

# Introduction to VHDL

Instructor: Saraju P. Mohanty

## Objective of this lecture

- Basic language concepts
- Basic design methodology
- Writing simple VHDL code

**Note:** The slides are from either text book or reference book authors or publishers.

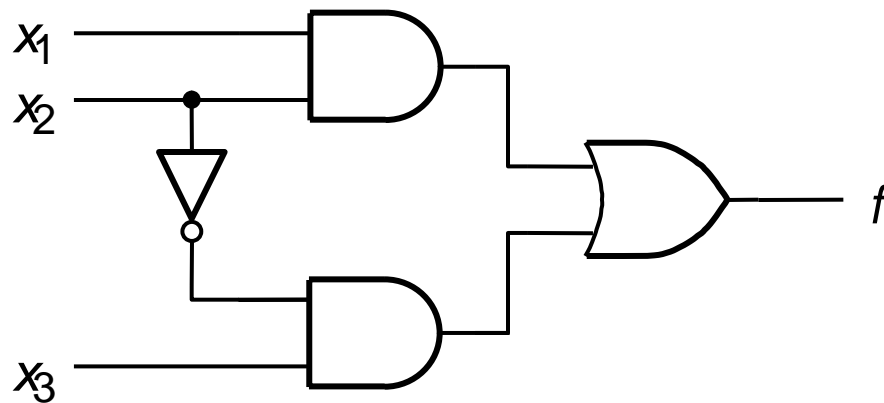
# VHDL Modeling Digital Systems

- Reasons for modeling
  - requirements specification
  - documentation
  - testing using simulation
  - formal verification
  - synthesis
- Goal
  - most reliable design process, with minimum cost and time
  - avoid design errors!

# Basic VHDL Concepts

- Interfaces
- Behavior
- Structure
- Test Benches
- Analysis, elaboration, simulation
- Synthesis

# Simple VHDL Code Example1 : The Circuit



Circuit

X1	X2	X3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Truth Table

A simple logic function.

## Simple VHDL Code Example1 : Entity

- **Step1:** Declare the input and output signals using the construct “entity”.

```
ENTITY example1 IS
```

```
    PORT ( x1, x2, x3   : IN   BIT ;  
           f           : OUT  BIT ) ;
```

```
END example1 ;
```

- “example1” is the name of entity.
- The I/O signals of the entity are called ports and it had reserve keyword “PORT”.
- PORT has modes: IN-input, OUT-output

## Simple VHDL Code Example1 : Architecture

- The signal represented by a PORT has an associated type, “BIT”.
- **Step2:** Circuit functionality is specified using a construct called “architecture”.

```
ARCHITECTURE LogicFunc OF example1 IS  
BEGIN
```

```
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
```

```
END LogicFunc ;
```

- “LogicFunc” is the architecture name.
- VHDL has built-in support for the Boolean operators.

# Simple VHDL Code Example1 : Complete VHDL

```
ENTITY example1 IS
```

```
    PORT ( x1, x2, x3  : IN  BIT ;
```

```
           f           : OUT BIT ) ;
```

```
END example1 ;
```

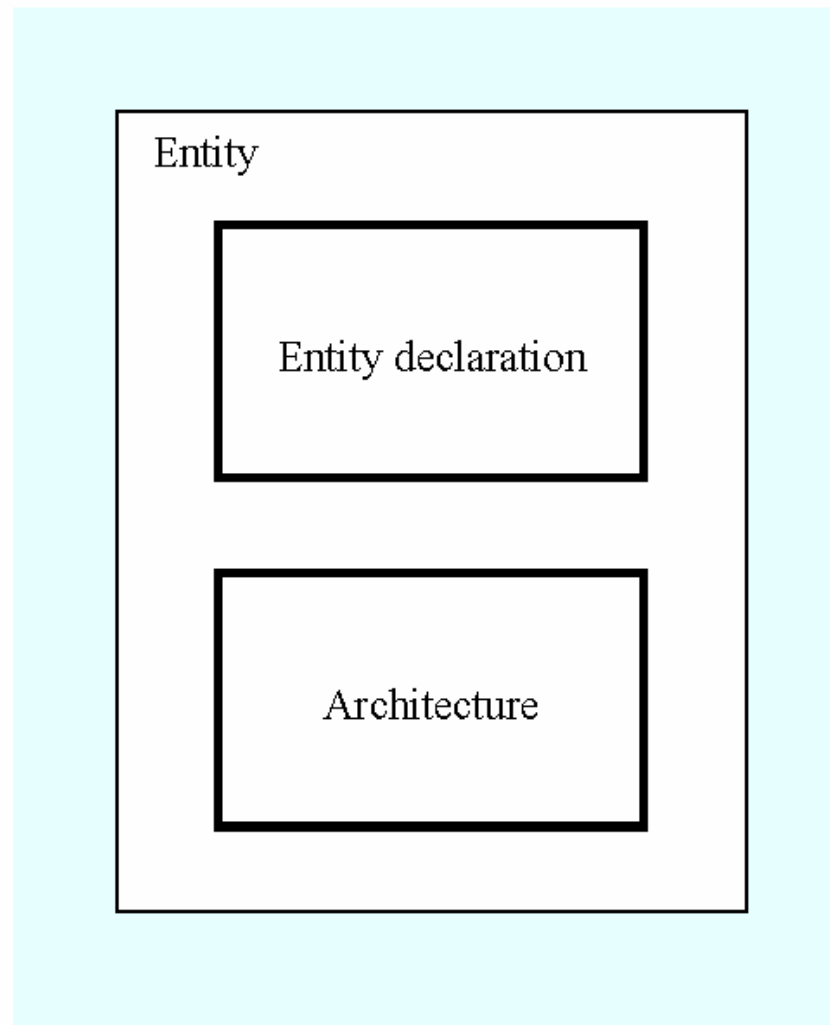
```
ARCHITECTURE LogicFunc OF example1 IS
```

```
BEGIN
```

```
    f <= (x1 AND x2) OR (NOT x2 AND x3) ;
```

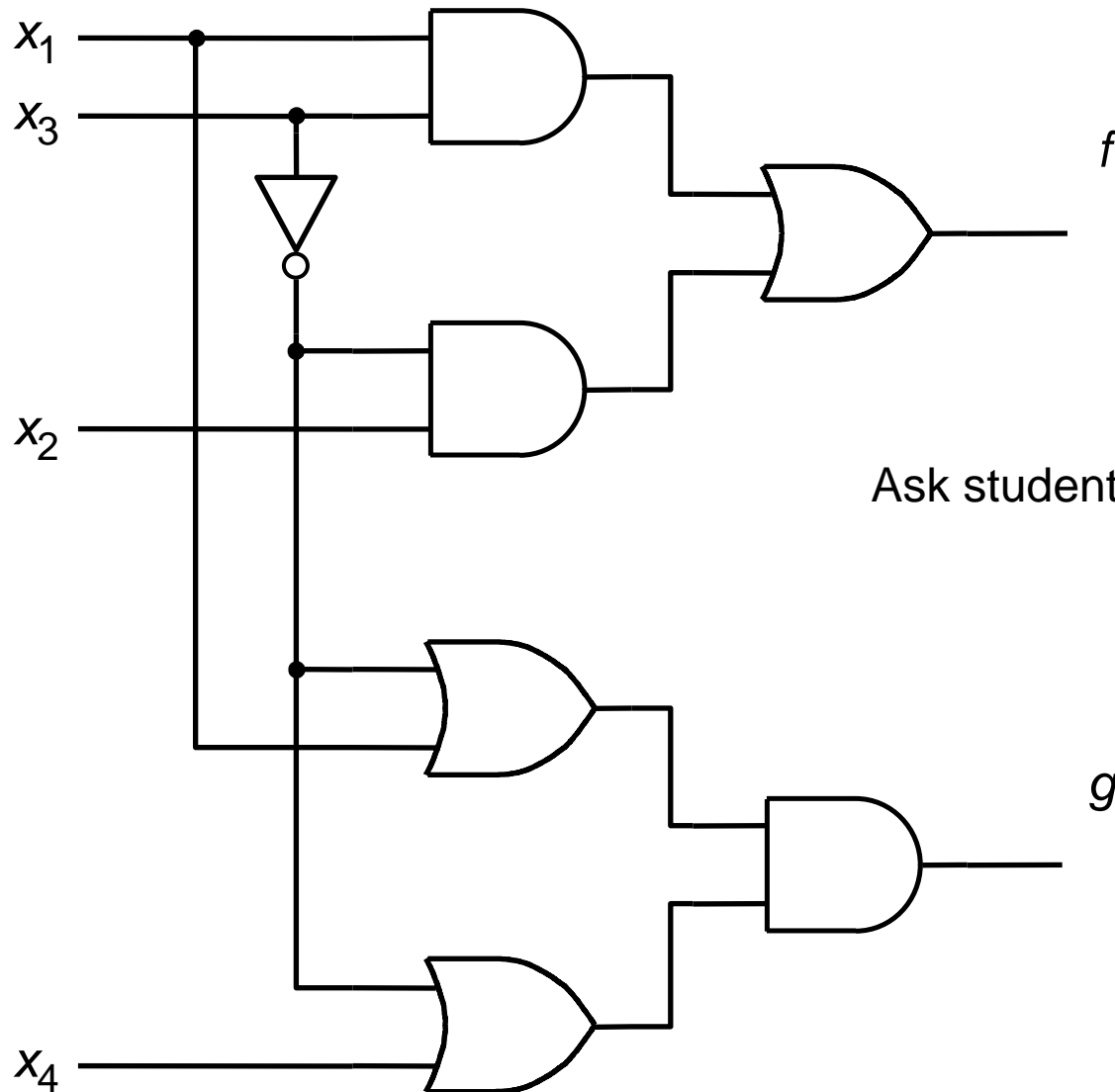
```
END LogicFunc ;
```

# Basic VHDL Concepts: Entity and Architecture





## VHDL Code Example2 :The Circuit



Ask students to write in class ??

## VHDL Code Example2 : Complete VHDL

```
ENTITY example2 IS
```

```
    PORT ( x1, x2, x3, x4  : IN    BIT ;  
           f, g             : OUT  BIT ) ;
```

```
END example2 ;
```

```
ARCHITECTURE LogicFunc OF example2 IS
```

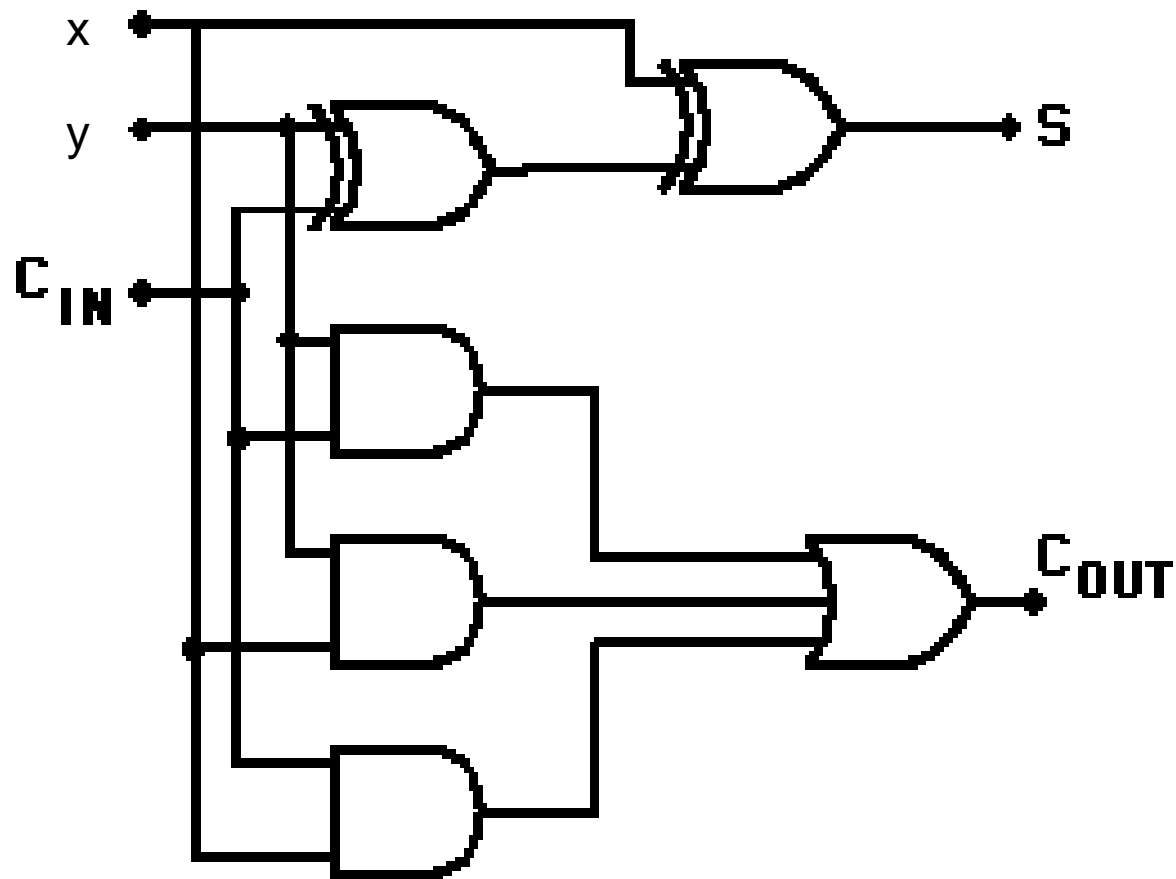
```
BEGIN
```

```
    f <= (x1 AND x3) OR (NOT x3 AND x2) ;
```

```
    g <= (NOT x3 OR x1) AND (NOT x3 OR x4) ;
```

```
END LogicFunc ;
```

## VHDL Code Example3 : Full Adder Circuit



## VHDL Code Example3 : Full Adder VHDL

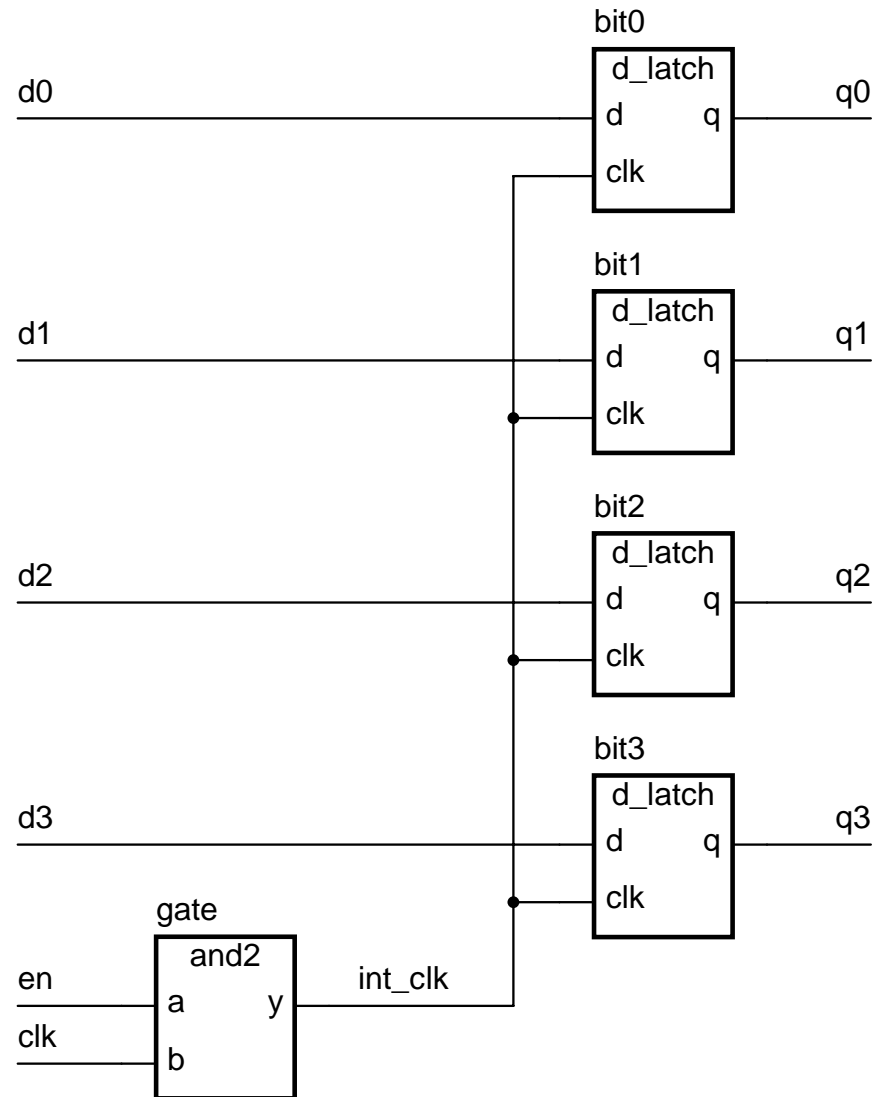
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y  : IN    STD_LOGIC ;
          s, Cout     : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin) ;
END LogicFunc ;
```

Figure A.4. Code for a full-adder.

# VHDL Code Example4 : The Circuit



# VHDL Code Example4 : Modeling Interfaces

- *Entity* declaration
  - describes the input/output *ports* of a module

*reserved words*    *entity name*    *port names*    *port mode (direction)*

```
entity reg4 is  
    port ( d0, d1, d2, d3, en, clk : in bit;  
          q0, q1, q2, q3 : out bit );  
end reg4;
```

*punctuation*

*port type*

A diagram showing a VHDL entity declaration code snippet enclosed in a rectangular box. Four dashed arrows point from labels above the box to specific parts of the code: 'reserved words' points to 'entity', 'entity name' points to 'reg4', 'port names' points to 'd0, d1, d2, d3, en, clk', and 'port mode (direction)' points to 'in'. A dashed arrow points from 'punctuation' to the semicolon at the end of the port list. Another dashed arrow points from 'port type' to 'bit' in the output port declaration.

# VHDL Code Example4 : Modeling Behavior

- *Architecture body*
  - describes an implementation of an entity
  - may be several per entity
- *Behavioral architecture*
  - describes the algorithm performed by the module
  - contains
    - *process statements*, each containing
    - *sequential statements*, including
    - *signal assignment statements* and
    - *wait statements*

# VHDL Code Example4 : Behavior Example

```
architecture behav of reg4 is  
begin  
    storage : process  
        variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;  
    begin  
        if en = '1' and clk = '1' then  
            stored_d0 := d0;  
            stored_d1 := d1;  
            stored_d2 := d2;  
            stored_d3 := d3;  
        end if;  
        q0 <= stored_d0 after 5 ns;  
        q1 <= stored_d1 after 5 ns;  
        q2 <= stored_d2 after 5 ns;  
        q3 <= stored_d3 after 5 ns;  
        wait on d0, d1, d2, d3, en, clk;  
    end process storage;  
end behav;
```



# VHDL Code Example4 : Modeling Structure

- *Structural* architecture
  - implements the module as a composition of subsystems
  - contains
    - *signal declarations*, for internal interconnections
      - the entity ports are also treated as signals
    - *component instances*
      - instances of previously declared entity/architecture pairs
    - *port maps* in component instances
      - connect signals to component ports
    - *wait statements*

## VHDL Code Example4 : Modeling Structure

- Can not directly instantiate entity / architecture pair
- Instead
  - include *component declarations* in structural architecture body
    - templates for entity declarations
  - instantiate components
  - write a *configuration declaration*
    - binds entity/architecture pair to each instantiated component

# VHDL Code Example4 : Modeling Structure

- First declare D-latch and and-gate entities and architectures

```
entity d_latch is  
    port ( d, clk : in bit; q : out bit );  
end d_latch;  
  
architecture basic of d_latch is  
begin  
    latch_behavior : process  
    begin  
        if clk = '1' then  
            q <= d after 2 ns;  
        end if;  
        wait on clk, d;  
    end process latch_behavior;  
end basic;
```

```
entity and2 is  
    port ( a, b : in bit; y : out bit );  
end and2;  
  
architecture basic of and2 is  
begin  
    and2_behavior : process  
    begin  
        y <= a and b after 2 ns;  
        wait on a, b;  
    end process and2_behavior;  
end basic;
```

## VHDL Code Example4 : Modeling Structure

- Declare corresponding components in register architecture body

```
architecture struct of reg4 is  
    component d_latch  
        port ( d, clk : in bit; q : out bit );  
    end component;  
    component and2  
        port ( a, b : in bit; y : out bit );  
    end component;  
    signal int_clk : bit;  
  
    ...
```

## VHDL Code Example4 : Modeling Structure

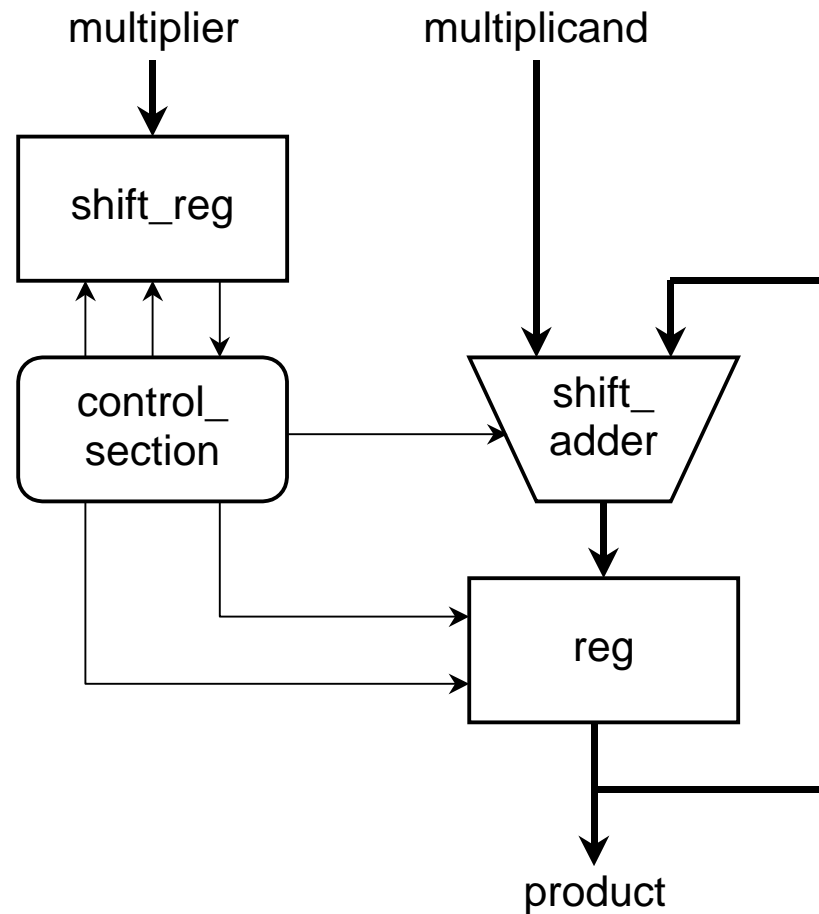
- Now use them to implement the register

```
...  
begin  
    bit0 : d_latch  
        port map ( d0, int_clk, q0 );  
    bit1 : d_latch  
        port map ( d1, int_clk, q1 );  
    bit2 : d_latch  
        port map ( d2, int_clk, q2 );  
    bit3 : d_latch  
        port map ( d3, int_clk, q3 );  
    gate : and2  
        port map ( en, clk, int_clk );  
end struct;
```

# Mixed Behavior and Structure

- An architecture can contain both behavioral and structural parts
  - process statements and component instances
    - collectively called *concurrent statements*
  - processes can read and assign to signals
- Example: register-transfer-level model
  - data path described structurally
  - control section described behaviorally

# VHDL Code Example5 : Mixed Example Circuit



# VHDL Code Example5 : Mixed Example VHDL

**entity** multiplier **is**

**port** ( clk, reset : **in** bit;  
          multiplicand, multiplier : **in** integer;  
          product : **out** integer );

**end entity** multiplier;

**architecture** mixed **of** mulitplier **is**

**signal** partial\_product, full\_product : integer;  
    **signal** arith\_control, result\_en, mult\_bit, mult\_load : bit;

**begin**

    arith\_unit : **entity** work.shift\_adder(behavior)  
        **port map** ( addend => multiplicand, augend => full\_product,  
                  sum => partial\_product,  
                  add\_control => arith\_control );

    result : **entity** work.reg(behavior)  
        **port map** ( d => partial\_product, q => full\_product,  
                  en => result\_en, reset => reset );

...



# VHDL Code Example5 : Mixed Example VHDL

```
...
multiplier_sr : entity work.shift_reg(behavior)
    port map ( d => multiplier, q => mult_bit,
              load => mult_load, clk => clk );
product <= full_product;

control_section : process is
    -- variable declarations for control_section
    -- ...
begin
    -- sequential statements to assign values to control signals
    -- ...
    wait on clk, reset;
end process control_section;
end architecture mixed;
```

# Testing the Designed Circuit Using Test Benches

- Testing a design by simulation
- Use a *test bench* model
  - an architecture body that includes an instance of the design under test
  - applies sequences of test values to inputs
  - monitors values on output signals
    - either using simulator
    - or with a process that verifies correct operation

# Test Bench Example: For the 4-bit register

```
entity test_bench is  
end entity test_bench;  
  
architecture test_reg4 of test_bench is  
    signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;  
begin  
    dut : entity work.reg4(behav)  
        port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );  
    stimulus : process is  
        begin  
            d0 <= '1'; d1 <= '1'; d2 <= '1'; d3 <= '1'; wait for 20 ns;  
            en <= '0'; clk <= '0'; wait for 20 ns;  
            en <= '1'; wait for 20 ns;  
            clk <= '1'; wait for 20 ns;  
            d0 <= '0'; d1 <= '0'; d2 <= '0'; d3 <= '0'; wait for 20 ns;  
            en <= '0'; wait for 20 ns;  
            ...  
            wait;  
        end process stimulus;  
end architecture test_reg4;
```

# Design Processing

- Analysis
- Elaboration
- Simulation
- Synthesis

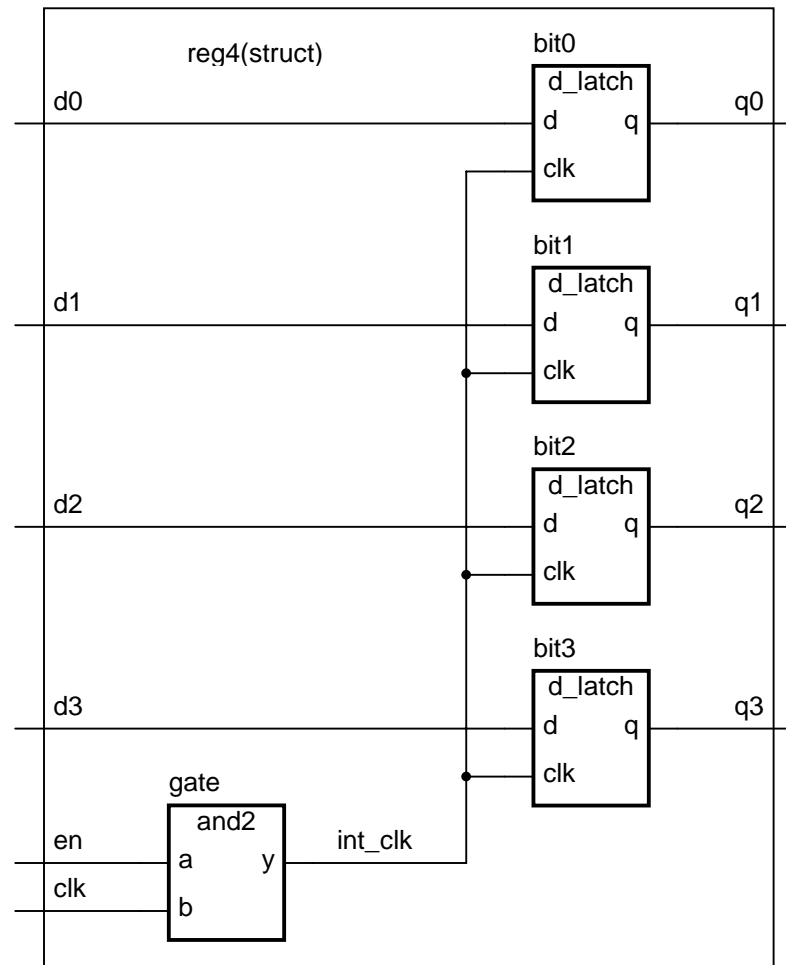
# Design Processing : Analysis

- Check for syntax and semantic errors
  - syntax: grammar of the language
  - semantics: the meaning of the model
- Analyze each *design unit* separately
  - entity declaration
  - architecture body
  - ...
  - best if each design unit is in a separate file
- Analyzed design units are placed in a *library*
  - in an implementation dependent internal form
  - current library is called work

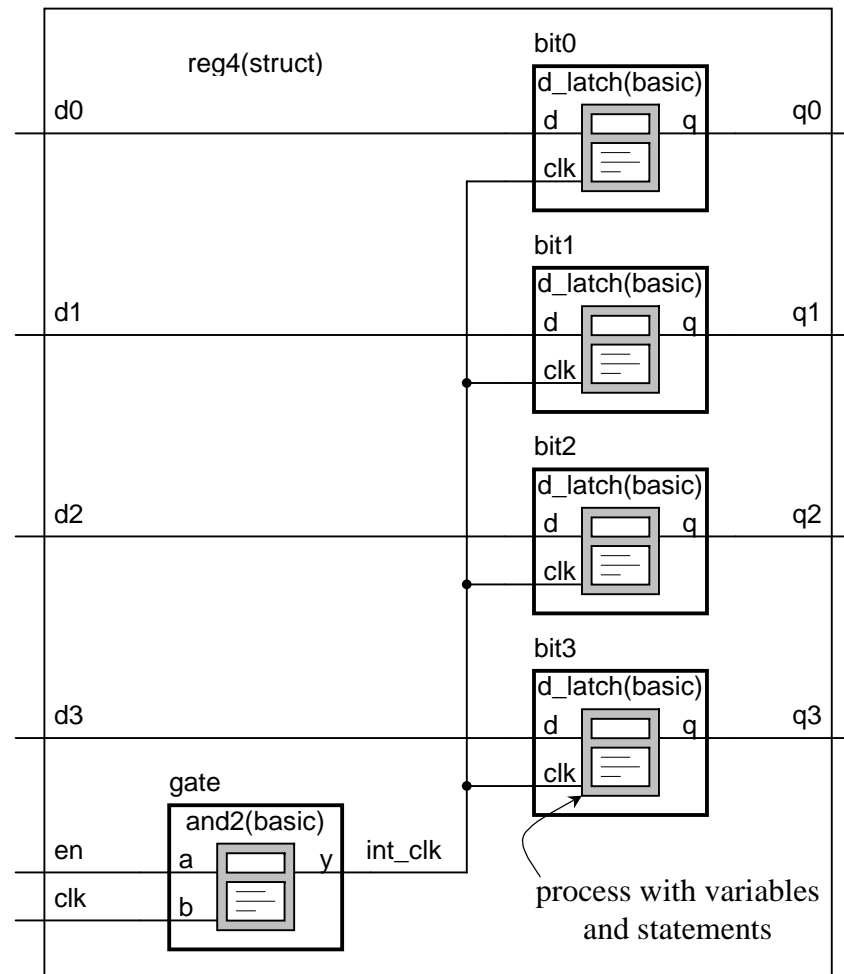
# Design Processing : Elaboration

- “Flattening” the design hierarchy
  - create ports
  - create signals and processes within architecture body
  - for each component instance, copy instantiated entity and architecture body
  - repeat recursively
    - bottom out at purely behavioral architecture bodies
- Final result of elaboration
  - flat collection of signal nets and processes

# Design Processing : Elaboration Example



# Design Processing : Elaboration Example





# Design Processing : Simulation

- Execution of the processes in the elaborated model
- Discrete event simulation
  - time advances in discrete steps
  - when signal values change—*events*
- A processes is sensitive to events on input signals
  - specified in wait statements
  - resumes and schedules new values on output signals
    - schedules *transactions*
    - event on a signal if new value different from old value

# Design Processing : Simulation Algorithm

- Initialization phase
  - each signal is given its initial value
  - simulation time set to 0
  - for each process
    - activate
    - execute until a wait statement, then suspend
      - execution usually involves scheduling transactions on signals for later times

# Design Processing : Simulation Algorithm

- Simulation cycle
  - advance simulation time to time of next transaction
  - for each transaction at this time
    - update signal value
      - event if new value is different from old value
  - for each process sensitive to any of these events, or whose “wait for ...” time-out has expired
    - resume
    - execute until a wait statement, then suspend
- Simulation finishes when there are no further scheduled transactions

# Design Processing : Synthesis

- Translates register-transfer-level (RTL) design into gate-level netlist
- Restrictions on coding style for RTL model
- Tool dependent: A subset of RTL is synthesizable depending on the tool