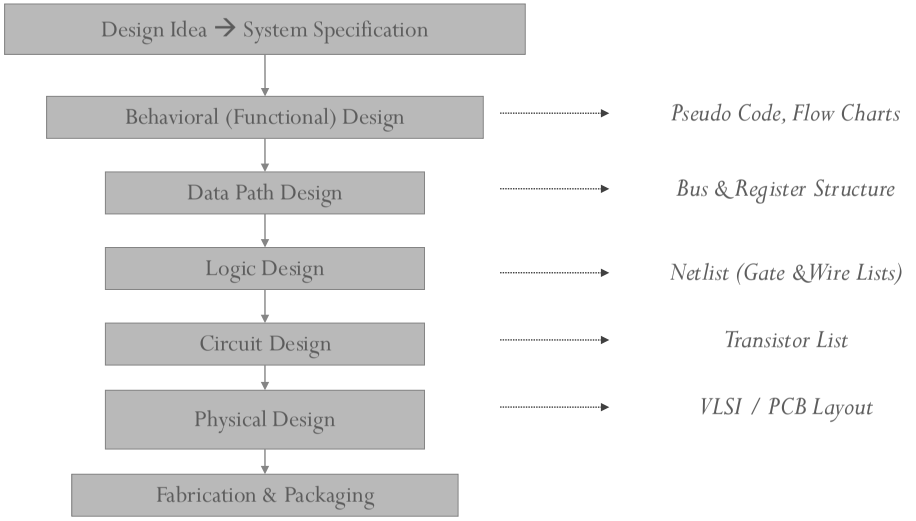
HARDWARE DESCRIPTION LANGUAGES

- HDL are used to describe the hardware for the purpose of modeling, simulation, testing, design, and documentation.
  - *Modeling*:behavior,flow of data, structure
  - *Simulation:* verification and test
  - *Design*: synthesis

Digital System Design Cycle



Purpose of VHDL

- Problem
  - Need a method to quickly design, implement, test, and document increasingly complex digital systems
  - Schematics and Boolean equations inadequate for million-gate IC
- Solution
  - A hardware description language (HDL) to express the design
  - Associated computer-aided design (CAD) or electronic design automation (EDA) tools for synthesis and simulation
  - Programmable logic devices for rapid implementation of hardware
  - Custom VLSI application specific integrated circuit (ASIC) devices for low-cost mass production

Design Automation & CAD Tools

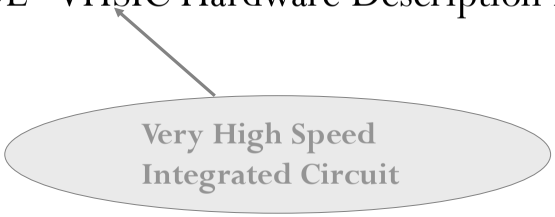
- Design Entry (Description) Tools
  - Schematic Capture
  - Hardware Description Language (HDL)
- Simulation (Design Verification) Tools
  - Simulators (Logic level, Transistor Level, High Language Level "HLL")
- Synthesis Tools
- Test Vector Generation Tools

VHDL: Why to use?

- **Reasons to use VHDL**
  - Power and flexibility
  - Device-independent design
  - Portability among tools and devices
  - Device and tool benchmarking capability
  - VLSI ASIC migration
  - Quick time-to-market and low cost (with programmable logic)
- **Problems with VHDL**
  - Loss of control with gate-level implementation (so what?)
  - Inefficient logic implementations via synthesis (engineer-dependent)
  - Variations in synthesis quality among tools(always improving)

History of VHDL

- Two widely-used HDLs today
- VHDL
- Verilog HDL (from Cadence, now IEEE standard)
- VHDL - VHSIC Hardware Description Language

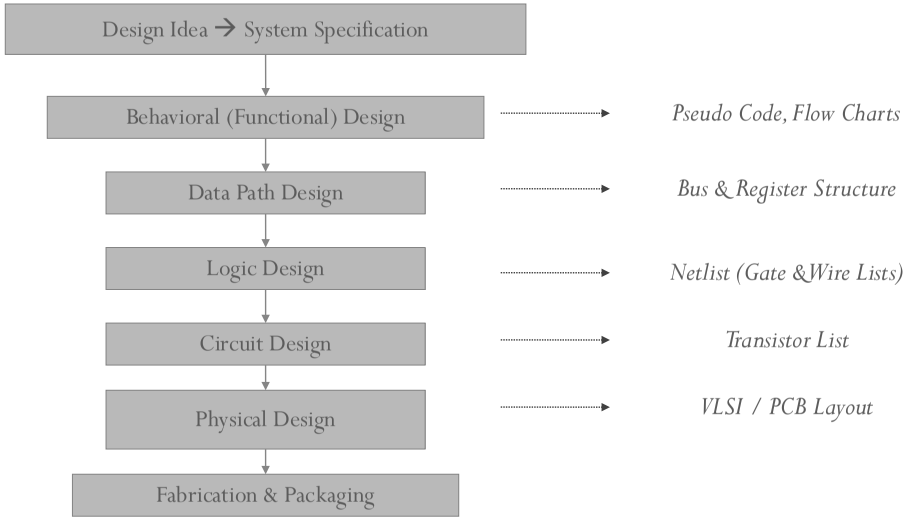


Design Flow in VHDL

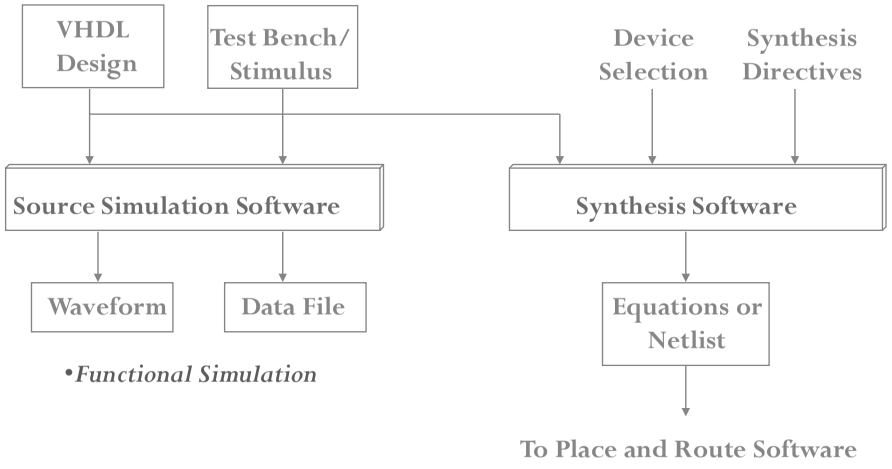
- Define the design requirements
- Describe the design in VHDL
  - Top-down, hierarchical design approach
  - Code optimized for synthesis or simulation
- Simulate the VHDL source code
  - Early problem detection before synthesis
- Synthesize, optimize, and (place and route) the design for a device
  - Synthesize to equations and/or netlist
  - Optimize equations and logic blocks subject to constraints
  - Fit into the components blocks of a given device
- Simulate the post-layout design model
  - Check final functionality and worst-case timing
- Program the device (if PLD) or send data to ASIC vendor

- VHDL history
  - Created by DOD to document military designs for portability
  - IEEE standard 1076 (VHDL) in 1987
  - Revised IEEE standard 1076 (VHDL) in 1993
  - IEEE standard 1164 (object types standard) in 1993
  - IEEE standard 1076.3 (synthesis standard) in 1996

Digital System Design Cycle



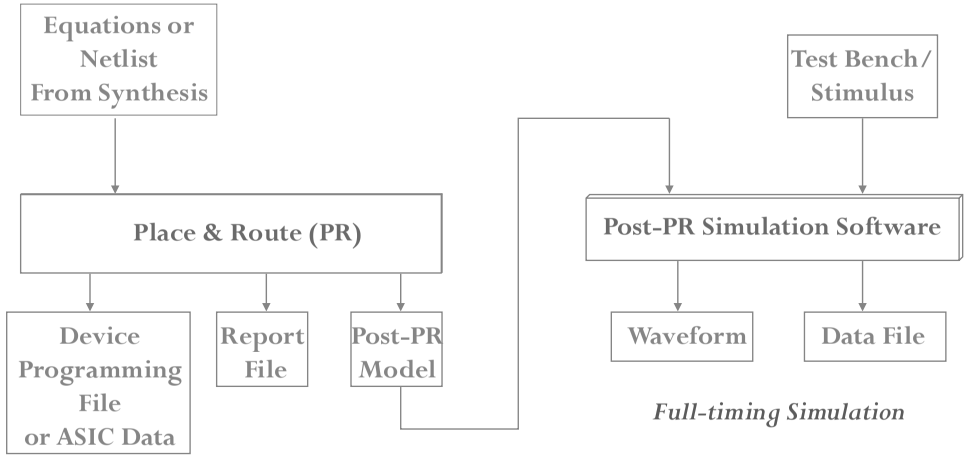
Design Tool Flow (1)



STYLES in VHDL

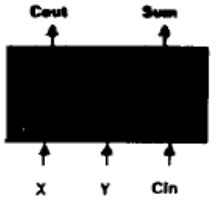
- **Levels of Abstraction (Architectural Styles):**
  - Behavioral
  - High level, algorithmic, sequential execution
  - Hard to synthesize well
  - Easy to write and understand (like high-level language code)
- **Dataflow**
  - Medium level, register-to-register transfers, concurrent execution
  - Easy to synthesize well
  - Harder to write and understand (like assembly code)
- **Structural**
  - Low level, netlist, component instantiations and wiring
  - Trivial to synthesize
  - Hardest to write and understand (very detailed and low level)

•Design Tool Flow (2)



An Interface Description for the Full Adder

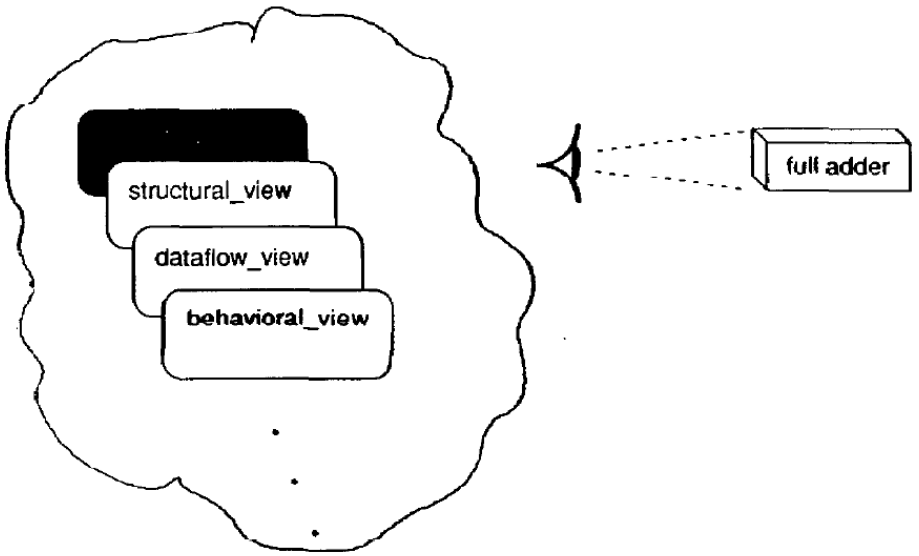
```
entity full_adder is
    port(
        X, Y, Cin:   in bit;   -- input ports
        Sum, Cout:   out bit); -- output ports
end full_adder;
```



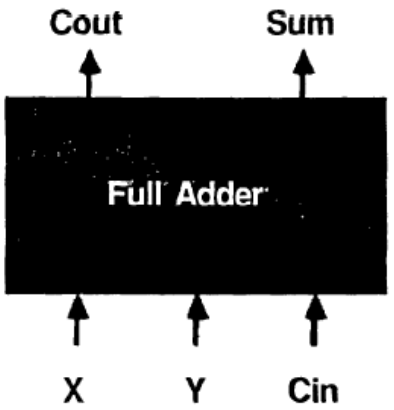
Introduction to VHDL

- A First Example
- Basic Building Blocks of VHDL Descriptions
- Structural Description in VHDL
- Data Flow Description in VHDL
- Behavioral Description in VHDL

Views of the Full Adder



A Full Adder

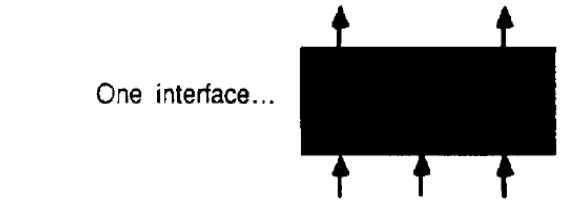


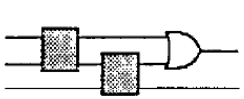
A Structural Architecture

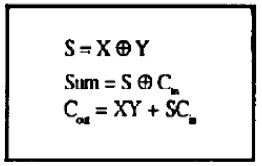
```
architecture structure_view of full_adder is

    component half_adder
        port (  I1, I2:   in bit;   - - inputs
                Carry:    out bit;  - - outputs
                Sum:      out bit);
    end component;
    component or_gate
        port (  I1, I2:   in bit;
                O:        out bit);
    end component;
    signal a, b, c: bit;

begin

    U1: half_adder   port map ( X, Y, a, b);
    U2: half_adder   port map ( b, Cin, c, Sum);
    U3: or_gate      port map (a, c, Cout);

end structure_view;
```

The Design Entity Concept

One interface...







...and any one of a collection of alternative architectures.

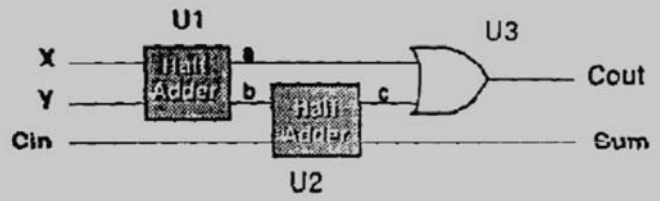
Boolean Equations for the Full Adder

$$S = X \oplus Y$$

$$Sum = S \oplus C_{in}$$

$$C_{out} = XY + SC_{in}$$

A Structure for the Full Adder



a, b, and c are intermediate nodes

A Behavioral Architecture

```
architecture behavioral_view of full_adder is

begin

    process
        variable N: integer;
        constant sum_vector: bit_vector (0 to 3) := "0101";
        constant carry_vector: bit_vector (0 to 3) := "0011";
    begin
        N := 0;
        if X = '1' then N := N+1; end if;
        if Y = '1' then N := N+1; end if;
        if Cin = '1' then N := N+1; end if;
        Sum <= sum_vector (N) after 20 ns;
        Cout <= carry_vector (N) after 30 ns;
        wait on X, Y, Cin;
    end process;

end behavioral_view;
```

A Dataflow Architecture

```
architecture dataflow_view of full_adder is

    signal S: bit;

begin

    S <=    X xor Y after 10 ns;
    Sum <= S xor Cin after 10 ns;
    Cout <= (X and Y) or (S and Cin) after 20 ns;

end dataflow_view;
```

Combining Styles of Description

```
architecture mixed_view of full_adder is

    component xor_gate
        port (   I1, I2:   in bit;
                 O:        out bit);
    end component;

    signal S: bit;

begin

    V1: Cout <= (X and Y) or (S and Cin) after 20 ns;
    V2: xor_gate port map ( X, Y, S);
    V3: xor_gate port map ( S, Cin, Sum);

end mixed_view;
```

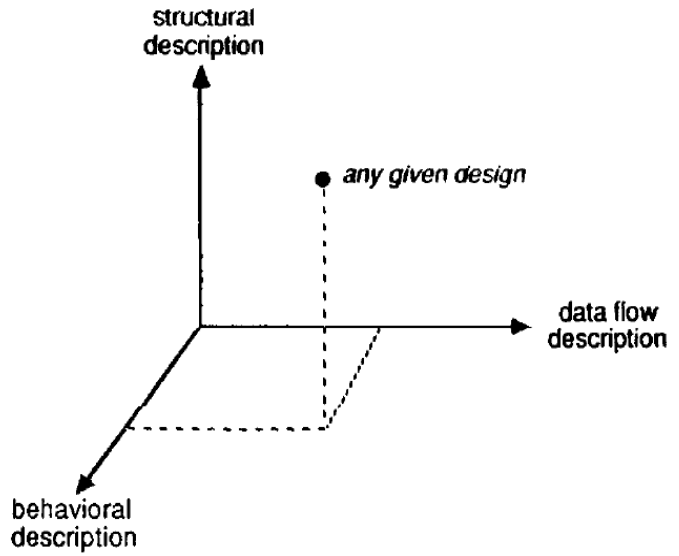
Function Table for the Full Adder

Inputs			Outputs	
Cin	X	Y	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A Configuration Declaration for the Full Adder

```
configuration alpha of full_adder is
    for structure_view
        for U1, U2: half_adder use
            entity work.half_add (macro4950),
        end for;
        for U3: or_gate use
            entity work.or2 (cell2396);
        end for;
    end for;
end alpha;
```

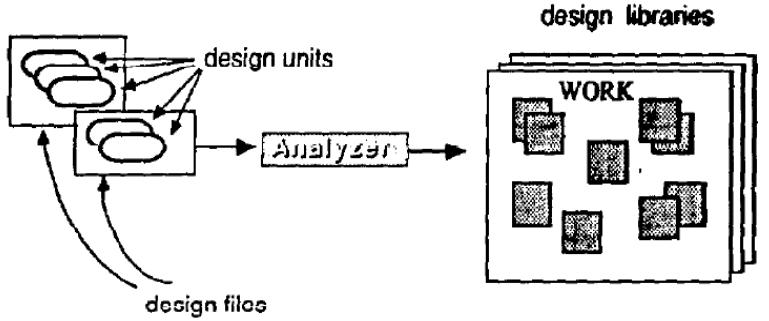
Design Space of VHDL



Characteristics of the Design Hierarchy

- A complete design consists of a hierarchy of interconnected design entities.
- Each design entity consists of an *entity declaration* and any one of the associated *architecture declarations*.
- All communication between design entities takes place through their interfaces.
- A design entity makes no assumptions about the context in which it is used.
- The replacement of one architecture by another (with the same functionality) will have no external effect.

The Design Library



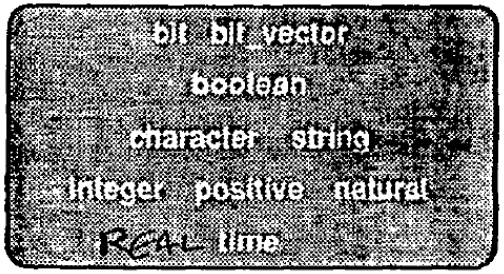
Objects

- *Objects* are containers of values. There are four classes of objects in VHDL: constants, variables, files and signals.
- *Constants* have a single value that may not be changed.
- *Variables* have a single value which can be changed by assignment.
- *Files* contain sequences of values that can be read or written.
- *Signals* have a past history of values, a present value, and set of projected future values; only future values can be changed by assignment.
- All objects are of some *type*.

Introduction to VHDL

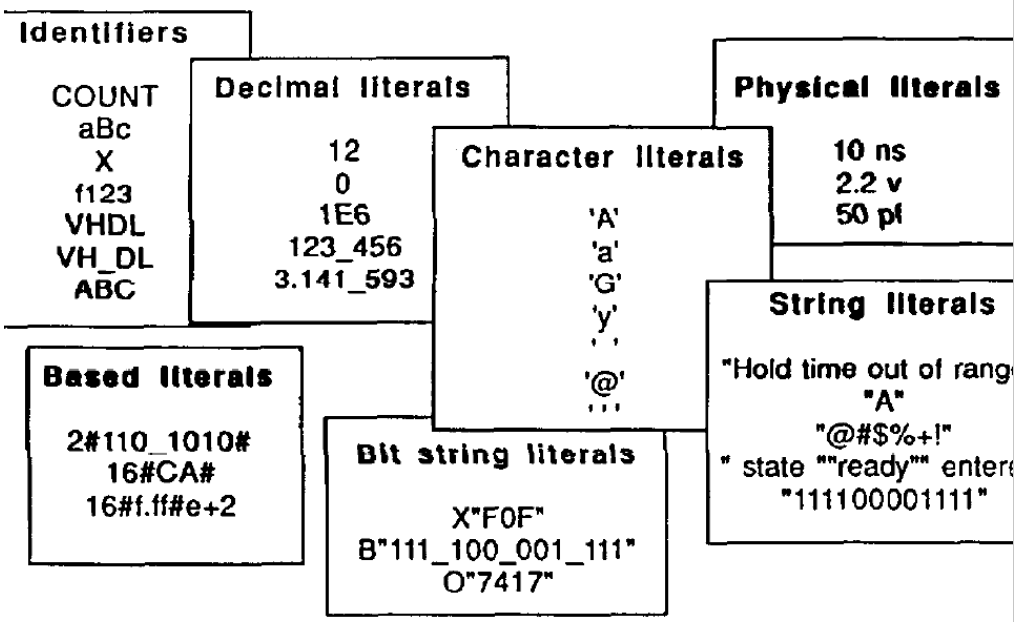
Types

- A type is a set of values.
- The type of each object is *static*; it cannot be changed during simulation.
- The type of an object determines how it can be used; for example, only certain operators are defined for a type.
- Several commonly used types and subtypes are predefined:

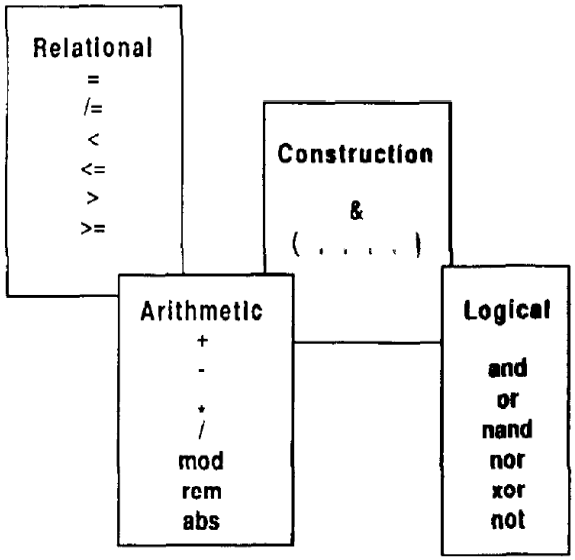


- The user may construct additional types using the predefined types as building blocks.

Lexical Elements



Operations



Examples of Types and Objects

```
type tri_value is ('0', '1', 'Z');
type system_state is (idle, test, target, fire, evaluate);
type small_integer is range 0 to 9;
type small_real is range 0.0 to 1.0;
type int_sequence is file of integer;

signal a, b, c: bit := '0';
signal stable: tri_value;

constant mid_range: small_integer := 5;

variable counter: small_integer;
variable random_number: small_real;

file counts: int_sequence is "project/data/rom243";
```

Expressions

Arithmetic (-b + sqrt (b**2) - 4.0*a*c) / (2.0*a)

Construction operation & register & base & displacement
(day => WED, date => (July, 4, 1776))

Relational delay <= 30 ns
name > "Jones"
event.date = (November, 12, 1984)

Logical CLOCK and not RESET
(A_reg xor memory_data) and B_reg
(A = '1' and B = '0') or (C = '0' and B = '1')

Examples of Types and Objects

```
type digit_display is array (1 to 20) of small_integer;
signal target_count: digit_display;

type state_descriptor is record
    state: system_state;
    sensors: bit_vector (1 to 16);
    operational: boolean;
end record;
type state_history is array (integer range <>) of state_descriptor;
variable R234_history: state_history (0 to 2047);
                .
                .
                .
R234_history (J) := (test, X"F0FC", false);
                .
                .
                .
```

Sequential Statements

```
Nfact := 1;
fact: for j in 2 to N loop
    Nfact := Nfact * j;
end loop fact;
```

```
case DAY
    when SAT | SUN => staff := 3;
    when MON | FRI => staff := 5;
    when TUE to THU => staff := 10;
end case;
```

```
while register(31) = '0' loop
    register := left_shift (register);
end loop;
```

```
loop
    wait for 20 ms;
    locate (target, distance);
    exit when distance < 0.25 km;
    range(target) := distance;
end loop;
```

```
if abs (A - B) < delta then
    delta := delta/2.0;
elsif A-B > 0.0 then
    guess := guess + delta;
else
    guess := guess - delta;
end if;
```

Statements

- *Statements specify the organization and operation of a design.*
- ***Sequential statements*** *specify algorithms (step by step instructions).*
- ***Concurrent statements*** specify

  Component interconnections
  Hierarchical structure
  Regular structure
  Data flow or register transfer operations

- Sequential statements are encapsulated in processes and subprograms for use in concurrent contexts.

Concurrent Statements

- The ***structure*** of a design is specified using

  Component instantiation statements
  Generate statements
  Block statements

- The ***behavior*** of a design is specified using

  Concurrent signal assignment statements
  Process statements
  Concurrent procedure calls
  Concurrent assertions

Sequential Statements

- Similiar to those in any high-level programming language:

  If and case statements
  Loop statements
  Procedure call and return statements
  Variable assignment statement
  Null statement

- Unique to VHDL:

  Signal assignment statement
  Wait statement
  Assertion statement

An Example of a Package

```
package tristate is

    type MVL is ('0', '1', 'Z', 'E');
    function "and" (X, Y: MVL) return MVL;
    function "or" (X, Y: MVL) return MVL;
    function "not" (X: MVL) return MVL;

    type MVL_vector is array (integer range <>) of MVL;
    function tri_resolve (sources: MVL_vector) return MVL;
    type tri_vector is array (integer range <>) of tri_resolve MVL;
    subtype byte is tri_vector (7 downto 0);

    constant high_Z: byte;
    function "and" (X, Y: tri_vector) return tri_vector;
    function "or" (X, Y: tri_vector) return tri_vector;
    function "not" (X: tri_vector) return tri_vector;

end tristate;
```

Concurrent Statements

```
S1 <= D1 after 10 ns, '0' after 30 ns;

S2 <=  X after 10 ns when C1 = '0' else
       Y after 7.5 ns when C2 = '0' else
       Z after 5 ns;

delay_line: for K in 1 to N generate
    S(K) <= S(K-1) after 100 fs;
end generate;
S(0) <= X after 100 fs;

with transmit_quality select
    channel <=  source when good,
                source - 2v when avera
                0.0v when poor;

parity: -- inputs X, Y, Z; output Result.
    block
        signal a: bit;
    begin
        a <= X xor Y after 5 ns;
        Result <= a xor Z after 5 ns;
    end parity;
```

An Example of a Package

```
package body tristate is
    constant high_Z: byte := "ZZZZZZZZ";
    function "and" (X, Y: MVL) return MVL is
            .
            .
            .
    end "and";

    function "or" (X, Y: MVL) return MVL is
            .
            .
            .
    end "or";
            .
            .
            .
end tristate;
```

Library Units

entity	Describes the interface with the outside world and any common characteristics of all implementations of a device.
architecture body	Describes the organization or operation of a device.
configuration	Selects design entities from a design library for component instances within an architecture.
package	Encapsulates a set of related declarations.
package body	Defines the bodies of subprograms and the values of deferred constants declared in a package.

Interface to the System

```
use tristate.all;
entity micro_system (
     DATA:   inout byte;
     ADDR:   inout bit_vector (11 downto 0);
     INT:    in bit;
     RW:     inout bit;
     IOREQ:  out bit )
end micro_system;
```

Introduction to VHDL

A First Example

Basic Building Blocks of VHDL Descriptions

Structural Description in VHDL

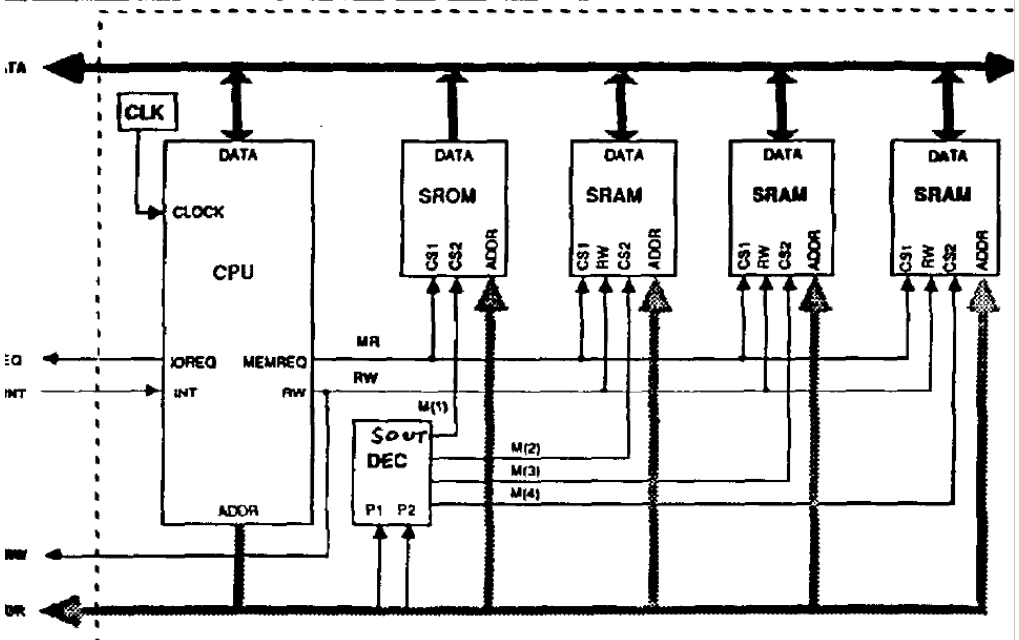
Data Flow Description in VHDL

Behavioral Description in VHDL

Components of the System

```
use tristate.all;
package parts is
     component clock
          port (C: out bit);
     end component;
     component ROM
          port ( DATA: out byte;  ADDR: in bit_vector (9 downto 0);
               CS1, CS2: in bit);
     end component;
     component RAM
          port ( DATA: inout byte; ADDR: in bit_vector (9 downto 0);
               CS1, CS2, RW: in bit);
     end component;
     component decoder
          port ( P: in bit_vector (1 to 2);  SOUT: out bit_vector (1 to 4));
     end component;
     component processor
          port ( DATA: inout byte; ADDR: out bit_vector (11 downto 0);
               CLOCK, INT: in bit;  MEMREQ, RW: out bit; IOREQ: out bit)
     end component;
end parts;
```

Simple Microprocessor System



Component Instantiation

- A ***component instance*** is created by a ***component instantiation statement***.
- Each instance is a new copy of the component, unrelated to any other instance.
- Each instance identifies the signal that is attached to each port of the component.
- Unused ports may be left unconnected.

Component Declarations

- A ***component declaration*** defines the interface to a subcomponent of a design.
- The declaration specifies:

  The name of the component
  The names and types of its ports
  The direction in which data flows through each port

Uses of Component Instantiation

- The designer can postpone decisions about the behavior of portions of his design during top-down decomposition.
- The designer can reuse a portion of a design that has already been created and stored in a design library.
- The design can choose either structural or functional decomposition.

Structural Architecture for the System

```
use parts.all;
architecture build1 of micro_system is
    signal CL, MR:   bit;
    signal M:        bit_vector (1 to 4);
begin
    CPU: processor   port map (DATA, ADDR, CL, INT, MR, RW, IOREQ);
    DEC: decoder     port map (P(1) => ADDR (11), P(2) => ADDR (10), M);
    CLK: clock       port map (CL);
    SROM: ROM        port map (DATA, ADDR (9 downto 0), MR, M(1));

    RAM_array:
    for i in 2 to 4 generate
        SRAM: RAM port map (DATA, ADDR (9 downto 0), MR, M(i), RW);
    end generate;
end build1;
```

Introduction to VHDL

A First Example

Basic Building Blocks of VHDL Descriptions

Structural Description in VHDL

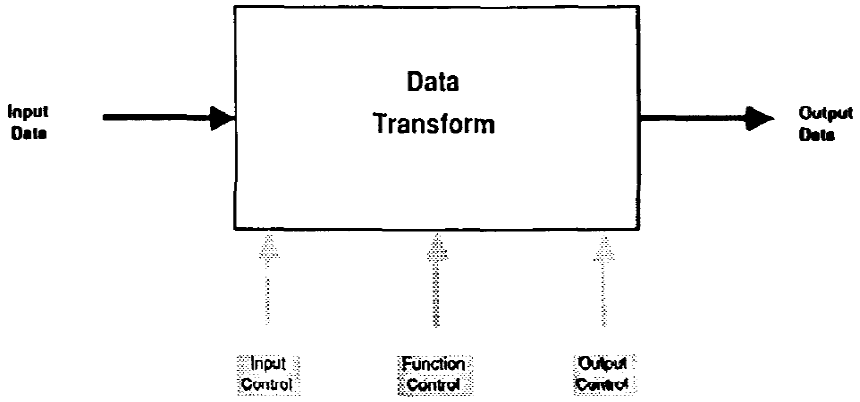
Data Flow Description in VHDL

Behavioral Description in VHDL

A Configuration of the System

```
library CMOS_A, CMOS_mem;
configuration simple of build1 is
    for micro_system                    (SWITCHED)
        for CPU: processor use
            entity CMOS_A.micro_processor (UP2400);
        end for;
        for DEC: decoder use
            entity CMOS_A.one_of_four (C0901);
        end for;
        for CLK: clock use
            entity CMOS_A.clock (C2310);
        end for;
        for SROM: ROM use
            entity CMOS_mem.ROM_1K8 (functional);
        end for;
        for RAM_array
            for SRAM(2 to 4): RAM use
                entity CMOS_mem.RAM_1K8 (functional);
            end for;
        end for;
    end for;
end simple;
```

General Model of Functional Devices

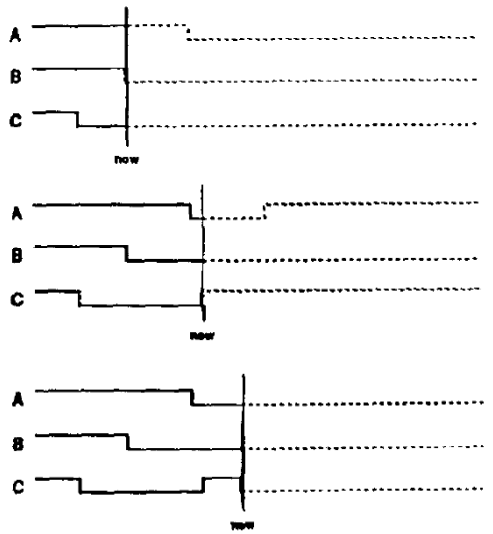


Binding Instances to Library Units

- A design system may support any number of ***design libraries.***
- Each component instance must be ***bound*** to an entity in some design library with a ***configuration specification.***
- An ***explicit*** configuration specification can appear in a configuration declaration, or right after the declaration of a component.
- If no explicit configuration specification for an instance appears, then the ***default*** configuration specification takes effect.

Tracing Signal Assignment Execution

A <= B or C after 5 ns;



Unconditional Signal Assignment

- The ***simple signal assignment*** models a data transform in which all inputs are data inputs:

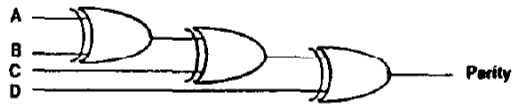
```
S <= intval (J) after 5 ns, abs K after 10 ns, (L+M)/2 after 15 ns;
```

- The ***target*** to the left of the arrow will receive the values defined by the ***waveform*** on the right.
- Each ***waveform element*** has a value part and a delay part.
- The value is any expression of the same type as the target.
- The delay is an expression of the physical type TIME. If the after clause is missing, a delay of 0 is assumed.
- All signals used in value parts are defined to be inputs.

Uses of Simple Signal Assignment

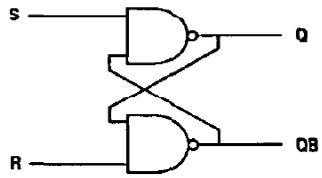
- Modeling combinational circuits:

```
parity <= A xor B xor C xor D;
```



- Modeling asynchronous sequential circuits:

```
Q<= S nand QB;
QB <= R nand Q;
```



Signal Assignment Execution

- A signal assignment statement executes in response to changes on its input signals.
- Each value in the waveform will be scheduled to appear on the target after the specified delay.
- If the assignment statement executes again, previously scheduled values may be overridden.
- A delay of zero represents an infinitesimally small delay – signal assignment never takes effect immediately.

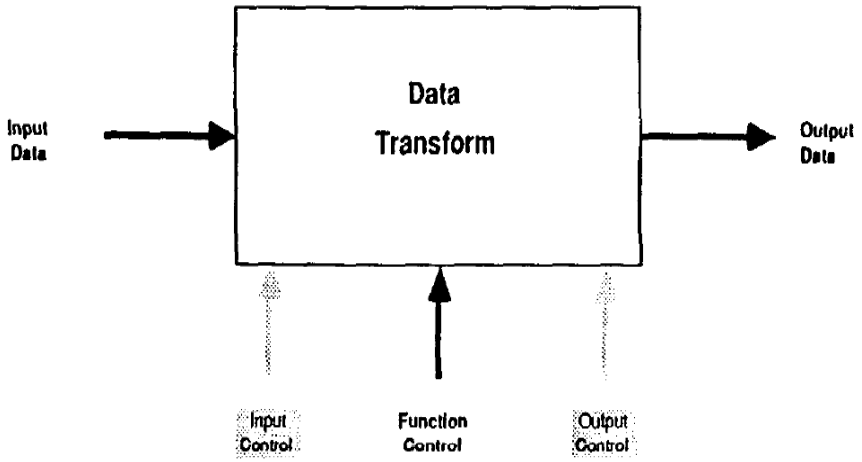
Uses of Conditional Signal Assignment

- To switch between two functions based on a single condition:

```
RegC <= RegA after 150 ns when A_enable else RegB after 150 ns;
```

- To switch among functions when multiple control lines must be considered in some order of precedence:

```
bus_cycle <=
        dma when dma_req else
        interrupt when interrupt_req and not interrupt_inhibit else
        sync when ext_sync else
        instruction;
```



Selected Signal Asignment

- A single expression selects which of several transforms is to be applied:

```
with opcode select
   result <=
        A and B when and_op,
        A or B when or_op,
        A xor B when xor_op,
        not A when not_op;
```

- The expression is evaluated when any input changes.
- The waveform associated with the value is assigned to the target.
- Every possible value of the expression must have a corresponding waveform.

Conditional Signal Assignment

- A series of conditions controls which of several functional transforms drives the target:

```
count <=
        3 when A='1' and B='1' else
        2 when A='1' else
        1 when B='1' else
        0;
```

- When any input changes, the conditions are evaluated in order.
- The waveform associated with the first true condition is assigned to the target.
- The last waveform must not have a condition – it is the default.

Modeling a Latch

- A device with input control can be modeled with a guarded block.

```
latch: block (load_enable = '1')
       begin
         data <= guarded Dbus;
       end block latch;
```

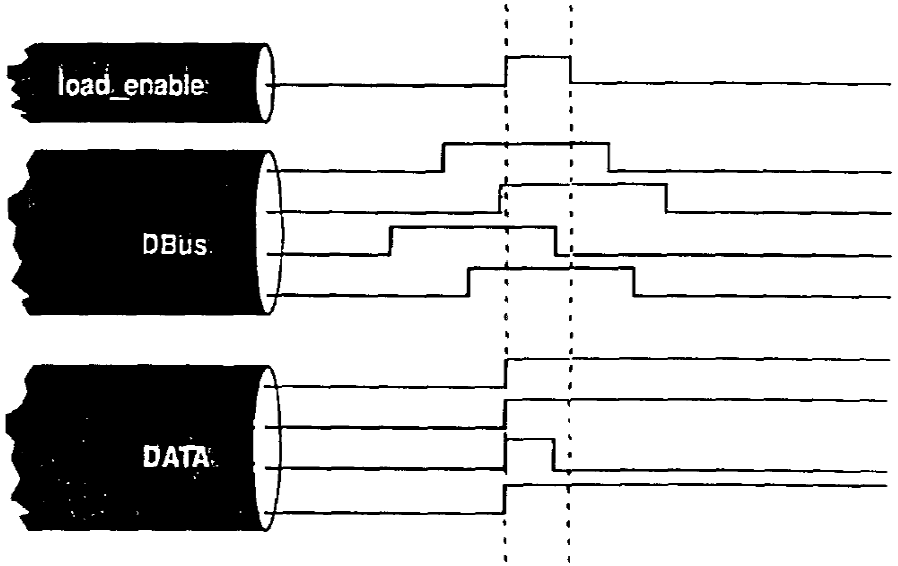
Uses of Selected Signal Assignment

- Selected signal assignment models multiplexors, selectors and similiar devices:

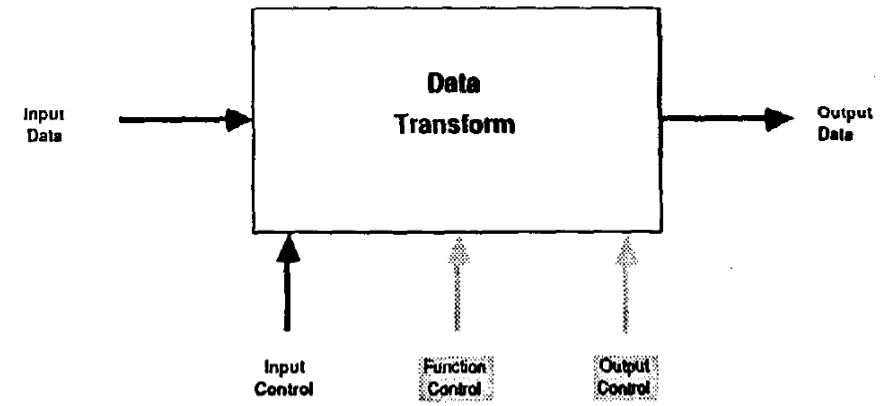
```
SM: with system_state select
next_state <=
  target when idle | test,
  fire when target,
  evaluate when fire,
  idle when evaluate;
```

```
DEC: with P1 & P2 select
SOUT <=
  "1000" after 12 ns when "00",
  "0100" after 12 ns when "01",
  "0010" after 12 ns when "10",
  "0001" after 12 ns when "11";
```

Latch Timing



Input Control



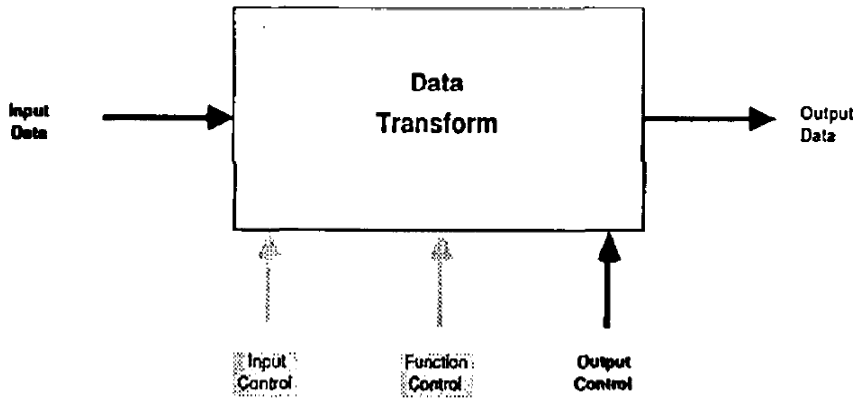
Modeling the Shift Register

```
shifter:
block (SH = '1' and rising (CLK))
    signal v: bit_vector (1 to 4); -- represents internal storage of register
 begin
    with LR & S1 & S2 select
        v <= guarded
              Q    &  v(1) &  v(2) &  v(3) when "000",
              '0'  &  v(1) &  v(2) &  v(3) when "001",
              '1'  &  v(1) &  v(2) &  v(3) when "010",
              v(4) &  v(1) &  v(2) &  v(3) when "011",
              v(2) &  v(3) &  v(4) &  Q    when "100",
              v(2) &  v(3) &  v(4) &  '0'  when "101",
              v(2) &  v(3) &  v(4) &  '1'  when "110",
              v(2) &  v(3) &  v(4) &  v(1) when "111";
    output <= v after 10 ns;
end block shifter;
```

Signal GUARD

- A block statement may have a ***guard expression*** .
- A block statement with a guard expression implicitly declares a signal named GUARD.
- The value of GUARD is always updated to the current value of the guard expression.
- Guarded signal assignments implicity reference signal GUARD.
- Signal GUARD may also be explicitly referenced like any other signal.

Output Control

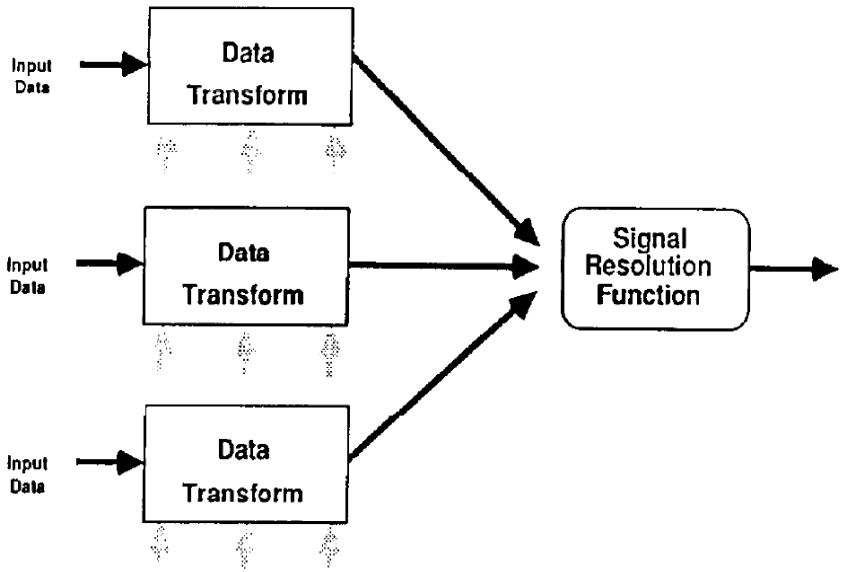


A Synchronous Serial-Input Shift Register

CLK	SH	LR	S1	S2	A	B	C	D
0	X	X	X	X	v(1)	v(2)	v(3)	v(4)
R	0	X	X	X	v(1)	v(2)	v(3)	v(4)
	1	0	0	0	Q*	v(1)	v(2)	v(3)
		0	0	1	0	v(1)	v(2)	v(3)
		0	1	0	1	v(1)	v(2)	v(3)
		0	1	1	v(4)	v(1)	v(2)	v(3)
		1	0	0	v(2)	v(3)	v(4)	Q*
		1	0	1	v(2)	v(3)	v(4)	0
		1	1	0	v(2)	v(3)	v(4)	1
		1	1	1	v(2)	v(3)	v(4)	v(1)

*Q : senal input

One Signal with Multiple Sources



Modeling Output Disconnection

- Tristate outputs can effectively disconnect from the circuits they drive.
- Output disconnection can be modeled indirectly by defining a data type that includes a value meaning "not driving":

```
type MVL is ('0', '1', 'Z', 'E');
```

Signal Resolution

```
package tristate is
                .
                .
                .
    type MVL_vector is array (integer range <>) of MVL;
    function tri_resolve (sources: MVL_vector) return MVL;
    type tri_vector is array (integer range <>) of tri_resolve MVL;
                .
                .
                .
end package tristate;
```

Using a "Not Driving" Value

```
signal common: tri_vector (0 to 7);
signal A_data, B_data, C_data, D_data: tri_vector (0 to 7);
signal A_enable, B_enable, C_enable, D_enable: MVL;
                .
                .
                .
U1: common <= A_data when A_enable = '1' else "ZZZZZZZZ";

U2: block (load = '1')
        signal tmp: tri_vector (0 to 7);
    begin
        tmp <= guarded B_data;
        common <= tmp when B_enable = '1' else "ZZZZZZZZ";
    end block U2;

U3: with C_enable & D_enable select
        common <=
            C_data when "10",
            D_data when "01",
            C_data or D_data when "11",
            "ZZZZZZZZ" when "00";
```

Summary

- Concurrent signal assignment provides data flow and register transfer level descriptive styles.
- Function control is modeled with conditional and selected signal assignment.
- Input control is modeled with guard expressions and guarded assignment.
- Output control can be modeled by using appropriate data types in conjunction with signal assignment statements.

Signal Resolution

```
package body tristate is
        .
        .
        .
function tri_resolve (sources: MVL_vector) return MVL is
    variable resolved_value: MVL := 'Z';
begin
  for i in sources'range loop
    if resolved_value = 'Z' then
      resolved_value := sources(i);
    elsif sources (i) /= 'Z'
      return 'E';
    end if;
  end loop;
  return resolved_value;
end tri_resolve;
        .
        .
        .
end package tristate;
```

Introduction to VHDL

A First Example

Basic Building Blocks of VHDL Descriptions

Structural Description in VHDL

Data Flow Description in VHDL

Behavioral Description in VHDL

Using a "Not Driving" Value

```
signal common: tri_vector (0 to 7);
signal A_data, B_data, C_data, D_data: tri_vector (0 to 7);
signal A_enable, B_enable, C_enable, D_enable: MVL;
                    .
                    .
                    .
U1: common <= A_data when A_enable = '1' else "ZZZZZZZZ";

U2: block (load = '1')
        signal tmp: tri_vector (0 to 7);
    begin
        tmp <= guarded B_data;
        common <= tmp when B_enable = '1' else "ZZZZZZZZ";
    end block U2;

U3: with C_enable & D_enable select
        common <=
            C_data when "10",
            D_data when "01",
            C_data or D_data when "11",
            "ZZZZZZZZ" when "00";
```

Behavioral Architecture of the RAM

```
architecture behavioral of RAM_1K8 is
begin
    process
        subtype matrix_item is tri_vector (7 downto 0);
        type matrix is array (0 to 1023) of matrix_item;
        variable memory: matrix;
    begin
        while CS1 and CS2 = '1' loop
            case RW is
                when '0' => DATA <= memory(intval(ADDR)) after 70ns;
                when '1' => memory(intval(ADDR)) := DATA;
            end case;
            wait on CS1, CS2, DATA, ADDR, RW;
        end loop;
        DATA <= "ZZZZZZZZ" after 55 ns;
        wait on CS1, CS2;
    end process;
end behavioral;
```

Modeling Complex Behavior

- Concurrent signal assignment statements model simple devices whose outputs are always a function of their inputs.
- A more general capability is required to model devices with internal (hidden) state information.
- A technique is required to describe device behavior in algorithmic (sequential) terms.
- The ***process statement*** provides a compact representation for arbitrary deterministic behavior.
- The ***subprogram*** encapsulates a sequence of sequential statements.

A Package for Byte Objects

```
use work.tristate.all;
package byte_objects is
    type byte_vector is array (natural range <>) of byte;
    type byte_sequence is file of byte;
    function read_bytes (file_name: string; length: natural) return byte_vector;
end byte_objects;

package body byte_objects is
    function read_bytes (file_name: string; length: natural) return byte_vector is
        file data: byte_sequence is in file_name;
        variable elements: byte_vector ( 0 to length-1);
    begin
        for i in elements'range loop
            read (data, elements (i));
        end for;
        return elements;
    end read_bytes;
end byte_objects;
```

Modeling A Random Access Memory

```
use work.tristate.all;
entity RAM_1K8 (
    DATA:       inout tri_vector;
    ADDR:       bit_vector;
    CS1, CS2:   bit;
    RW:         bit)
end RAM_1K8;
```

Behavioral Description

- Processes are concurrent statements; function calls appear in expressions; procedure calls are sequential statements.
- Processes and subprograms contain:
  - Declarations of constants, variables, subprograms, etc.
  - Sequential statements specifying how output values are computed.
- The description contains no information about the implementation of a device.

Files

- A *file type definition*
  - Gives the file type a name
  - Specifies what sort of objects it contains
  - Implicitly declares READ and WRITE procedures
- A *file declaration*
  - Creates an object of some file type
  - Associates the object with an external file
  - Specifies whether the file is to be read or written

More, More, More!

- Generic parameters
- Guarded signals
- Libraries and configuration
- Global signals
- User defined attributes
- Assertion of operating conditions and characteristics
- The VHDL type model
- Signals, events and the simulation cycle
- Modeling methods

A Self-Initializing ROM

```
use work.tristate.all, work.byte_objects.all;
entity ROM_1K8
    port ( DATA: out byte; ADDR: in bit_vector (9 downto 0);
      CS1, CS2: in bit);
end ROM_1K8;

use work.twos_comp.all;
architecture functional of ROM_1K8 is
    constant address_bits: integer := 10;
    constant file_name: string := "SYS$LOGIN:[JON.DATA]R1234";
    constant ROM_contents: byte_vector :=
                read_bytes (file_name, 2**address_bits);
begin
    DATA <=ROM_contents (intval ( ADDR)) when CS1&CS2 = "11"
                else high_Z;
end functional;
```