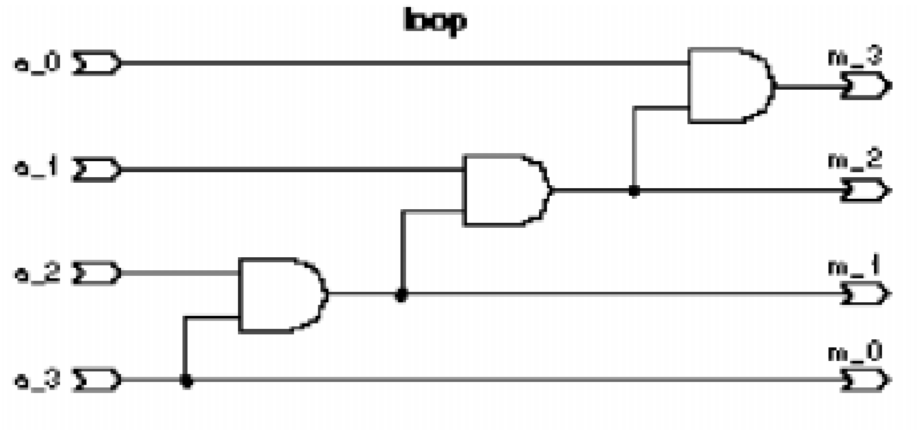
```
for I in 1 to 8 loop
  next when COPY_ENABLE(I) = '0';
  A(I) <= B(I);
end loop;
```

Replicated Logic (contd.)

- While statement
  - Supported by the synthesizer only when the termination can be determined at the time of synthesis.
  - Unconditioned loops are not synthesizable.

Replicated Logic (contd.)

```
entity while_stmt is
    port (a: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end while_stmt;
architecture example of while_stmt is
begin
    process (a)
        variable b: bit;
        variable i: integer;
    begin
        i := 0;
        while i < 4 loop
            b := a(3-i) and b;
            m(i) <= b;
        end loop;
    end process;
end example;
```



Replicated Logic (contd.)

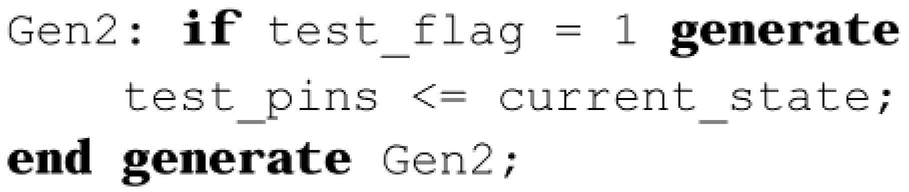
- Generate statement
  - Replicates one or more concurrent statements.
  - Has two forms: **for** and **if**.

```
Gen1: for i in 0 to 3 generate
SM: mod1 port map(A(i),B(i),Y(i));
end generate Gen1;
```

```
SM(0): mod1 port map(A(0),B(0),Y(0));
SM(1): mod1 port map(A(1),B(1),Y(1));
SM(2): mod1 port map(A(2),B(2),Y(2));
SM(3): mod1 port map(A(3),B(3),Y(3));
```

Replicated Logic (contd.)

- If generate statement
  - describe a conditional selection of concurrent statements.



```
Gen2: if test_flag = 1 generate
    test_pins <= current_state;
end generate Gen2;
```

  - the conditional expression (in this case "test_flag = 1") must be a metalogic value (one that does not depend on a signal or variable.)

Replicated Logic (contd.)

– Example:

```
for i in 0 to 3 generate
   if i /= 2 generate
      SM: modl port map A(i),B(i),Y(i);
   end generate;
end generate;
```

Sequential Logic

- There is no
  - register assignment operator
  - special attributes or dot extensions for specifying clocks and resets.
- There are often several ways to describe a particular behavior. There is no right-style, however.
- Two commonly-used methods
  - `if-then` and `wait` statements

Sequential Logic (contd.)

- Conditional specification
  - Relies on the inherent behavior of **if** statement.

```
process (clk)
    begin
    if clk='1' then
        y <= a;
    else
            -- default: holds previous value
    end if;
end process;
```

  - ABEL-HDL makes the signal **zero** without **else** clause.

Sequential Logic (contd.)

– If both condition has been written, the behavior would be **mux**.

– However, the following code results in error because **a** and **b** are not included in the **sensitivity list**.

```
process(clk)
    begin
    if clk='1' then
        y <= a;
    else
        y <= b;
    end if;
end process;
```

Sequential Logic (contd.)

- Inverting conditional logic

```
process(clk)
    begin
    if clk='1' then
            -- hold
    else
        y <= a;
    end if;
end process;
```

```
process(clk)
    begin
    if clk='0' then
        y <= a;
    end if;
end process;
```

Sequential Logic (contd.)

- Rising-edge flip-flop

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

Sequential Logic (contd.)

- IEEE 1164 std_logic (or std_ulogic)
  - Improved accuracy of simulation.

```
process(clk)
    begin
    if rising_edge(clk) then
        y <= a;
    end if;
end process;
```

Sequential Logic (contd.)

- Caution
  - Checking **absence of clock edge** has no hardware counterpart.

```
       if(CLK_B'event and CLK_B = '1') then  --
Illegal
         C <= B;
       end if;
     end process;
```

Sequential Logic (contd.)

– Do not assign a value in **false** branch

  • Also checks absence of clock edge

```
process(CLK)
begin
  if(CLK'event and CLK = '1') then
    SIG <= B;
  else
    SIG <= C;      -- Illegal
  end if;
end process;
```

Sequential Logic (contd.)

- Summary
  - Incomplete **if** statement implies a flip-flop or latch primitive.
  - Incomplete assignments within **case** statements can also result in latches using combinational feedback rather than latch primitives.
  - Complete **if** statement specifies (using an **else** clause) a combinational function.

Sequential Logic (contd.)

– The clock input (in this case **clk**) can have any other name.

– Implied flip-flops and latches can occur on either signals or variables.

Sequential Logic (contd.)

- Wait statement

```
process
    wait until expression;
    .
    .
    .
end process;
```

```
process
    wait until clk'event and clk='1'
    y <= a;
end process;
```

Sequential Logic (contd.)

- Suspends evaluation (over time) until an event occurs, and the expression evaluates to **true**.
- When used in a process, **no process sensitivity list** is required (or allowed).
- VHDL synthesizer limits that
  - **wait** statements must be located at either the beginning or end of a process.
  - There may not be more than one **wait** statement in a process.
- **Wait** statements are not recommended for use in synthesizable designs.

Sequential Logic (contd.)

- Other examples of latches
  - Current conditional assignment

```
architecture dataflow of latch is
begin
    y <= a and b when clk else y;
end dataflow;
```

Sequential Logic (contd.)

– Concurrent procedure call

```
architecture dataflow of flipflop is
    procedure my_ff(signal clk,a,b: Boolean;
        signal y : out Boolean)
    begin
        if clk='1' and clk'event then
            y <= a and b;
        end if;
    end;
begin
    ff_1: my_ff (clock,input1,input2,outputA);
    ff_2: my_ff (clock,input1,input2,outputB);
end dataflow;
```

Sequential Logic (contd.)

- Gated clock and clock enable
  - Gated clock
    - Unreliable and/or unsynthesizable

```
if clk='1' and clk1'event and ena then
    q <= d;
end if;
```

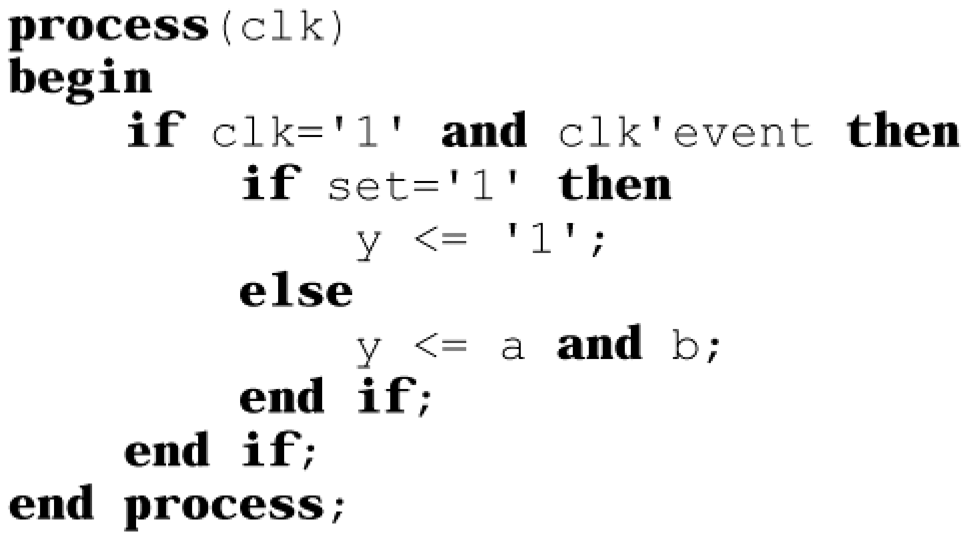
Sequential Logic (contd.)

- Clock enable feature
  - By specifying in VHDL synthesizer.
  - Otherwise **mux** will be generated.

```
if clk='1' and clk1'event then
    if ena then
        q <= d;
    end if;
end if;
```

Sequential Logic (contd.)

- Synchronous set/reset



```
process(clk)
begin
    if clk='1' and clk'event then
        if set='1' then
            y <= '1';
        else
            y <= a and b;
        end if;
    end if;
end process;
```

Sequential Logic (contd.)

– Mixture of set/reset for an arbitrary encoding

```
process (clk)
begin
    if clk='1' and clk'event then
        if init='1' then
            y <= 7;      -- y is type integer
        else
            y <= a + b;
        end if;
    end if;
end process;
```

Sequential Logic (contd.)

- Asynchronous set/reset

```
process (clk,reset)
begin
    if reset='1' then
        y <= false;         -- y is type Boolean

    elsif clk='1' and clk'event then
        y <= a and b;
    end if;
end process;
```