Behavioral VHDL

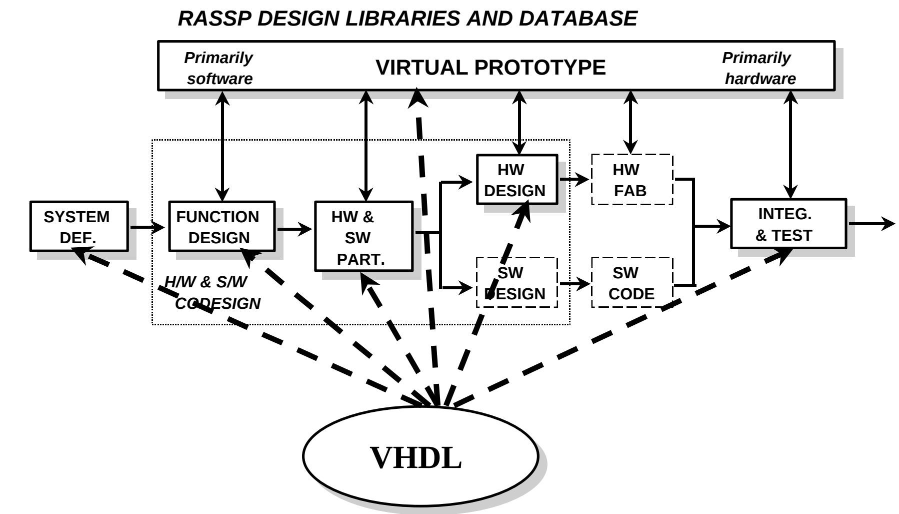
RASSP Education & Facilitation
Module 12

Version 3.00

Methodology
RASSP
Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

RASSP Roadmap

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL



Module Goals

- **Increase comprehension of behavioral VHDL constructs**
- **Expand knowledge of VHDL concepts and syntax**
- **Assist in understanding the application of behavioral VHDL to a real example**

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Outline

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **Introduction**
- **Behavioral Modeling**
  - **Processes**
  - **Sequential statements**
  - **Testbenches**
  - **Subprograms**
  - **Bus Resolution**
  - **Blocks and Guards**
  - **Packages**
  - **Problems to avoid**
- **Examples**
- **Summary**

The purpose of this module is to acquaint the student with how to define behavioral models using VHDL.

The basics of VHDL behavioral modeling are discussed and illustrated with examples.

Upon completion of this module, it is hoped the student has both the knowledge and practical understanding to create efficient and useful VHDL models at the behavioral level.

Methodology RASSP Reinventing Electronic Design Architecture Infrastructure
DARPA • Tri-Service

Introduction to Behavioral Modeling in VHDL

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **Abstraction levels of VHDL models**
  - **Structural level**
  - **Behavioral/structural mixed (i.e., data flow)**
  - **Behavioral**
- **Behavioral Modeling**
  - **Functional performance is the goal of behavioral modeling**
  - **Timing optionally included in the model**
  - **Software engineering practices should be used to develop behavioral models**
    - **Structured design**
    - **Iterative refinement**
    - **Abstract data typing**
    - **Loose coupling, strong cohesion**

Using VHDL, a system designer can model a circuit (i.e. a component or system) at multiple levels of abstraction. In prior lessons, we have concentrated on the basic elements and the structural forms of describing models in VHDL. In this module we concentrate on the behavioral view, i.e. describing how the circuit is to perform.

In behavioral modeling, we are vitally interested in the functionality of the circuit and less interested in its structural composition. At the highest levels of behavioral abstraction, we may even ignore timing.

When modeling in VHDL, it is important to follow standard practices of software engineering. Otherwise, the model may be hard to maintain, even by the person who wrote it. In addition, to aid the reuse of models, even "throw-away" models should be created with care, and with the thought that others may use it.

Typical model design and coding practices include structuring the design, iteratively refining a high-level view of the model down to its final form, and organizing the individual model components so that they are loosely coupled (small number of interface signals) and have strong cohesion (keep strongly related functions in the same architectural body).

Example Behavioral VHDL Model


```
USE TEXTIO.all, mypackage.all;
ENTITY module IS
  PORT (X, Y: IN BIT; Z: out BIT_VECTOR(3 DOWNTO 0));
END module;
ARCHITECTURE behavior OF module IS
  SIGNAL A, B: BIT_VECTOR(3 DOWNTO 0);
BEGIN
  A(0) <= X AFTER 20 ns; A(1) <= Y AFTER 40 ns;
  PROCESS (A)
      VARIABLE P, Q: BIT_VECTOR(3 DOWNTO 0);
  BEGIN
      P := fft(A);
      B <= P AFTER 10 ns;
  END PROCESS;
  Z <= B;
END behavior;
```


This slide shows a simple example of a behavioral model. Note that the VHDL *process* is a key construct in behavioral models and much of this module is devoted to presenting VHDL features associated with *processes*.

Module Outline

- Introduction
- **Behavioral Modeling**
  - Processes
  - Sequential statements
  - Testbenches
  - Subprograms
  - Bus Resolution
  - Blocks and Guards
  - Packages
  - Problems to avoid
- Examples
- Summary

RASSP Reinventing Electronic Design
Methodology Architecture Infrastructure
DARPA • Tri-Service

VHDL Processes

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **A VHDL process statement is used for all behavioral descriptions**
- **Example simple VHDL process:**

```
ARCHITECTURE behavioral OF clock_component IS
BEGIN
  PROCESS
    VARIABLE periodic: BIT := '1';
       BEGIN
         IF en = '1' THEN
           periodic := not periodic;
         END IF;
         ck <= periodic;
         WAIT FOR 1 us;
  END PROCESS;
END behavioral;
```


We now turn our attention to the VHDL *process* statement. The process is the key structure in behavioral VHDL modeling. A process is the only means by which the executable functionality of a component is defined. In fact, for a model to be capable of being simulated, all components in the model must be defined using one or more processes.

Statements within a process are executed sequentially (although care needs to be used in signal assignment statements since they do not take effect immediately; this was covered in the VHDL Basics module when the VHDL timing model was discussed). Variables are used as internal place holders which take on their assigned values immediately.

All processes in a VHDL description are executed concurrently. That is, although statements within a process are evaluated and executed sequentially, all processes within the model begin executing concurrently.

In the example process given here, the variable *periodic* is declared and assigned the initial condition '1'. As long as *en* is '1', *periodic* changes value leading to a potentially new value (called a transaction) to be scheduled for *ck* by the simulator. The process then suspends for one microsecond of simulation time. The signal *ck* actually assumes its new value one delta cycle after the process suspends . After the one microsecond suspension, the process once again executes beginning with the IF statement. Note that only variables can be declared in a process, and signals (declared outside of a process) are used primarily for control (e.g., *en* in this case), inputs into a process, or outputs from a process (e.g., *ck* in this case).

Process Syntax


```
[ process_label 8 ] PROCESS
[( sensitivity_list )]

  process_declarations

BEGIN

  process_statements

END PROCESS [ process_label ] ;
```

NO SIGNAL DECLARATIONS!

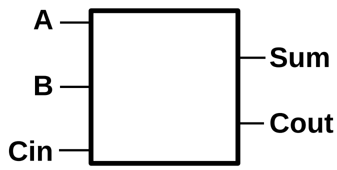
The use of *process_label* at the beginning and end of a process is optional but recommended to enhance code readability.

The *sensitivity_list* is optional in that a process may have either a *sensitivity_list*, or it must include WAIT statements. However, a process cannot include both a *sensitivity_list* and WAIT statements. WAIT statements will be covered in a subsequent section.

The *process_declaration* includes declarations for variables, constants, aliases, files, and a number of other VHDL constructs.

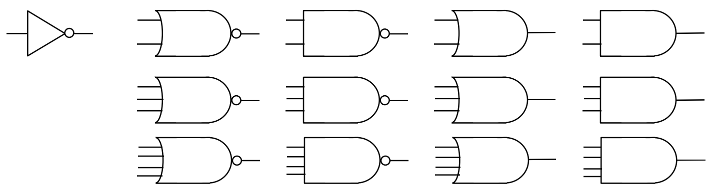
The *process_statements* include variable assignment statements, signal assignment statements, procedure calls, wait statements, if statements, while loops, assertion statements, etc.

Let's Write a VHDL Model ...




```
ENTITY full_adder IS
    PORT ( A, B, Cin : IN BIT;
           Sum, Cout : OUT BIT );
END full_adder;
```

Can we build the Full Adder's architecture using these gates?



As was seen in an earlier module, a VHDL model contains an entity and an architecture. Here the entity, which defines the model's interface to the outside world, is shown.

DARPA ● Tri-Service

Full Adder Architecture

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

for Cout (I.e. Carry Out):

	Cin (I.e. Carry In)	
A B	**0**	**1**
0 0	0	0
0 1	0	1
1 1	1	1
1 0	0	1

for Sum:

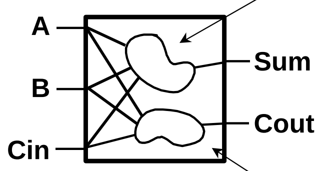
	Cin (I.e. Carry In):	
A B	**0**	**1**
0 0	0	1
0 1	1	0
1 1	0	1
1 0	1	0

A one-bit full adder will be used in the next few pages as an ongoing example.

One way to describe the function of a full-bit adder is as a look-up table. In other words, we can define every mapping of the inputs to the outputs, and encode them as a case statement in the body of the process. Here, the logic tables used to generate the outputs, *Sum* and *Cout*, are shown.

Two Full Adder Processes

```
Summation:
PROCESS( A, B, Cin)
BEGIN
   Sum <= A XOR B XOR Cin;
END PROCESS Summation;
```




```
Carry:
PROCESS( A, B, Cin)
BEGIN
  Cout <= (A AND B) OR
    (A AND Cin) OR
    (B AND Cin);
END PROCESS Carry;
```


Alternatively, the *Sum* and *Carry* functions of a full-bit adder can each be represented in VHDL with a single sequential assignment statement:

Sum <= *A* XOR *B* XOR *Cin*;

Carry <= *A* AND *B* OR *A* AND *Cin* OR *B* AND *Cin*;

We can represent these two functions each in separate process statements (but both in the same architecture), as shown above, or together in the same process statement.

In the example shown here, the sensitivity lists contain all the signals on the right-hand side of the signal assignment statements. This allows any change on any of the right-hand-side signals to cause an evaluation of the VHDL statements to determine a potential new value for the output signals.

Methodology RASSP Reinventing Electronic Design Architecture Infrastructure DARPA ● Tri-Service

Complete Architecture

RASSP E&F SCRA • GT • UVA Raytheon • UCinc • ADI

```
ARCHITECTURE example OF full_adder IS
   -- Nothing needed in declarative block...
BEGIN

 Summation:  PROCESS( A, B, Cin)
   BEGIN
       Sum <= A XOR B XOR Cin;
 END PROCESS Summation;

 Carry:  PROCESS( A, B, Cin)
   BEGIN
       Cout <= (A AND B) OR
               (A AND Cin) OR
               (B AND Cin);
 END PROCESS Carry;

END example;
```


Now we put the entire architecture together. The two process defined on the previous slide are placed in the same architecture. Note that the *Sum* and *Carry* processes execute concurrently.

This model does not use explicit time (that is, there are no AFTER phrases or "wait for" statements. Thus, this model is purely functional. If timing is important, delay phrases (i.e., AFTER clauses) can be added to the signal assignment statements, or "WAIT FOR" statements can be added in the processes.

Note that if wait statements are used in a process, the process cannot have a sensitivity list.

Alternate Carry Process


```
Carry:  PROCESS( A, B, Cin)
   BEGIN
       IF ( A = '1' AND B = '1' ) THEN
              Cout <= '1';
       ELSIF ( A = '1' AND Cin = '1' )
THEN
              Cout < = '1';
       ELSIF ( B = '1' AND Cin = '1' )
THEN
              Cout <= '1';
       ELSE
              Cout <= '0';
       END IF;
 END PROCESS Carry;
```


Alternatively, the *Carry* output could have been described using programming language constructs instead of the logic equations shown previously. Here, a set of nested if-then-else statements is used to implement the table lookup method. A case statement could also be used.

VHDL Sequential Statements

- **Assignments executed sequentially in processes**
- **Sequential statements**
  - **{Signal, variable} assignments**
  - **Flow control**
    - **IF <condition> THEN <statements> [ELSIF <statements] [ELSE <statements>] END IF;**
    - **FOR <range> LOOP <statements> END LOOP;**
    - **WHILE <condition> LOOP <statements> END LOOP;**
    - **CASE <condition> IS WHEN <value> => <statements>**
{WHEN <value> => <statements>}
[WHEN others => <statements>]
END CASE;
  - **WAIT [ON <signal>] [UNTIL <expression>] [FOR <time>] ;**
  - **ASSERT <condition> [REPORT <string>] [SEVERITY <level>] ;**

Sequential statements are used within processes and are executed in a top-down fashion. The illustrative (but incomplete) list shown on this page includes many of the commonly used forms. The VHDL Language Reference Manual provides a complete list.

Many of these statement types will be explained in further sections of this module. Some of you may note that these control structures operate almost exactly like their counterparts in Ada except for the assert and sequential signal assignment statements.

A Design Example

2-bit Counter


```
ENTITY count2 IS
   GENERIC(prop_delay : TIME := 10 ns);
   PORT (clock : IN BIT;
      q1, q0: OUT BIT);
END count2;

ARCHITECTURE behavior OF count2 IS
BEGIN
   count_up: PROCESS (clock)
      VARIABLE count_value: NATURAL := 0;
   BEGIN
      IF clock='1' THEN
         count_value := (count_value+1) MOD 4;
         q0 <= bit'val(count_value MOD 2) AFTER prop_delay;
         q1 <= bit'val(count_value/2) AFTER prop_delay;
      END IF;
   END PROCESS count_up;
END behavior;
```


In this example, we show a model for a simple 2-bit counter which counts clock pulses. The component has *clock* as an input, and two outputs which represent the LSB and MSB of a two-bit unsigned number.

Several of the constructs used above have not been shown before in this series of educational modules. For example, bit'val(*count_value* mod 2) is a function which returns a value of type bit; count_value is a natural number (i.e., an integer greater than, or equal to, zero); count_value mod 2 returns the LSB value of the counter value; but the LSB value is of type natural. Since we want the LSB to be of type bit instead, we cast it by using the 'val (read as "tic val") attribute on the type bit.

Also note the use of the generic parameter used to facilitate the assignment of delays in the signal assignment statements.

Methodology
RASSP
Reinventing Electronic Design
Architecture Infrastructure
DARPA • Tri-Service

The Wait Statement

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **The *wait* statement causes the suspension of a process statement or a procedure**
- **wait [sensitivity_clause] [condition_clause] [timeout_clause] ;**
  - **sensitivity_clause ::= ON signal_name { , signal_name }**

    ```
    WAIT ON clock;
    ```
  - **condition_clause ::= UNTIL boolean_expression**

    ```
    WAIT UNTIL clock = ‘1’;
    ```
  - **timeout_clause ::= FOR time_expression**

    ```
    WAIT FOR 150 ns;
    ```


Wait statements are used to suspend the execution of a process until some condition is satisfied. Processes in VHDL are actually code loops. The execution of the last statement in the process is followed by the execution of the first statement in the process, and process execution continues until a wait statement is reached. For this reason, every process must have at least one wait statement (a sensitivity list is actually an implied wait statement which will be described in the next page of this module).

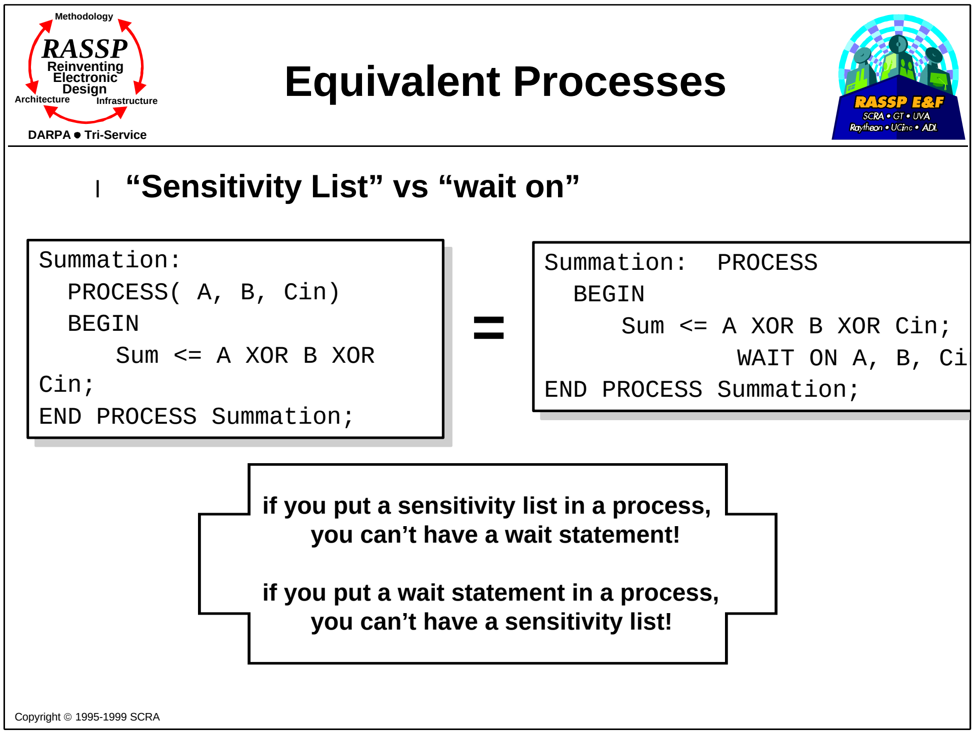
The structure of a wait statement contains optional clauses which can be used in any combination:

The sensitivity_clause: the wait statement will only evaluate its condition clause when there is an event (i.e. a change in value) on at least one of the signals listed in the sensitivity_clause. If no sensitivity_clause is given, the signals listed in the condition_clause constitute an implied sensitivity_clause.

The condition_clause: an expression which must evaluate to TRUE in order for the process to proceed past the wait statement. If no condition_clause is given, a TRUE value is implied when there is an event on a signal in the sensitivity_clause.

The timeout_clause: specifies the maximum amount of time the process will remain suspended at this wait statement. If no timeout_clause is given, [STD.STANDARD.TIME'HIGH-STD.STANDARD.NOW] (effectively until the end of simulation time) is assumed.

Wait statements assist the modeling process by synchronizing concurrently executing processes, implementing delay conditions in a behavioral model, and establishing event communications between processes. Sometimes, wait statements are used to sequence process execution relative to the simulation cycle.



A process with a sensitivity list, as shown in the process on the left, is implemented as a process with a “WAIT ON *sensitivity_list*” as its last statement (as shown in the process on the right). This allows every process with a sensitivity list to execute once at the beginning of a simulation and suspend at the bottom waiting for a relevant signal event to occur. Note that the VHDL standard prohibits the use of both process sensitivity lists and wait statements within the same process.

"wait until" and "wait for"

- **What do these do?**

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
    WAIT UNTIL A = '1';
END PROCESS Summation;
```

```
Summation:  PROCESS
  BEGIN
    Sum <= A XOR B XOR Cin;
  WAIT FOR 100 ns;
END PROCESS Summation;
```


The first process above executes once at the beginning of the VHDL simulation and then suspends until the input *A* is assigned a value of '1' before it executes again. This cycle continues in that the process executes every time *A* is assigned a value of '1'. Note that if *A* has been established at '1' before the WAIT command is executed, the WAIT command will wait forever.

The second process also executes once at the beginning of the VHDL simulation, but it then waits for 100ns of simulation time and executes again. This cycle continues with the process executing every 100ns of simulation time.

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA • Tri-Service

Mix and Match

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **Within an architecture we have two signals and the following process**

```
DoSomething:  PROCESS
   BEGIN

       WAIT ON TheirSignal;

       OurSignal <= ‘1’;
       WAIT FOR 10 ns;

       OurSignal <= ‘0’;
       WAIT UNTIL (TheirSignal = ‘1’);

       OurSignal <= ‘1’;

END PROCESS DoSomething;
```


We show here how wait statements can be used to synchronize the execution of the process, and also how to sensitize a process to signal changes in another.

In this example, the process does not execute until *TheirSignal* changes value. Then we schedule a transaction to ‘1’ on *OurSignal* and wait for 10 ns. Note, however, that since there is no AFTER clause in the assignment for *OurSignal*, it will assume its new value in one delta cycle.

After waiting 10 ns, DoSomething assigns a value of '0' to *OurSignal*; Again, *OurSignal* will actually take on the new value after one delta cycle. Execution of DoSomething is then suspended until *TheirSignal* becomes '1'. When execution resumes, *OurSignal* is set to '1' and the process immediately repeats from the top.

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Testbenches

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **Testbench is the system's top level component**
  - **Its entity declaration does not contain any PORT signals**
  - **It instantiates all the necessary components that comprise the system**
- **Testbenches may serve three additional useful purposes:**
  - **May generate stimulus for simulation:**
    - **Behavioral descriptions can be used to generate input vectors**
  - **May apply stimulus to the entity under test**
    - **Locally declared signals can be connected to PORTS of components in the system**
  - **May compare output responses with expected values**
    - **Behavioral descriptions can be used to compare model outputs to expected responses**

The testbench is the self-contained top level component in the system hierarchy, Therefore, it does not have any I/O pins (I.e., there are no PORT signals in its entity).

In addition to instantiating the necessary components necessary to describe the system, a testbench may contain a behavioral VHDL description which may be used to generate stimulus patterns to the system as well as expected results which can then be compared with the outputs of the system's subcomponents.

It should be noted that many modern VHDL simulators provide versatile mechanisms for *forcing* or driving a VHDL model's PORT signals via simulation scripts and such, thus making the use of testbenches often unnecessary

Testbenches (Cont.)

- **Incomplete example of a testbench:**

```
ENTITY testbench IS
-- no PORT statement necessary
END testbench;

ARCHITECTURE example IS testbench
  COMPONENT entity_under_test
    PORT(...)
  END COMPONENT;
BEGIN
  Generate_waveforms_for_test;
  Instantiate_component;
  Monitoring_statements;
END example;
```

Freedom of Expression

```
ARCHITECTURE example
    OF testbench IS
   .
     .
BEGIN

    MakeReset( ResetSignal,
                     100 ns );
    MakeClock( ClockSignal,
                     10 ns );

     .

END example;
```

```
PROCESS (ResetSignal)
BEGIN
    MakeReset( ResetSignal,
                     100 ns );
END PROCESS;
```

```
PROCESS
BEGIN
       MakeClock( ClockSignal,
                     10 ns );
       WAIT ON ClockSignal;
END PROCESS;
```


Because VHDL is a rich language, there are several ways to say the same thing. This example illustrates how the concurrent VHDL statements shown on the left side (as procedure calls, actually) are equivalent to the one-statement processes shown on the right. Note that the “sensitivity list” for a process is functionally equivalent to a “wait on” statement at then end of the process (e.g. the process for MakeClock on the right).

RASSP Reinventing Electronic Design — Methodology, Architecture, Infrastructure

DARPA ● Tri-Service

Freedom of Expression (Cont.)

RASSP E&F — SCRA • GT • UVA — Raytheon • UCinc • ADL

```
ARCHITECTURE example OF
      full_adder IS
BEGIN
    Summation:  PROCESS( A, B, Cin)
      BEGIN
        Sum <= A XOR B XOR Cin;
    END PROCESS Summation;

    Carry:  PROCESS( A, B, Cin)
      BEGIN
        Cout <= (A AND B) OR
                (A AND Cin) OR
                (B AND Cin);
    END PROCESS Carry;
END example;
```

```
ARCHITECTURE example OF
      full_adder IS

BEGIN

    Sum <= A XOR B XOR Cin;

    Cout <= (A AND B) OR
            (A AND Cin) OR
            (B AND Cin);
END example;
```


As in the previous page, we see that the concurrent signal assignment statements on the right can be described using one-statement processes as seen on the left.

Signal Assignment Statements

```
ARCHITECTURE stuff OF my_entity IS
SIGNAL ThisBit : BIT;
   SIGNAL ThisBitVector : BIT_VECTOR(1 TO 
5);
   SIGNAL ThisInteger : INTEGER;
   SIGNAL ThisString : STRING(1 TO 4);
BEGIN
   ThisBit <= '1';
  ThisBitVector <= "10010";
  ThisInteger <= 567 AFTER 10 ns;
  ThisString <= "VHDL" AFTER 10 ns,
                " is " AFTER 20 ns,
                "fun!" AFTER 30 ns;
END stuff;
```

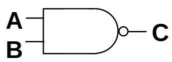

The structure of signal assignment statements allows some flexibility. However, the signal type of the result on the right hand side must match the type of the signal being assigned. This is illustrated in the first two signal assignment statements.

The third assignment shows the use of a single after clause used to control how much simulation time must pass before the assigned signal takes on its new value.

As seen in the fourth example, multiple assignments can be made in a single statements by separating them with commas. This sequence of assignments is called a "waveform".

If a signal assignment statement has no after clause, a clause equivalent to "after 1 delta cycle" is implied. Delta cycles are key to the VHDL timing model and have been previously discussed in the Basic VHDL module.

Inertial vs Transport Delays



Transport Timing

```
ENTITY nand2 IS
   PORT( A, B : IN BIT; C : OUT BIT);
END nand2;

ARCHITECTURE behavior OF nand2 IS
BEGIN
  C <= TRANSPORT NOT(A AND B)
                     AFTER 25 ns;
END behavior;
```

Inertial Timing

```
ENTITY nand2 IS
   PORT( A, B : IN BIT; C : OUT
BIT);
END nand2;

ARCHITECTURE behavior OF nand2 IS
BEGIN
  C <= NOT(A AND B) AFTER 25 ns;
END behavior;
```


Note that in the example on the left, a 20ns pulse on *A* would initially result in an assignment to C to be scheduled 25ns in the future as a result of the first transition on *A*. However, the second transition on *A* (defining the 20ns pulse) would schedule a second transition on C such that *C* would then itself show a 20 ns pulse. This leads to both the assignments to *A* being suppressed so that the inertial timing requirements are satisfied.

Since the example on the right explicitly calls for a transport delay (inertial delay is the default if neither form is specified), a 20ns pulse on *C* is allowed even when the NAND gate has a 25ns propagation delay.

Subprograms

- Similar to subprograms found in other languages
- Allow repeatedly used code to be referenced multiple times without rewriting
- Break down large blocks of code into small, more manageable parts
- VHDL provides functions and procedures

Subprograms in VHDL are in the form of functions and procedures. Functions return a value and can be used in signal and variable assignment statements:

e.g. A <= abs(-1); -- where abs() returns absolute value

Procedures, on the other hand, do not have return values but can manipulate the signals or variables passed to them as parameters:

e.g. absolute(A);

absolute() here is a procedure that directly assigns *A* its absolute value

The use of functions and procedures enables code compaction, enhances readability, and supports hierarchy by allowing code sequences that are used frequently to be defined and subsequently reused easily.

Methodology
RASSP
Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Functions

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- Produce a single return value
- Called by expressions
- Cannot modify the parameters passed to them
- Require a RETURN statement

```
FUNCTION add_bits (a, b : IN BIT) RETURN BIT IS
BEGIN  -- functions cannot return multiple values
       RETURN (a XOR b);
END add_bits;
```

```
FUNCTION add_bits2 (a, b : IN BIT) RETURN BIT IS
 VARIABLE result : BIT;  -- variable is local to function
  BEGIN
   result := (a XOR b);
   RETURN result;  -- the two functions are equivalent
END add_bits2;
```


Functions must have a return value and a return statement and cannot modify the parameters passed to them. They are called in statements where a value is needed (e.g. assignment statements, conditional tests). Note that only one value can be returned by a function call.

A function can have multiple return statements; the simulator will exit the function when it encounters the first return statement in the execution of the function.

Functions

```
ARCHITECTURE behavior OF adder IS
BEGIN
  PROCESS (enable, x, y)
  BEGIN
  IF (enable = '1') THEN
   result <=   add_bits(x, y);
   carry  <= x AND y;
  ELSE
   carry, result <= '0';
  END PROCESS;
END behavior;
```

```
FUNCTION add_bits

(a, b : IN BIT)
```

- **Functions must be called by other statements**
- **Parameters use positional association**

This example illustrates the use of a function call in a signal assignment statement where the value returned by the function add_bits() is assigned to the signal result. Also note that the parameters passed to the function during the call are associated either by position (as in the example above) or by name. In this example, *x* is associated with parameter *a*, and *y* is associated with parameter *b*.

Procedures

- May produce multiple output values
- Are invoked by statements
- May modify the parameters

```
PROCEDURE add_bits3 (SIGNAL a, b, en : IN BIT;
           SIGNAL temp_result, temp_carry : OUT BIT) IS

BEGIN -- procedures can return multiple values
   temp_result <= (a XOR b) AND en;
   temp_carry <= a AND b AND en;

END add_bits3;
```

- Do not require a RETURN statement

Unlike functions, procedures may modify multiple signals and variables in a single call. Procedures can operate on their parameters and are able to make assignments to signals and variables in their parameter lists that are of mode OUT or INOUT. A procedure call, therefore, is itself a complete VHDL statement.

Methodology RASSP Reinventing Electronic Design Architecture Infrastructure

DARPA ● Tri-Service

Procedures (Cont.)

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

```
ARCHITECTURE behavior OF adder IS
BEGIN
  PROCESS (enable, x, y)
  BEGIN
  add_bits3(x, y, enable,
          result, carry);
  END PROCESS;
END behavior;
```

- **With parameter passing, it is possible to further simplify the architecture**

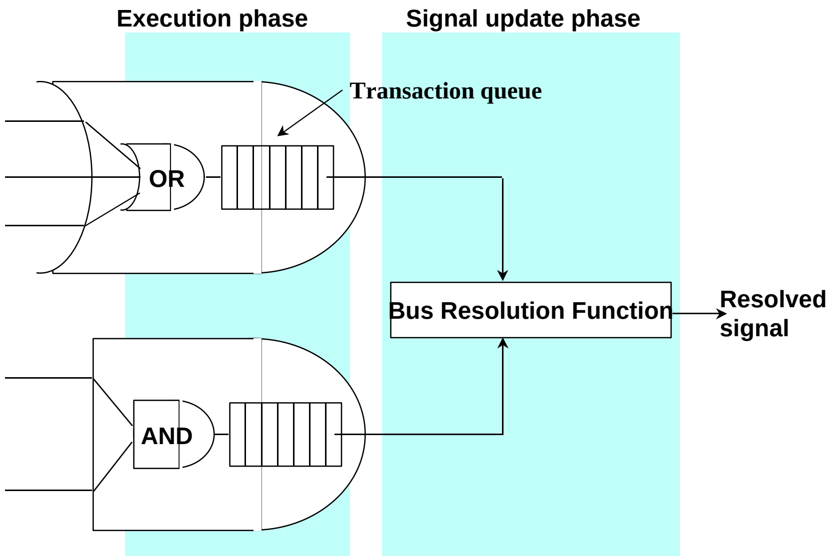
- **The parameters must be compatible in terms of data flow and data type**

```
PROCEDURE add_bits3

(SIGNAL a, b, en : IN BIT;
 SIGNAL temp_result,
    temp_carry : OUT BIT)
```


Parameter types and modes must be compatible with the signals in the parameter list during a procedure call.

Actually, procedure overloading is achieved by defining multiple procedures (or functions, for that matter) with different parameter types to distinguish among them in procedure (or function) calls.

Signal Resolution and Buses



A Bus Resolution Function (BRF) is used to determine the value of a signal that has two or more simultaneously active drivers. Each active driver provides an input to the BRF, and the BRF calculates the single value that will be read by any process using the signal as an input.

The input to the BRF is an array which includes all the active signal drivers. This input array is constructed and maintained implicitly by the simulator.

Bus Resolution
Smoke Generator

- **VHDL does not allow multiple concurrent signal assignments to an *unresolved* signal**
  - **Multiple sequential signal assignments are allowed**

```
LIBRARY attlib; USE attlib.att_mvl.ALL;
-- this code will generate an error
ENTITY bus IS
  PORT (a, b, c : IN MVL; z : OUT MVL);
END bus;

ARCHITECTURE smoke_generator OF bus IS
  SIGNAL circuit_node : MVL;
BEGIN
  circuit_node <= a;
  circuit_node <= b;
  circuit_node <= c;
  z <= circuit_node;
END smoke_generator;
```


Each concurrent signal assignment statement or process has a driver for any signal being assigned. Special care must be used when multiple concurrent signal assignment statements and/or processes drives the same signal.

Note that multiple signal assignment statements within the same process (i.e. sequential signal assignment statements) are allowed because they are executed sequentially and all use the one signal driver of their process.

Methodology RASSP Reinventing Electronic Design Architecture Infrastructure DARPA ● Tri-Service

Bus Resolution Functions

RASSP E&F SCRA • GT • UVA Raytheon • UCinc • ADI

- **Are used to determine the assigned value when there are multiple signal drivers to the same signal**

```
FUNCTION wired_and (drivers : MVL_VECTOR) RETURN MVL IS
  VARIABLE accumulate : MVL := '1';
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := accumulate AND drivers(i);
  END LOOP;
  RETURN accumulate;
END wired_and;
```

- **Bus resolution functions may be user defined or called from a package**

To review, bus resolution functions are used to determine the value assigned to a signal connected to two or more active drivers. The input to the bus resolution function is a VHDL simulator-generated array of the active drivers to the signal in question. The user-defined bus resolution function can use this array of drivers to determine what value the signal will have at all ports where it will be read. Examples include wired-or and wired-and functions, but the user may define much more sophisticated abstract functions based on user-defined status fields, etc.

Note that Bus Resolution Functions are ordinary functions except for the fact that they are called implicitly by the simulator rather than explicitly by the VHDL programmer. Also note that the input is an array of signals each of which is the same type as the returned signal.

Each process that makes an assignment to a particular signal is a driver of that signal. Note that concurrent signal assignment statements are equivalent to one line processes; thus, no two concurrent signal assignment statements may make assignments to the same signal without a bus resolution function.

Bus Resolution

Smoke Generator Fixed

- **A signal which has a bus resolution function associated with it may have multiple drivers**

```
LIBRARY attlib; USE attlib.att_mvl.ALL;
USE WORK.bus_resolution.ALL;

ENTITY bus IS
  PORT (a, b, c : IN MVL; z : OUT MVL);
END bus;

ARCHITECTURE fixed OF bus IS
  SIGNAL circuit_node : wired_and MVL;
BEGIN
  circuit_node <= a;
  circuit_node <= b;
  circuit_node <= c;
  z <= circuit_node;
END fixed;
```


Because the the BRF *wired_and* is associated with the signal *circuit_node* above, the VHDL simulator can use the BRF to determine the value to assign to *circuit_node* even though it has multiple active drivers.

RASSP Reinventing Electronic Design — Methodology, Architecture, Infrastructure — DARPA • Tri-Service

Null Transactions

RASSP E&F — SCRA • GT • UVA — Raytheon • UCinc • ADL

- How can a driver be disconnected (i.e. not influence the output at all)?
  - Use the null waveform element
- Example

```
bus_out <= NULL AFTER 17 ns;
```

- What happens if all drivers of a resolved signal are disconnected?
  - Use *register* kind in signal declaration to keep most recently determined value
  - Use *bus* kind in signal declaration if resolution function will determine the value
- Example

```
signal t : wired_bus BUS;
signal u : BIT REGISTER;
```


A NULL transaction is used to deactivate a signal driver. This is analogous to putting a tri-state driver in a high-impedance state. In such a case, the value of the signal is determined by the other active driver(s). Of course, if there is more than one active driver at any one time, a Bus Resolution Function would be needed.

There are two actions that can take place if all drivers of a signal are disconnected:

1. Use the last known value
2. Require that a bus resolution function specify a value

The keyword REGISTER is used if the last known value action is desired, and the keyword BUS is used if a bus resolution function must specify a value. The action to be used is established when the signal is declared.

Entity Statements

- **Entities may contain statements, but they can only be :**
  - **Concurrent assertion statements**
  - ***Passive* concurrent procedure calls**
  - ***Passive* process statements**
- **Example :**

```
ENTITY multiplexor IS
   PORT (a, b: IN BIT; select: IN BIT;
         output: OUT BIT);
BEGIN
   check: PROCESS(a, b)
   BEGIN
      ASSERT NOT(a=b) REPORT "a equals b"
      SEVERITY NOTE;
   END PROCESS;
END multiplexor;
```


An entity can contain passive statements to perform actions such as timing or validity checks at the interface of a component. Assertion statements in an entity, for example, may be used to check that setup and hold requirements are satisfied.

A passive statement is one which does not change the state of the system being simulated. For example, the execution of a passive statement does not lead to any signal assignments.

Blocks and Guards

- **Blocks are concurrent statements and provide a mechanism to partition an architecture description**
  - **Items declared in declarative region of block are visible only inside the block, e.g. :**
    - **signals, subprograms**
- **Blocks may be nested to define a hierarchical partitioning of the architectural description**

Blocks may be used to define a partitioning and a hierarchy within a design and to group together signal assignments and other concurrent statements which may share some common locally declared objects.

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Blocks and Guards

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

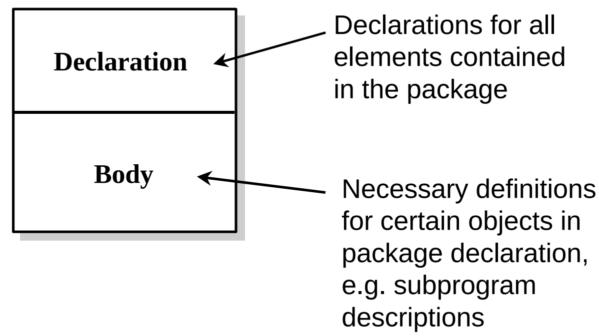
- **Unique to blocks is the GUARD construct**
  - **A *guarded* signal assignment statement schedules an assignment to the signal driver only if the GUARD expression is true. If the GUARD is false, the corresponding signal drivers are *disconnected***
  - **Example**

```
ARCHITECTURE guarded_assignments OF n_1_mux IS
BEGIN
  bi: FOR j IN i'RANGE GENERATE
    bj: BLOCK (s(j)='1' OR s(j)='Z')
       BEGIN
         x <= GUARDED i(j);
       END BLOCK;
  END GENERATE;
END guarded_assignments
```


A conditional GUARD can be included in the BLOCK declaration. If such a GUARD expression exists, then any signal assignment statement in the block which has the keyword GUARDED will disconnect the corresponding signal driver if the GUARD expression evaluates to false. This is one mechanism which can be used to guarantee that there `e only one active driver on any signal at any one time.

VHDL Packages

- **Packages encapsulate elements that can be globally shared among two or more design units**
- **A package consists of two parts**



VHDL packages are collections of reusable declarations and descriptions of VHDL types, subtypes, subprograms, aliases, constants, attributes, components, etc.

The *declaration* section of a package contains declaration statement for all the elements in the package. For several elements (e.g. TYPE definitions), the declaration is all that is needed. For some elements, however (e.g. subprograms), a functional description is also needed. This additional information is placed in the *body* section of the package.

RASSP Reinventing Electronic Design
Methodology · Architecture · Infrastructure
DARPA ● Tri-Service

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

Packages

- **Example package contents include:**
  - **Subprograms (i.e. functions and procedures)**
  - **Data and type declarations such as**
    - **User record definitions**
    - **User types and enumerated types**
    - **Constants**
    - **Files**
    - **Aliases**
    - **Attributes**
  - **Component declarations**
- **Entities and Architectures *cannot* be declared or defined in a package**
- **To use a package, it must be made visible via the *use* construct**

This slide lists many of the VHDL constructs frequently included in packages. The contents of a package are made available to other VHDL descriptions (i.e. other packages, entities, and architectures) by way of USE clauses that are analogous to the INCLUDE statements of other programming languages.

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Potential Problems to Avoid

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

- **Objects defined by subtypes derived from a base type are considered to be of the same type**
  - **Example**

```
PROCESS
 SUBTYPE smallintA IS INTEGER RANGE 0 TO 10;
 SUBTYPE smallintB IS INTEGER RANGE 0 TO 15;
   VARIABLE A: smallintA := 5;
   VARIABLE B: smallintB := 8;
   VARIABLE C: INTEGER;
   BEGIN
       B := B * A;      -- OK
       C := B + 1;      -- OK
   END;
```


Interestingly, even though VHDL is considered to be strongly typed, the developers of the language decided to stpongly type only with respect to the base type, not derived subtypes.

Thus, the VHDL analyzer will not be aware of inconsistent subtypes in the example shown here, and the simulator will execute the statements as expected. Note, however, that the result after multiplying A and B may be out of the range of B's subtype resulting in a runtime subtype range violation.

Potential Problems to Avoid (Cont.)

- **Avoid using shared variables**
  - **Debugging potential asynchronous errors very difficult**
  - **Concept likely to change in future VHDL standards**
- **Overloaded items cannot be resolved by return type**
  - **Example: These overloaded functions cannot be disambiguated**

```
FUNCTION "-" (a,b: NATURAL) RETURN INTEGER;
FUNCTION "-" (a,b: NATURAL) RETURN NATURAL;
```


The use of shared variables requires careful attention to ensure that correct values are communicated among relevant processes. For example, if one process writes a shared variable in the same simulation cycle that another process reads the variable, the VHDL standard does not define what value is read. Similarly, if two or more processes write to the same shared variable in the same simulation cycle, the standard does not define what value should be written to the variable.

Care must be taken if overloaded functions are differentiated solely by the type of their return values. The previous version of the VHDL standard, 1076-1987, did not require that differentiations on output types be supported. The current standard, 1076-1993, however, has included the requirement that differentiations based solely on output type be supported.

RASSP Reinventing Electronic Design
Methodology · Architecture · Infrastructure
DARPA ● Tri-Service

Resolving Difficulties

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADI

- **Overloaded items cannot be resolved if the argument types include common literals, i.e.,**

```
TYPE twobit IS ('0', '1');
TYPE fourbit IS ('U', '0', '1', 'Z');
FUNCTION abc (x: twobit) RETURN INTEGER;
FUNCTION abc (x: fourbit) RETURN INTEGER;
     ....
y <= abc('0')); -- Which function do we use?
```

  - **Resolve the ambiguity by qualifying the literal:**

```
y <= abc(twobit'('0'));
```

  - **General tip: Use qualification to avoid numerous problems where the compiler cannot seem to select a specific meaning, e.g., *read (abc, string'("abcabc"));***

Because literals in VHDL are semantically ambiguous (e.g., "abc" can be a string or a vector of enumerated values 'a', 'b', 'c'), it is often impossible for the VHDL analyzer to determine the exact type of a literal, and thus resolve the overloaded function, if it is dependent on the literal.

For instance, note that in the upper example, '0' appears in the definition for both enumerated types, *twobit* and *fourbit*. Therefore, calling *abc* with '0' as its parameter does not allow for a distinction between the two versions of the *abc* function.

It is a good idea to use qualification when passing literals as subprogram parameters both to ensure that inadvertent ambiguities are avoided and to improve the readability of the VHDL code.

Module Outline

- Introduction
- Behavioral Modeling
- **Examples**
- Summary

Examples

- Create a tri-state bus resolution function for a four-valued logic
- Build a state machine description of a control unit for an unsigned 8 bit multiplier
- Implement a Quicksort routine in sequential VHDL

Package with Bus Resolution Function
(Package Declaration)

```
PACKAGE resources IS

 -- user defined enumerated type
   TYPE level IS ('X', '0', '1', 'Z');

 -- type for vectors (buses)
   TYPE level_vector IS ARRAY (NATURAL RANGE <>) OF level;

 -- subtype used for delays
   SUBTYPE delay IS time;

 -- resolution function for level
   FUNCTION wired_x (input : level_vector) RETURN level;

 -- subtype of resolved values
   SUBTYPE level_resolved_x IS wired_x level;

 -- type for vectors of resolved values
   TYPE level_resolved_x_vector IS
      ARRAY (NATURAL RANGE <>) OF level_resolved_x;

 END resources;
```


This is the revised package for the *level* type with the inclusion of the necessary types, subtypes and the actual resolution function. This example simply illustrates how a bus resolution function is defined.

Note that the resolution function takes in a *level_vector,* which is really an unconstrained array, and returns a single *level* value as is required by the language. The subtype *level_resolved_x* is the signal subtype that is associated with the bus resolution function and *level_resolved_x_vector* is an array of signals of that subtype.

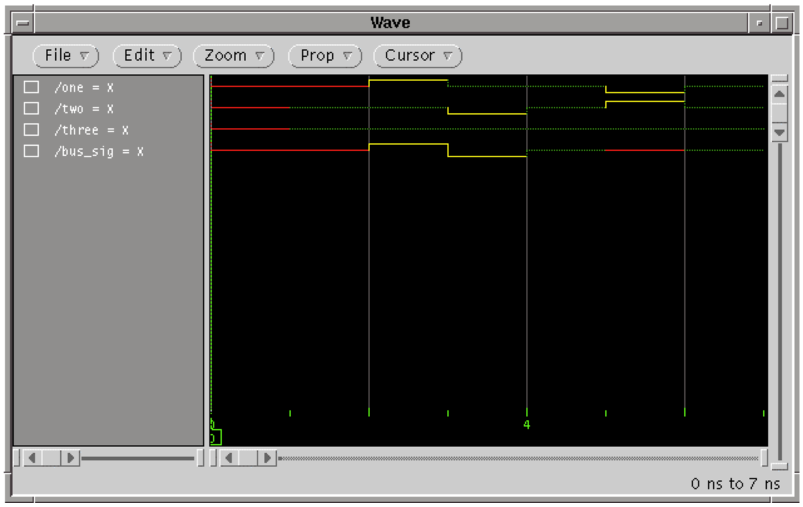
Package with Bus Resolution Function
(Package Body)

```
PACKAGE BODY resources IS
  -- resolution function
  FUNCTION wired_x (input : level_vector) RETURN level IS

    VARIABLE has_driver : BOOLEAN := FALSE;
    VARIABLE result     : level   := 'Z';
    BEGIN
      L1 : FOR i IN input'RANGE LOOP

        IF(input(i) /= 'Z') THEN
          IF(NOT has_driver) THEN
            has_driver := TRUE;
            result := input(i);
          ELSE                      -- has more than one driver
            result := 'X';
            EXIT L1;
          END IF;
        END IF;

      END LOOP L1;

      RETURN result;
  END wired_x;
END resources;
```


This is the package body showing the implementation of the *wired_x* function. Notice that the loop index spans input'RANGE which ensures that the BRF function can examine all the elements in the unconstrained array of signal drivers (i.e. the actual number of signal drivers may not be known *a priori*).

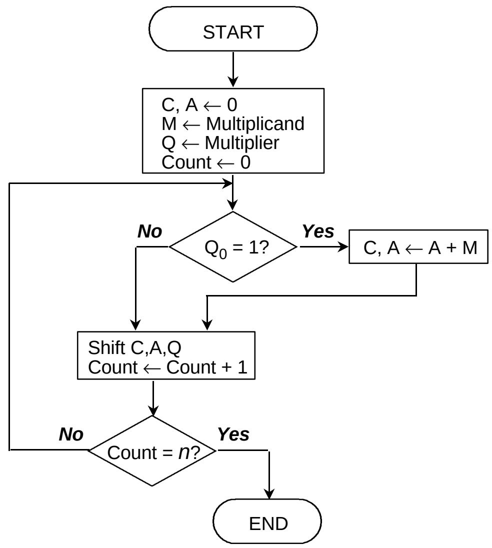
Basically, the function returns 'Z' if all drivers are 'Z', the value of any single non-'Z' driver if there is one, and an 'X' if there is more than one active driver.

Bus Resolution Function Simulation Results



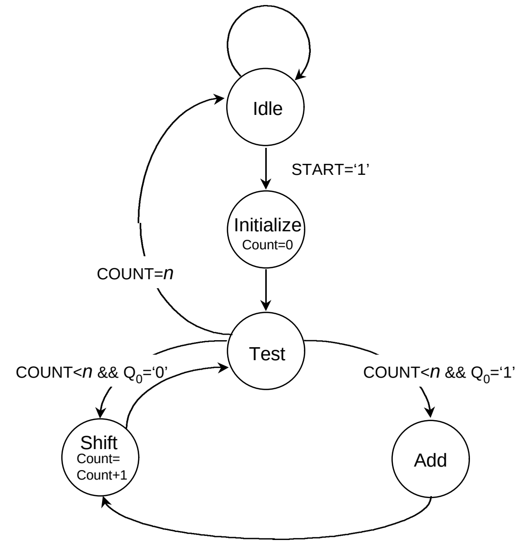
This a simulation result of three level signals driving a *level_resolved_x* signal.

Flow Chart for Unsigned 8 Bit Multiplier Controller



Now we are going to develop a behavioral description of the controller for the unsigned 8 bit multiplier. This is a flow chart of the algorithm that the controller uses.

State Diagram for Unsigned 8 Bit Multiplier Controller



This is state diagram of the controller state machine. The outputs for each state aren't shown for clarity. Notice that there is also a "count" variable that must be included in the state machine to count the number of iterations through the loop. The count variable is actually implemented as another state variable.

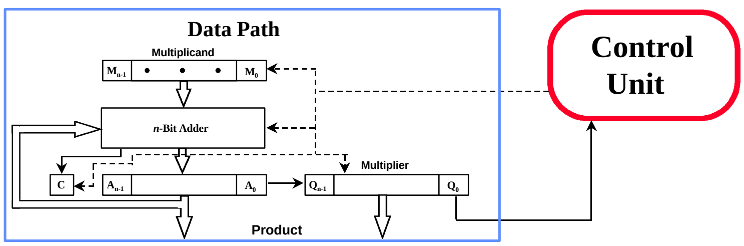
Unsigned 8 Bit Multiplier Control Unit Behavioral Description

- **Synthesizable VHDL state machine description**
- **Two internal state variables**
  - **present_state**
  - **present_count**
- **Three interacting VHDL processes**
  - **Clock (or register) process**
  - **State Transition process**
  - **Output process**
- **Moore machine**
- **Asynchronous reset signal and synchronous start signal**

The state machine actually has two state variables, the current state of the control state machine (e.g., initialize, shift, add), and the present loop count. The loop count is a state variable in that it has a present value and a next value, and it is updated in the clock process. However, the value of count only affects the next control state the machine goes to and doesn't affect the outputs. The implementation is actually more like two state machines in the same architecture.

Unsigned 8 Bit Multiplier Control Unit
(Entity)

```
LIBRARY gate_lib;
USE gate_lib.resources.all;

ENTITY mult_controller_behav IS
  PORT(reset    : IN level;   -- global reset signal
       start    : IN level;   -- input to indicate start of process
       q0       : IN level;   -- q0 ,input from data path
       clk      : IN level;   -- clock signal
       a_enable : OUT level;  -- clock enable for A register
       a_reset  : OUT level;  -- Reset control for A register
       a_mode   : OUT level;  -- Shift or load mode for A
       c_enable : OUT level;  -- clock enable for c register
       m_enable : OUT level;  -- clock enable for M register
       q_enable : OUT level;  -- clock enable for Q register
       q_mode   : OUT level); -- Shift or load mode for Q
END mult_controller_behav;
```



This the entity description for the unsigned 8 bit multiplier control unit. It hooks to the datapath via the control signals listed.

Unsigned 8 Bit Multiplier Control Unit

(Architecture - Clock Process)


```
ARCHITECTURE state_machine OF mult_controller_behav IS

  SUBTYPE count_integer IS INTEGER RANGE 0 TO 8;
  TYPE states IS (idle,initialize,test,shift,add);
  SIGNAL present_state : states := idle;
  SIGNAL next_state    : states := idle;
  SIGNAL present_count : count_integer := 0;
  SIGNAL next_count    : count_integer := 0;

  BEGIN

    CLKD : PROCESS(clk,reset)
      BEGIN
        IF(reset = '1') THEN
          present_state <= idle;
          present_count <= 0;
        ELSIF(clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN
          present_state <= next_state;
          present_count <= next_count;
        END IF;
    END PROCESS CLKD;
```


This is the beginning of the architecture of the control unit. Note that the constrained subtype of integer is for synthesis - unconstrained integers are hard to synthesize! Also note the state variables are enumerated types. This allows the synthesis tools to encode the state variable using different schemes.

Also included here is the clock process. Note that it is edge triggered and that both present_state and present_count are updated on the clock edge. Also note the asynchronous reset signal.

Unsigned 8 Bit Multiplier Control Unit

(Architecture - State Transition Process)

```
STATE_TRANS : PROCESS(present_state,present_count,start,q0)
      BEGIN
        next_state <= present_state;        -- default case
        next_count <= present_count;        -- default case
        CASE present_state IS
          WHEN idle =>
            IF(start = '1') THEN
              next_state <= initialize;
            ELSE
              next_state <= idle;
            END IF;
            next_count <= present_count;
          WHEN initialize =>
            next_state <= test;
            next_count <= present_count;
          WHEN test =>
            IF(present_count < 8) THEN
              IF(q0 = '0') THEN
                next_state <= shift;
              ELSE
                next_state <= add;
              END IF;
            ELSE
              next_state <= idle;
            END IF;
            next_count <= present_count;
```

```
  WHEN add =>
   next_state <= shift;
   next_count <= present_count;
  WHEN shift =>
   next_state <= test;
   next_count <= present_count +
   1;
  WHEN OTHERS =>
   next_state <= idle;
   next_count <= present_count;
 END CASE;
END PROCESS STATE_TRANS;
```


This is the state transition process for the state machine. Note the default assignment of next_state = present_state which is only really required for (some) synthesis tools.

Unsigned 8 Bit Multiplier Control Unit

(Architecture - Output Process)

```
OUTPUT : PROCESS(present_state)
  BEGIN
    CASE present_state IS
      WHEN idle =>
        a_enable <= '0';
        a_reset  <= '1';
        a_mode   <= '1';
        c_enable <= '0';
        m_enable <= '0';
        q_enable <= '0';
        q_mode   <= '1';
      WHEN initialize =>
        a_enable <= '1';
        a_reset  <= '0';
        a_mode   <= '1';
        c_enable <= '0';
        m_enable <= '1';
        q_enable <= '1';
        q_mode   <= '1';
      WHEN test =>
        a_enable <= '0';
        a_reset  <= '1';
        a_mode   <= '1';
        c_enable <= '0';
        m_enable <= '0';
        q_enable <= '0';
        q_mode   <= '1';
```

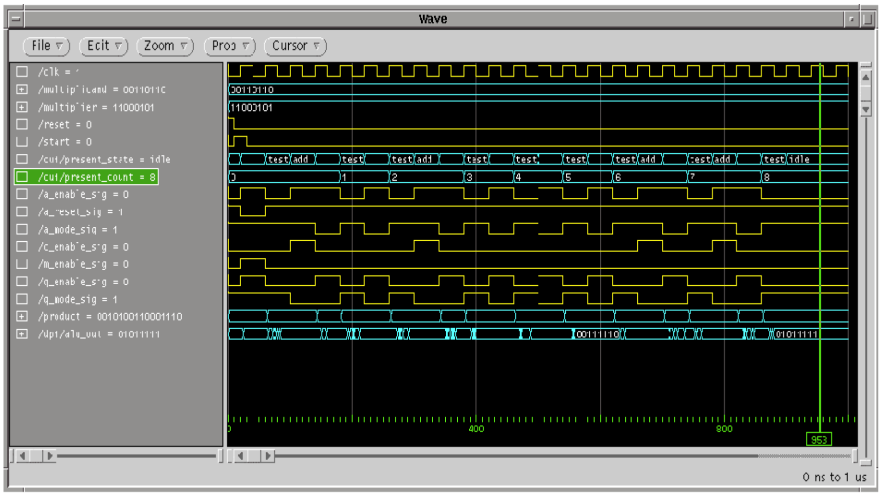
```
WHEN add =>
            a_enable <= '1';
            a_reset  <= '1';
            a_mode   <= '1';
            c_enable <= '1';
            m_enable <= '0';
            q_enable <= '0';
            q_mode   <= '0';
          WHEN shift =>
            a_enable <= '1';
            a_reset  <= '1';
            a_mode   <= '0';
            c_enable <= '0';
            m_enable <= '0';
            q_enable <= '1';
            q_mode   <= '0';
          WHEN OTHERS =>
            a_enable <= '0';
            a_reset  <= '1';
            a_mode   <= '1';
            c_enable <= '0';
            m_enable <= '0';
            q_enable <= '0';
            q_mode   <= '1';
        END CASE;
    END PROCESS OUTPUT;

END state_machine;
```


This is the output process. Note that the outputs are only dependent on the present_state variable (Moore machine).

Methodology
RASSP
Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Full Unsigned 8 Bit Multiplier Simulation Results
(Control Unit & Data Path)

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL



This is the simulation results for the full multiplier consisting of the control unit and the data path combined together in an overall structural description.

Package for Quicksort Routine

(Package Declaration)

```
PACKAGE qsort_resources IS

  CONSTANT maxarray : INTEGER := 100;

  TYPE integer_array IS ARRAY (NATURAL RANGE 0 to maxarray) OF integer;

  PROCEDURE quicksort(VARIABLE a : INOUT integer_array;
                               l : INTEGER;
                               r : INTEGER);

END qsort_resources;
```


The final example is the coding of a quicksort routine in sequential VHDL. This is the package for the quicksort code. It includes a constant for the array size (to avoid dynamic allocation), and integer array that will be sorted, and the declaration of the quicksort procedure.

In the example code for the modules, there is a C code implementation of quicksort that shadows the VHDL implementation.

Package for Quicksort Routine
(Package Body - Quicksort Procedure)


```
PACKAGE BODY qsort_resources IS

  PROCEDURE quicksort(VARIABLE a : INOUT integer_array;
                               l : INTEGER;
                               r : INTEGER) IS
    VARIABLE v, t : INTEGER;
    VARIABLE i, j : INTEGER;

    BEGIN
      IF(r > l) THEN
        v := a(r);
        i := l - 1;
        j := r;
        LOOP
          LOOP
            i := i + 1;
            EXIT WHEN(a(i) >= v);
          END LOOP;
          LOOP
            j := j - 1;
            EXIT WHEN(a(j) <= v);
          END LOOP;
          t := a(i);
          a(i) := a(j);
          a(j) := t;
          EXIT WHEN(j <= i);
        END LOOP;
        a(j) := a(i);
        a(i) := a(r);
        a(r) := t;
        quicksort(a, l, i - 1);
        quicksort(a, i + 1, r);
      END IF;
  END quicksort;
END qsort_resources;
```


This is the package body the implements the quicksort routine. Note that it is implemented recursively.

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Quicksort Routine
(Entity & Architecture)

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

```
LIBRARY STD;
USE STD.TEXTIO.all;
LIBRARY work;
USE work.qsort_resources.all;

ENTITY qsort IS
  GENERIC(infile  : STRING := "default";
          outfile : STRING := "default");
END qsort;

ARCHITECTURE test OF qsort IS

  BEGIN

    P1 : PROCESS

      VARIABLE nelements, i, tempint, temppointer :
   integer;
      VARIABLE iarray : integer_array;
      VARIABLE fresult : FILE_OPEN_STATUS := STATUS_ERROR;
      VARIABLE l : LINE;
      FILE in_fd  : TEXT;
      FILE out_fd : TEXT;
```


This is the entity and architecture that “runs” the quicksort routine on a set of data. The data is read and written from files and only variables are used to hold data. The code is sequential in that there is only one process statement and it runs one time through to completion. Therefore, there are no concurrent constructs.

Quicksort Routine
(Architecture Cont.)

```
BEGIN

  file_open(fresult,in_fd,infile,READ_MODE);
  IF(fresult /= OPEN_OK) THEN
    ASSERT FALSE
      REPORT "Usage: qvsim qsort
              -ginfile=<infile>
              -goutfile=<outfile>"
    SEVERITY FAILURE;
  END IF;

  FILE_OPEN(fresult,out_fd,outfile,WRITE_MODE);
  IF(fresult /= OPEN_OK) THEN
    ASSERT FALSE
      REPORT "Usage: qvsim qsort
              -ginfile=<infile>
              -goutfile=<outfile>"
    SEVERITY FAILURE;
  END IF;

-- read in file and set number of elements
nelements := 0;
WHILE(NOT ENDFILE(in_fd)) LOOP
  READLINE(in_fd,l);
  READ(l,iarray(nelements));
  nelements := nelements + 1;
END LOOP;
```

```
    -- find minimum element and place in
    -- element zero for sentinel
    tempint := iarray(0);
    temppointer := 0;
    FOR i IN 1 TO nelements - 1 LOOP
      IF(iarray(i) < tempint) THEN
        tempint := iarray(i);
        temppointer := i;
      END IF;
    END LOOP;
    IF(temppointer /= 0) THEN
      iarray(temppointer) := iarray(0);
      iarray(0) := tempint;
    END IF;

    -- do the quicksort!
    quicksort(iarray,0,nelements-1);

    -- write out results
    FOR i IN 0 TO nelements - 1 LOOP
      WRITE(l,iarray(i));
      WRITELINE(out_fd,l);
    END LOOP;

    WAIT;
  END PROCESS P1;

END test;
```


This is the main part of the architecture. It reads in the integer array data from a file, finds the minimum (least) element and puts it in element 0 as a sentinel (required by the quicksort routine) calls quicksort on the entire array, and then writes the result back out to a file.

Methodology
RASSP Reinventing Electronic Design
Architecture Infrastructure
DARPA ● Tri-Service

Quicksort Routine Simulation Results

RASSP E&F
SCRA • GT • UVA
Raytheon • UCinc • ADL

This is the simulation results for the quicksort showing the state of some internal variables during one of the recursive calls to quicksort. The final results are best observable in the output file.

Module Outline

- Introduction
- Behavioral Modeling
- Examples
- **Summary**

Methodology RASSP Reinventing Electronic Design Architecture Infrastructure DARPA ● Tri-Service

Summary

RASSP E&F SCRA • GT • UVA Raytheon • UCinc • ADL

- Behavioral VHDL is used to focus on the behavior, and not the structure, of the device
- Several familiar programming constructs are available, e.g CASE, IF-THEN-ELSE
- Subprograms allow large parts of code to be broken down into smaller, more manageable parts
- Bus resolution functions determine assigned value of signal with multiple active signal drivers

RASSP Reinventing Electronic Design — Methodology, Architecture, Infrastructure — DARPA ● Tri-Service

References

[Ashenden] Peter Ashenden, “The VHDL Cookbook,” Available via ftp from ftp://ftp.cs.adelaide.edu.au/pub/VHDL/VHDL-Cookbook/

[IEEE] All referenced IEEE material is used with permission.

[IEEE93] “The VHDL Language Reference Manual,” *IEEE Standard* 1076-93, 1993.

[Jain91] Ravi Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, 1991.

[Navabi93] Zain Navabi, VHDL: Analysis and Modeling of Digital Systems McGraw Hill, 1993.

[Mohanty95] Sidhatha Mohanty, V. Krishnaswamy, P. Wilsey, “Systems Modeling, Performance Analysis, and Evolutionary Prototyping with Hardware Description Languages,” *Proceedings of the 1995 Multiconference on Simulation*, pp 312-318.

[Richards97] Richards, M., Gadient, A., Frank, G., eds. *Rapid Prototyping of Application Specific Signal Processors*, Kluwer Academic Publishers, Norwell, MA, 1997