PART I THE VHDL LANGUAGE

CHAPTER 1: TOP LEVEL VHDL OVERVIEW

1.0 VHDL design Units

The VHDL language is specifically tailored to designing circuits at both behavioral and gate levels. Although it may be possible to use VHDL to program general purpose software, that is not the intention of the language. The very structure of the language suggests hardware design. There are three components to the basic VHDL program. They are: Entity, Architecture and Configuration.

Briefly, the entity describes the interface to the circuit, the architecture describes the behavior of the "black box", and the configuration binds the architecture and entity together as well as all internally declared components.

These are two other components found in more complicated VHDL programs. They are the package and package body. Briefly, the PACKAGE defines constants and interfaces to the routines found within the PACKAGE BODY. The following sections will detail these statements and also introduce the actual language constructs.

1.1 Entity

The entity construct declares the interface to the system. This interface can be viewed as a "black box". A very simple example would be to look a nand gate. The simple nand gate is pictured in Figure 1, and VHDL code in Listing 1.

(Note: In VHDL a comment is indicated by two dashes '--'. Also, by convention, reserved words are all in capitols.)

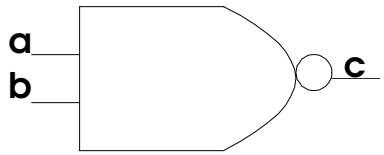

Figure 1. Simple NAND Gate

```
LIBRARY ieee;                -- use the ieee library
USE ieee.std_logic_1164.ALL;
ENTITY nand IS
   GENERIC(out_delay   : time := 5 ns); --Delay info
   PORT(a     : IN std_logic;     -- Inputs
        b     : IN std_logic;
        c     : OUT std_logic);   -- Outputs
END nand;
```

Listing 1. Flip-Flop ENTITY Statement

Here we define generics which are constants passed into components, usually counts or delays and the port statement allows us to define system I/O. The following are signal types which can be declared in the port statement:

IN	Input to the system
OUT	Output from the system
INOUT	A bi-directional SIGNAL
BUFFER	A register attached to an output. (note: it is normally impossible to read an output, but a buffer allows this)

The generic is defined with type time because it is a delay value of 5 ns, although it is possible to pass in any VHDL type. Along these lines it should be noted that throughout this document, and except for synthesis, only the IEEE standard 9 value logic system (std_logic) will be used. It is more robust than the bit or bit_vector types and is the logic system used exclusively at Sanders.

Entities are also important in that it is possible to include passive processes within the entity. The impact of this is that it is possible to include setup and hold checking. This is only one possible use, but the one of greatest importance.

1.2 Architecture

The architecture is the actual structure which makes up a given design. It is in here where we describe in some manner the actual behavior or function which defines the operation of the system. There are two methods used for writing VHDL, behavioral and structural models.

1.2.1 Behavioral Models

A behavioral model is one which defines the behavior of a system, how a system acts. We can flesh out the nand gate above in a behavioral model as follows:

```
ARCHITECTURE behavior OF nand IS
   -- declare internal signals or aliases
BEGIN
   c <= NOT(a AND b);
END behavior;
```

Listing 2. Nand Architecture

The above code could be taken as behavioral or functional as nand is a VHDL primitive. A larger purely behavioral code will be introduced later.

1.2.2 Structural Models

A Structural model can be many levels deep, starting with primitive gates and building to describe a complete system. An example of structural code would be the following RS-Flip Flop constructed from the simple NAND gate above.

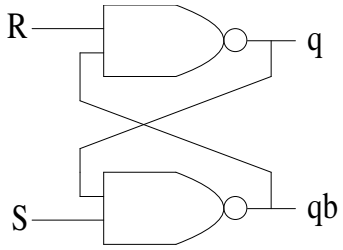

Figure 2. RS-Flip Flop

```
ENTITY rsff IS
   PORT(  r    : IN std_logic;
          s    : IN std_logic;
          q    : OUT std_logic;
          qb   : OUT std_logic);
END rsff;
ARCHITECTURE behav OF rsff IS
   COMPONENT nand   -- define our nand gate
     GENERIC(delay  : time);  -- this is a copy of the
     PORT(a    : IN std_logic; -- nand entity statement
          b    : IN std_logic;
          c    : OUT std_logic);
   END COMPONENT;
BEGIN
   u1: nand    -- instantiate u1 as a nand component
     GENERIC MAP(5 ns)   -- here you would change delay
   values
     PORT MAP(s, qb, q); -- map I/O to components

   u2: nand    -- instantiate u2 as a nand component
     GENERIC MAP(5 ns)
     PORT MAP(q, r, qb);
END behav;
```

Listing 3. RS Flip Flop Example

The above code illustrates many powerful features of VHDL. The Flip Flop (FF) is defined as two interconnected nand gates, and the nand gates are defined in separate VHDL code. This allows us to use different levels in coding. For example, we could then instantiate the RS-FF component to form a shift register, and from there a bank of registers, etc.

1.3 Configuration

The configuration section of VHDL allows us to bind our entities and architecture's together to form a single design unit. It is this which allows us to have different architecture's bound to an entity statement. This allows us to start our coding with a pure behavioral function and then to progress through synthesis or hand coding to a structural level. This level would be more akin to a gate level description of the circuit. Either of these could then be bound, through the configuration, to the entity which defines the I/O.

Also, we could have different entities, one with setup and hold checking and one without. Thus once we knew that the interface meets setup and hold times, we could forgo the time checks for an increase in simulation speed. The following is the configuration for the above RS-FF.

```
CONFIGURATION con OF rsff IS
   FOR behav
     FOR u1, u2 : nand
          USE ENTITY work.nand(behavior);
     END FOR
   END FOR
END con;
```

Listing 4. RS Flip Flop Configuration

This configuration binds the nand component to the RS FF architecture. If there were multiple definitions for nand, the compiler would be unable to know which one to use without this.

The USE clause defines which ARCHITECTURE will be bound to the configuration. The syntax is:

```
USE ENTITY library.design_entity;
USE ARCHITECTURE library.design_architecture;
```

The USE statement allows us to mix and match design units. We can substitute behavioral and gate level models by changing a line.

1.4 Packages and Package Bodies

Packages allow convenient ways of defining functions and constants which will be used in multiple VHDL programs. Packages act like an ENTITY, they declare the interfaces to the functions and subprograms found in the package body. Packages can't contain the actual subprogram or function, they must be included in the package body. Packages are also used to hide a designs complexity. For example, at Sanders, we have defined a parity function *parity()* which will return the odd parity of any *std_logic_vector()* passed into it. This is a standard package which is available to all of our designers and can be used by including it in a given piece of code.

A package body just contains the functionality of a given package, much like the architecture associated with a given entity. The difference is that only one package body can be associated with a given package. Below is the package which contains the parity function:

```
USE ieee.std_logic_1164.ALL;
PACKAGE sanders IS
   FUNCTION  parity(s  :  IN  std_logic_vector)  RETURN
   std_logic;
END sanders;
PACKAGE BODY sanders IS
   FUNCTION  parity(s  :  IN  std_logic_vector)  RETURN
   std_logic IS
     VARIABLE inter : std_logic := '0';
   BEGIN
     FOR i IN s'RANGE LOOP
          inter := inter XOR s(i);
     END LOOP;
     RETURN (NOT(inter));
   END parity;
END sanders
```

Listing 5. Parity Package

Functions are described in more detail in the function subsection.

1.4.1 Overloading

VHDL allows the use of overloaded functions. Overloaded functions are two or more functions with the same name which have different signal lists, which may return different values. This allows the creation of one function type, i.e. parity, but use different parameters. For example, we could declare two parity functions, one for std_logic types and one for bit types:

```
FUNCTION  parity(s  :  IN  std_logic_vector)  RETURN
   std_logic;
FUNCTION parity(s : IN bit_vector) RETURN bit;
```

Listing 6. Overloading

The above two functions could then be called at any point within a program. The compiler would pick the function which matched the parameters passed in. (i.e. if *parity(bit_vector)* was called, the compiler would know to use the second function and not the first.) The only

stipulation is that the functions must have unique parameters. Otherwise the compiler will not know which function to use and will generate an error.

CHAPTER 2: VHDL OBJECTS

2.0 VHDL Objects

Objects in VHDL are place holders which contain values. There are three type of VHDL objects; Signals, Variable, and Constants.

2.1 Signals

Signals are objects to which assignments are made in the future, that is, signal assignments are not instantaneous. Signals are the only way to transfer information between processes and design units.

2.2 Signal Operators

The following are pre-defined attributes which can be attached to any signal. Attributes are very useful for a variety of tasks. For example, we need to use 'STABLE and 'DELAYED for our setup and hold checks. We also use them for parity checking and many other functions are possible.

2.2.1 Signal Type Attributes

Signal type attributes return information about the signal type to which the attribute is attached. These functions are useful for exchanging data between different types. For the following example, we will use a signal of type BIT, which is defined as {'0', '1'}.

signal'LEFT

Left bound of the type.

The left bound is the leftmost value in the type declaration. For a signal of type BIT this is '0'.

signal'RIGHT

Right bound of the type.

The right bound is the rightmost value in the type declaration. For a signal of type BIT this would be '1'.

signal'HIGH

Highest value of the signal type.

The highest value is the largest value of the enumerated type. For a signal of type BIT, this would be '1'.

signal'LOW

Lowest value of the signal type.

The lowest value is the lowest value of the enumerated type. For a signal of type BIT, this would be '0'.

signal'LEFTOF(*x*)

Returns the value to the left of the input.

BIT'LEFTOF('1') would return '0'.

signal'RIGHTOF(*x*)

Returns the value to the right of the input.

BIT'RIGHTOF('0') would return '1'.

signal'POS(*x*)

Returns the position (number) of the input

BIT'POS('0') would return 0.

BIT'POS('1') would return 1.

signal'VAL(*x*)

Returns the value that corresponds to the inputs position

BIT'VAL(0) would return a '0'.

BIT'VAL(1) would return a '1'.

signal'BASE

Returns the base type of signal

If bit_vector is defined as a vector of type bit, bit_vector'BASE will return bit. Note this cannot be used in conjunction with an expression.

2.2.2. Array Attributes

These attributes apply to arrays of a type. We will define a bit_vector, bv, as: bv(5 DOWNTO 0) = "011110"

array'LENGTH[(*n*)]

Returns the number of values in the *n*th dimension, if n is not specified, it defaults to 1.

bv'LENGTH is 6

array'LEFT[(*n*)]

Returns the left bound of the *n*th dimension.

bv'LEFT is '0'.

array'RIGHT[(*n*)]

Returns the right bound of the *n*th dimension.

bv'RIGHT is '0'.

array'RANGE[(*n*)]

Returns the *n*th dimension range of array.

bv'RANGE is 5 DOWNTO 0.

array'REVERSE_RANGE[(*n*)]

Returns the range reversed

br'REVERSE_RANGE is 0 to 5

2.2.3 Signal Attributes that Return a Value

The following attributes return values based upon the signal to which they are attached.

signal'EVENT

Returns TRUE when an event occurs on the signal. An event is defined as whenever a signal has its value changed, even if the result is the same, i.e. from a '1' -> '1'.

signal'LAST_EVENT

Returns time elapsed since last event occurred

signal'LAST_VALUE

Returns previous value on signal, before the previous change

2.2.4 Signal Attributes that Return a Signal

The following attributes return a signal based upon the signal to which it is attached. These attributes are more expensive to use, as a new signal is created upon each use of an attribute. A signal is considered more expensive to use as it is globally accessible and must be constantly monitored as it may be used anywhere.

signal'DELAYED[(*time*)]

Creates a new signal delayed by *time* from the original signal.

signal'STABLE[(*time*)]

Creates a boolean signal that is TRUE when signal has not had an event for *time*.

signal'QUIET[(*time*)]

Creates a Boolean signal that is true when signal has not had a transaction for *time*. A transaction differs from an event in that a transaction is whenever a signal changes state.

signal'TRANSACTION

Creates a signal of type bit that toggles value for every transaction on signal.

2.3 Signal Drivers

Drivers are created whenever any of the following VHDL statements are used:

- Signal Assignments
- Processes
- Procedures
- Component Instantiations

A Driver contains the current value of the driver as well as a list of the scheduled future values of the driver. This future list is also known as the transaction queue. If a signal has more than one driver, then the final signal value will be a function of all effective driver values. This function is known as the signal resolution function.

```
PROCESS(b)
     BEGIN
     a <= b;
END PROCESS;
PROCESS(c)
     BEGIN
     a <= c;
END PROCESS;
```

Listing 7. Signal Contention.

The above code would cause a contention from the two process trying to drive signal a. Thus a must be defined as a resolved signal with a resolution function, or the compiler will generate an error. This is explained in the next section.

2.4 Signal Resolution Functions

A resolution function accepts an array of values and returns a value of the resolved type. The resolution function must take into account all possible input signals or an error will result when trying to resolve the

signals. VHDL comes with no standard resolution functions, all resolution functions must be user written. Currently there are many resolution functions available and most VHDL systems come with standard user written resolution functions, but they are not part of the IEEE standard. Resolution functions are the main reason that makes VHDL transportable amongst different technologies. It would be possible to write different functions for different technologies, a CMOS function, TTL function, etc. Signals with multiple drivers are called "resolved signals". A resolved signal must have a resolution function associated with it.

```
-------------------------------------------------------------------
-- Resolution function
-------------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
  --      ---------------------------------------------------     |   |
  --      |  U    X    0    1    Z    W    L    H    -      |     |
  --      ---------------------------------------------------     |---|
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), --  | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), --  | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', '0' ), --  | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', '1' ), --  | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'Z' ), --  | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W' ), --  | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L' ), --  | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'H' ), --  | H |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-' )  --  | - |
    );

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
    VARIABLE result : std_ulogic := '-';  -- weakest state default
BEGIN
    IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW);
    ELSE
        -- Iterate through all inputs
        FOR i IN s'RANGE LOOP
            result := resolution_table(result, s(i));
        END LOOP;
        -- Return the resultant value
        RETURN result;
    END IF;
END resolved;
```

Listing 8. Sample Resolution Function

2.5 Variables

Variables are assigned immediately. These are good for holding intermediate values during signal assignments or computations. Variables are only local to a process, are only be monitored within that process, and only accessible within that process.

2.6 Constants

Constants are any object declared and defined during initialization.

2.7 Generics

Generics are used to pass local constant values into the entities and architecture's of VHDL. Common uses include passing in delay information, counts and values into entities or architecture's. Generics are treated as local constants within the given entities and architecture's. Architecture generics overwrite generics within the entity. This is useful, once it allows a program which instantiates a component to declare new delays in the new upper level component. This can be useful for back annotation. Back annotation is usefull for testing synthesis results. Actual timing delays can be passed into the VHDL program for more accurate results.

Finally, generics within the configuration override all other generics. This is especially useful for back annotation, as one top level configuration can be back annotated with delays for an entire design at a later time.

CHAPTER 3: VHDL DATA STRUCTURES

3.0 VHDL Types

VHDL contains many types of data structures, from simple signals to composite types, arrays, and file types. All VHDL objects have a type associated with them.

3.1 Signals, Variables and Constants

We have already looked at these VHDL types. In review:

3.1.1 Signals

Signals represent physical data connections. All values assigned to signals are "scheduled" for a time in the future. Signals are used for communication between processes, as this is the only way to pass data between different processes. Present and future values are stored within the drivers, which allows us to use attributes on the signals.

3.1.2 Variables

Variables represent local storage internal to a process or subprogram. Variable assignments take place instantaneously, unlike signals which are scheduled.

3.1.3 Constants

Constants can only be assigned a value once, at initialization.

3.2 Scalar Types

Scalar types are data types which don't directly correspond to hardware. There are four scalar types. They are: Integer, Reals, Enumerated, and Physical.

3.2.1 Integer

VHDL represents integers in the same way as the target machine or compiler. This is to say that integers are of machine dependent range. Integers are the same as normal integers in math, and normal arithmetic functions (+, -, *, /) apply.

3.2.2 Real

Real numbers are as integers, machine specific. They allow a wider variation in values than integer, the largest range possible.

3.2.3 Enumerated

Enumerated types are user defined types. It is important when defining the order of the names, to put them in lexical order. This is important for the default initialization, 'LEFT. Names can be single characters or identifiers. Note, user defined enumerated types must have user defined functions and resolution functions, as standard VHDL functions and operators will not apply to the new type.

An enumerated type example can be anything. Here we will define a type, traffic which will have values of red, yellow or green.

```
TYPE traffic IS (red, yellow, green);
```

Listing 9. Enumerated Types

3.2.4 Physical

Physical types are defined to represent physical quantities. Examples would be: time, distance, and weight. The base unit must be declared, and then new values can be calculated from the base unit.

Physical types represent physical quantities. Here we will define a type, capac, to represent capacitance.

```
TYPE capac IS RANGE 0 to 1000000000
  UNITS
    pf;       --pico farads
    nf = 1000 pf;  -- nano farads
    uf = 1000 nf;  -- micro farads
    mf = 1000 uf;  -- milli farads;
  END UNITS;
```

Listing 10. Physical Types

3.3 Composite Types

There are two composite types available in VHDL, Arrays and Records.

3.3.1 Arrays

Arrays are groups of elements of the same type. Elements can be scalars or composite types. Arrays are useful, in that they are useful for modeling RAMs or ROMs. They are particularly useful in that specific values can be accessed by an index.

The following is a sample array of four eight bit (std_logic) words.

```
TYPE four IS ARRAY( 0 TO 7) OF std_logic_vector(7
  DOWNTO 0);
```

Listing 11. Array Example

3.3.2 Records

Records are groups of different type objects. Like Arrays, the elements can be scalar or composite types. They are extremely useful for passing data. For instance, it is possible to format a record to look like an op code and operand, further clarifying the functionality of a microprocessor system.

A simple record example would be an eight bit op code with a 16 bit operand.

```
TYPE instruction IS
   RECORD
     opcode    : std_logic_vector(7 DOWNTO 0);
     data      : std_logic_vector(15 DOWNTO 0);
   END RECORD;
```

Listing 12. Record Example

3.4 Subtypes

A subtype is a subset of a base type. Assignments between the base type and the subtypes are allowed. The use of subtypes allows the addition of constraints to increase the usefulness of range checking and minimize the number of options in a CASE statement.

An example of a subtype would be to define *positive*, the positive integers from 1 to 'HIGH. The type integer is pre-defined as the integers from 2^{32}-1 DOWNTO - 2^{32}.

```
SUBTYPE pos IS integer RANGE 0 TO integer'HIGH;
```

Listing 13. Subtype Example

Note the use of the *'HIGH* operator. This allows us to express our subtype in a more readable manner. Recall that *'HIGH* returns the largest value of integer.

3.5 Alias

Aliases are an alternative designation for a signal, variable or constant. Aliases are very useful for splitting vectors into pieces. For example, we can split up a word into op code and operand:

```
SIGNAL instruction: std_logic_vector(15 DOWNTO 0);
ALIAS op code: std_logic_vector(7 DOWNTO 0) IS
  instruction(15 DOWNTO 8);
ALIAS operand: std_logic_vector(7 DOWNTO 0) IS
  instruction(7 DOWNTO 0);
```

Listing 14. Alias Example

3.6 Unconstrained Types

Unconstrained types are useful for defining common functions with different width inputs. By using an unconstrained type, one universal function can be written. The way we can do this is to define a type with no range, using the 'box' symbol '<>'. This allows us to define an unconstrained array of integers. We must define this range when we instantiate the type.

```
TYPE std_logic_vector IS ARRAY (INTEGER RANGE <>) OF
  std_logic;
```

Listing 15. Unconstrained Types

The Instantiation would be: std_logic_vector(31 DOWNTO 0). (31 DOWNTO 0) would replace the box character.

3.7 Overloading

The concept of overloading is to create multiple subprograms with the same name, or same enumeration value in multiple types. This allows for more readable models, for instance, an AND function can be used the same on std_logic, tw_logic(a 46 value logic system) or bit types.

3.8 User Defined Attributes

User defined attributes are a user defined data passing mechanism. They allow the attachment of elements to VHDL descriptions. Attributes must consist of a type, and their values are considered constants and cannot be reassigned. Attributes can be associated with almost any VHDL construct, whereas GENERICS can only be associated with entities.

3.9 Generate

Generate statements allow the VHDL programmer to easily create repetitive structures. This allows for simple generation of memory arrays, registers and other repetitive constructs. It is possible to use IF or FOR clauses to allow conditional generation for even further control.

3.10 Access Types

Access types, similar to pointers in C, are used to perform programming language operations which have no direct hardware definition. Although they do not directly model hardware, they do allow for the creation of dynamic memory allocation for queue's, FIFO's or memories. The values of access types are pointers to dynamically

allocated types. In the following example, we declare an access type ptr_fifo, which acts as a pointer to the type fifo.

```
TYPE fifo_element IS ARRAY(0 TO 3) OF std_logic;
TYPE ptr_fifo IS ACCESS fifo_element;
```

Listing 16. Access Types

3.11 Incomplete Types

Incomplete types are used as a place holder for a type which will be defined later. They are useful for access type declarations.

3.12 File Types

File types allow for the input of files. File types may not be of a straight ASCII format. Files are of a user defined file type and can be made up of values of a particular type. The file type is defined at declaration. Types, IN or OUT, are available, INOUT is not. READ, WRITE, and end of file functions are provided in VHDL. Files are very useful in the generation of VHDL test benches.

CHAPTER 4: VHDL EXECUTION

4.0 Concurrent Processing

The statement section of the architecture executes in parallel. All statements in this section will be executed at the same time, the only exception to this is that statements within a PROCESS execute sequentially within the PROCESS, but in parallel with other statements within the statement section. We will discuss this in more detail in the sequential processing section. See the following code fragment for examples.

```
BEGIN
1  c <= a AND b AFTER delay;  -- executes in parallel
2  d <= b AND c AFTER delay;  -- executes in parallel
   PROCESS     -- executes serially, but in parallel
                    with
     BEGIN     -- other statements in this code
3         e <= a AND b AFTER delay; -- fragment
          WAIT FOR 0 ns;
4         f <= b AND c;
   END PROCESS;
END arch;
```

Listing 17. Concurrent Processing

In the above fragment 1, 2, and 3 will execute simultaneously, however statement 4 will not be executed until the next time unit, in which statements 1, 2, and 4 will be executed in parallel. The reason for this is the WAIT FOR 0ns line, which allows one time unit to pass. This is important in that it allows us to model hardware as parallel blocks of code, a whole microprocessor could be modeled at once, all functional blocks operating in parallel.

Sequential blocks are good for modeling blocks of code which operate sequentially, usually sensitive to a clock. The best example of this would be modeling a finite state machine, a sequential adder or a sequential multiplier. Another example would be modeling a pipelined system. At the top level, usually there will be sequential blocks of code wired together and executing in parallel.

4.1 Transport and Inertial Delays

Inertial delays are the default VHDL delay and the common delay type used in VHDL. The disadvantage of this type is that if a spike occurs which is smaller than the delay through the device, the spike is swallowed and lost. This may or may not reflect the true device.

Transport delays are treated as ideal delays. That is, any size pulse width will propagate through the device. This type of delay is good for wire delay and delay line modeling.

Comparing the two delays using the following code, if we input a pulse less than 20ns, the pulse will be lost. Only pulses of greater width than 20ns will be assigned to signal b.

```
b <= TRANSPORT a AFTER 20 ns; -- model a delay line
```

Listing 18. Transport delay example

4.2 Concurrent Statements

It was introduced before that VHDL simulates in parallel, and in fact the whole language is based around allowing sequential and parallel blocks to execute at the same time. The following statements or code blocks run concurrently in VHDL.

- Signal Assignments
- Component Instantiations
- Processes
- Blocks
- Procedure Call
- Assertion statements
- Generate Statements

These statements all execute in parallel when encountered within a VHDL program. Certain structures such as Processes run sequentially internally, but they also run concurrent with other statements.

4.3 Sequential Processing

Sequential processes run as any other programming language, each line is executed in sequence. The main tool for sequential processing is the PROCESS.

4.4 Processes

Processes are the basic unit of behavioral descriptions. A process is considered a series of sequential statements which represents a single action during simulation. During simulation, the process executes until a wait statement or the end of process is reached, this is all considered a single concurrent action. Processes are treated as individual components, they have access to the ports and signals declared in the entity, but may also have their own local variables, constants, types and subprograms. It is important to realize that a process will run until a condition occurs to cause it to halt execution. It is possible to set up a process to run infinitely without advancing time. This would cause the simulator to cycle the

process with no error generated or anything actually simulated. Another very important point is that since processes execute sequentially until the end of the process, only the last signal assignment to a given output is made. For example, the following code:

```
c <= a AND b AFTER 0 ns;
c <= d OR e AFTER 10 ns;
```

would be equivalent to:

```
c <= d OR e AFTER 10 ns;
```

Listing 19. Process Equivalence

The first assignment to c is lost. The original meaning to multiplex the output c with the two statements has been lost. The option to restore the original meaning would be to use transport delays or make the assignments separately and OR the results together, the latter being a more realistic interpretation.

4.4.1 Sensitive Processes

A sensitive process is one which isn't activated until a transition occurs on a signal listed in the sensitivity list. This type of process is good for asynchronous components, but not very good for synchronous ones. It is important to note, if a sensitive process is used, wait statements are not allowed and will result in an error.

```
PROCESS(a, b)
BEGIN
   c <= NOT(a AND b) AFTER delay;
END PROCESS;
```

Listing 20. Sensitive Process (nand gate)

The above process would activate only when a transition (0 -> 1 or 1 -> 0) occurs on the a or b signals. When either signal switches, c will be assigned NOT(a AND b) AFTER a delay period.

4.4.2 Non-Sensitive Process

A process which doesn't contain a sensitivity list is good for modeling synchronous processes. The reason for this, is that we can use a wait statement, WAIT UNTIL clock'EVENT and clock = '1', which will only be true on the rising edge of a clock. A sensitive process would execute on both the rising and falling clock edges. A function called *rising_edge(clock)* is used, it is equivalent to *(clock'EVENT AND clock = '1')*. See section 2.2.3 on attributes for more information. The following is an example of a process without a sensitivity list:

```
PROCESS
BEGIN
   WAIT UNTIL (rising_edge(clock));
   q <= d AFTER delay;
   qb <= NOT(d) AFTER delay;
END PROCESS;
```

Listing 21. Process Example (D Flip Flop)

4.4.3 Important Notes Regarding Synthesis

It is a good idea to introduce the idea of synthesis at this point. VHDL Synthesis tools are very different and each one has different rules to follow. At this time, however, the following general rules do apply.

- Sensitive processes do not synthesize well. Most synthesizers drop the sensitivity list.
- Use only one wait statement within a process, preferably at the top.

These general rules are good guidelines to generating processes. If you look at the *wait until* construct, you can see that a sensitive process is nothing more than a process with a *wait on* statement at the top. See the wait section, section 4.2.2.5, for more information.

4.5 Sequential Statements

The code which makes up a process is a series of sequential statements. Assert statements are the only statements which are allowed in either a process or as concurrent statements.

4.5.1 IF-THEN-ELSE

The if-then-else clause is used only within a process or function. These statements would in fact be meaningless outside of a sequential environment. A VHDL if statement is defined as follows:

```
IF (if_condition) THEN
   Statements_if_true;
ELSIF (elsif_condition) THEN  -- more than one allowed
   Statements_if_true;        -- note spelling (this is
                                 optional)
ELSIF (elsif_condition) THEN  -- more than one allowed
   Statements_if_true;        -- note spelling (this is
                                 optional)
ELSE                          -- only one allowed
   Statements;
END IF;                       -- required
```

Listing 22. VHDL IF-THEN-ELSE clause

More than one ELSIF is allowed, but only one ELSE statement is allowed.

4.5.2 Case Statement

The case statement is one of the most versatile of all VHDL statements. It has many uses, the foremost being that it is ideal for emulating state machines. Since most sequential logic operates in discrete steps, this statement will prove to be extremely useful.

```
CASE expression IS
   WHEN choice_1 =>
     statements_1;
          .
          .
   WHEN choice_n =>       -- as many as needed
     statements_n;
   WHEN OTHERS =>         -- optional "fall through"
     statements_others;
END CASE;
```

Listing 23. VHDL CASE statement

Case statements must be defined for every possible state. This is where the OTHERS choice comes in. VHDL types can have many different strengths and values. For example, std_logic has 9 values, and tw_logic has 46 values. These logic systems differentiate between different strengths of ones and zeros.

Usually you will want to test for strong types and allow weak or undefined types to generate errors. The following case statement exemplifies this:

```
PROCESS
   VARIABLE switcher : std_logic_vector(1 DOWNTO 0);
BEGIN
   CASE switcher IS
     WHEN "00" =>
          ASSERT FALSE   -- always execute
               REPORT "case 0"
                    SEVERITY note;
     WHEN "01" =>
          ASSERT FALSE   -- always execute
```

```
                REPORT "case 1"
                     SEVERITY note;
     WHEN "10" =>
          ASSERT FALSE   -- always execute
                REPORT "case 2"
                     SEVERITY note;
     WHEN "11" =>
          ASSERT FALSE   -- always execute
                REPORT "case 3"
                     SEVERITY note;
     WHEN OTHERS =>
          ASSERT FALSE   -- always execute
                REPORT "invalid state, undefined"
                     SEVERITY error;
  END CASE;
END PROCESS;
```

Listing 24. CASE statement example

Now for a finite state machine example:

```
PROCESS
  VARIABLE state : integer :=0; -- start at state 0
BEGIN
  WAIT UNTIL (rising_edge(clock)); -- make a
                                -- synchronous process
  CASE state IS
    WHEN 0 => -- initial state
          reset <= '0';  -- <= signal assignment
          state := 1;    -- := is a variable assignment
    WHEN 1 => -- state 1
          reset <= '1';
          input <= "0010100";
          state := 2;
    WHEN 2 =>
          run <= '0';
          state := 3;
    WHEN OTHERS => -- done
          ASSERT FALSE
                REPORT "done"
                     SEVERITY note;
  END CASE;
END PROCESS;
```

Listing 25. Finite State Machine example

The wait statement in the above process makes it synchronous to the rising edge of the clock. We define a variable state, which is used to point to the next state of the machine. It is initialized to 0, such that the first state will be equal to 0 and then we will step through every rising edge, because we change the state variable to point to the next state. It is important to note, only one WHEN clause will be executed during each cycle, this is in fact what allows us to do this.

4.5.3 Looping

VHDL provides functionality for looping with two constructs, the for loop and the while loop. Each loop type has its own uses.

4.5.3.1 The For Loop

For loops are useful for stepping sequentially. There are a few rules for using a FOR statement. They are:

- Can't manipulate the loop variable
- Can only step by ones
- You don't need to declare the loop variable

For loops are more limited than while loops, however they are still powerful.

```
FOR i in 10 DOWNTO 0 LOOP
   a(i) <= b(i);
END LOOP;
```

Listing 26. FOR loop example

4.5.3.2 While loops

While loops allow you to loop while the condition is true, and to exit the loop when the condition becomes false. While loops differ from for loops in the following ways:

- The loop variable needs to be manipulated by the program, all increments, decrements or changes to the variable must come from the code within the loop.
- Because of the above, the variable can be stepped in any direction by any count.
- The loop variable must be declared in the program

```
PROCESS(a)
   VARIABLE looper : integer := 0;
BEGIN
   WHILE looper < 8 LOOP
     b(looper) <= a(looper);
     looper = looper + 1; -- must include to increment
                          -- variable
   END LOOP;
END PROCESS;
```

Listing 27. While Loop Example

4.5.3.3 Exit

The exit statement can be used to break out of a loop at any time. This can be useful for modeling interrupts in a processor or error handling. An example of this appears below:

```
counter:
PROCESS
   BEGIN(clock, interrupt)
   LOOP
        i <= i + 1;
        WAIT FOR 10 ns;
```

```
        IF (interrupt = '1') THEN
             exit;
        END IF;
   END LOOP;
END PROCESS;
```

Listing 28. Exit Example

When the loop starts, it will only stop when an interrupt occurs.

4.6 Assert

The assert statement allows the programmer to generate error or status messages to the console during program execution. It is very important to use asserts, even if just to let the user know that the program is operating successfully. Simulations involving large and complex designs can take from minutes to days to simulate, and the user should be told intermittently that the code is executing and the status of it. To help with the use of asserts, there are different levels of severity. They are: Note, Warning, Error, and Failure. Asserts are good for timing checks, debugging and status information.

4.6.1 Timing checks

It is important to have good setup and hold checking, and asserts offer a very good method for performing this:

```
setup_check:
PROCESS(rising_edge(clock))
BEGIN
   ASSERT (input'LAST_EVENT > setup_time) -- will
                              -- execute if false
     REPORT "setup error"
          SEVERITY error;
END PROCESS;
hold_check:
PROCESS(clock'DELAYED(hold_time))
BEGIN
   ASSERT    (input'LAST_EVENT    >    hold_time    OR
   input'LAST_EVENT = 0)
```

```
        REPORT "hold violation"
             SEVERITY error;
END PROCESS;
```

Listing 29. ASSERT examples

4.7 WAIT

Wait statements are used for process synchronization. A wait statement in a process causes the process to suspend until the wait statement becomes true. Sensitive processes already have an implicit *wait on* statement and cannot contain a wait statement. There are three types of wait statements available.

- *wait on*, wait until a transition occurs on a signal
- *wait until*, waits until a specified condition is true
- *wait for*, wait a specified time period

The above types can be combined to form complex wait statements. A process with no wait statement or sensitivity list will be flagged as an error, as it would loop infinitely with no time passing.

4.7.1 Wait Examples

The following are different versions of the wait statement with a description of what they do.

WAIT;

Process will execute until this statement is reached and then halt forever.

WAIT ON (a, b, ... c);

Halts on this statement until an event occurs on any of the signals.

WAIT UNTIL (*expression*);

Halts program until *expression* becomes true.

WAIT FOR n ns;

Suspends execution for n nanoseconds.

4.8 Subprograms

Subprograms are functions or procedures which are called by a main VHDL program. Subprograms contain sequential VHDL statements, similar to processes. Like processes, subprograms may declare local variables, constants and sub-programs. Subprograms are not sensitive to anything, so they will execute in sequence until complete. Whenever subprograms are called, their variables are initialized, unlike processes which only initialize the first time they run. Subprograms are passed values through a parameter list, however references to higher level signals are allowed. There is a side effect to this, that is that you may modify a signal not in the parameter list.

4.8.1 Functions

Functions differ slightly from subprograms in that they return only one value and have no side effects. A function cannot affect signals not passed into the function. Below is a small function which acts as an and gate.

Function Call

```
c <= and(a, b);
```

Function Declaration

```
FUNCTION and(a : std_logic, b : std_logic) RETURN
  std_logic;
```

Function Body

```
FUNCTION and(a: std_logic, b: std_logic) RETURN
  std_logic IS
  BEGIN
    RETURN (a AND b);
  END and;
```

Listing 30. Function Example

4.8.2 Procedures

A procedure acts as a process does, but allows us to move it to a common package where it may be invoked by any component in the system. A procedure can "return" many values, and a procedure can have side effects. A procedure call looks like a function call, however no value is directly returned.

Procedure Call

```
xor(c, a, b);
```

Procedure Declaration

```
PROCEDURE xor( SIGNAL c: OUT std_logic;
               SIGNAL a, b: IN std_logic);
```

Procedure Body

```
PROCEDURE xor( SIGNAL c: OUT std_logic,
               SIGNAL a, b: IN std_logic) IS
BEGIN
  c <= a XOR b;
END xor;
```

Listing 31. Procedure Example

PART II VHDL EXAMPLES

CHAPTER 5: 8-1 MUX EXAMPLE

5.0 A Complete VHDL Example

Below is a complete 8-1 MUX constructed hierarchicaly from two 4-1 MUX's. This example will give a good idea as to how VHDL models are constructed, and that they actually can simulate hardware effectively. This is purely behavioral and of the form accepted by most synthesis tools.

5.1 Behavioral 4-1 MUX

We will construct behavioral code to represent the 4-1 MUX pictured in figure 3. This code will be purely behavioral in nature, utilizing the VHDL case statement.

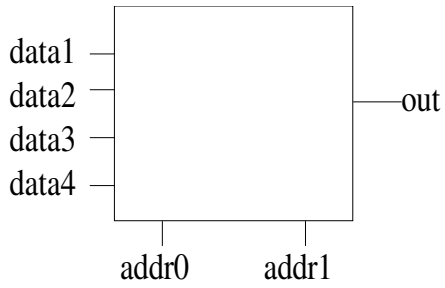

Figure 3. 4-1 MUX

5.1.1 4-1 MUX Entity

The Interface to the 4-1 MUX is defined in the entity statement. The following code comprises the 4-1 MUX entity.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY 4_1_mux IS
   GENERIC(out_delay : time := 5 ns);
   PORT(  data1:    IN   std_logic;
          data2:    IN   std_logic;
          data3:    IN   std_logic;
          data4:    IN   std_logic;
          addr:     IN   std_logic_vector(1 DOWNTO 0);
          output:   OUT  std_logic);
END 4_1_mux;
```

Listing 32. 4-1 MUX Entity

5.1.2 Architecture

The actual behavior of the MUX is defined in the Architecture. The following code comprises the 4-1 MUX architecture.

```
ARCHITECTURE behav OF 4_1_mux IS
BEGIN
   mux_it:
   PROCESS(addr, data1, data2, data3, data4)
   BEGIN
     CASE addr IS
          WHEN "00" =>
               output <= data1 AFTER out_delay;
          WHEN "01" =>
               output <= data2 AFTER out_delay;
          WHEN "10" =>
               output <= data3 AFTER out_delay;
          WHEN "11" =>
               output <= data4 AFTER out_delay;
          WHEN OTHERS =>
               ASSERT FALSE
                    REPORT "addr out of range!"
                         SEVERITY ERROR;
   END PROCESS;
END behav;
```

Listing 33. 4-1 MUX Architecture

5.1.3 Configuration

The VHDL code which binds the ENTITY and ARCHITECTURE together is in the CONFIGURATION. The following code comprises the configuration of the 4-1 MUX.

```
CONFIGURATION config4 OF 4_1_mux IS
   FOR behav
   END FOR;
END config4;
```

Listing 34. 4-1 MUX Configuration

Since no components internal to the architecture have been declared, the configuration is empty. This is often referred to as a default configuration and may be omitted.

5.2 Behavioral 8-1 MUX

A behavioral 8-1 MUX can be created by using the two behavioral 4-1 MUX's. This is a good example of structured VHDL.

5.2.1 Entity

The following code comprises the entity of the 8-1 MUX.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY 8_1_mux IS
   GENERIC(out_delay : time := 5 ns);
   PORT(  data1:    IN   std_logic;
          data2:    IN   std_logic;
          data3:    IN   std_logic;
          data4:    IN   std_logic;
          data5:    IN   std_logic;
          data6:    IN   std_logic;
          data7:    IN   std_logic;
          data8:    IN   std_logic;
          addr:     IN   std_logic_vector(3 DOWNTO 0);
          output:   OUT  std_logic);
END 8_1_mux;
```

Listing 35. 8-1 Mux Entity

For this code, we define a delay for all signal assignments of 5ns. If we had written a flat behavioral model, we would use this delay for all

gates. In this case, however, we are using a structural model and we will "map" values to the individual components.

5.2.2 Architecture

The following code comprises the architecture of the 8-1 MUX.

```
ARCHITECTURE behav8 OF 8_1_mux IS
  ALIAS addr3: std_logic IS addr(3);
  ALIAS  address:  std_logic_vector(1  DOWNTO  0)  IS
  addr(1 DOWNTO 0);
  SIGNAL out1: std_logic;
  SIGNAL out2: std_logic;
  COMPONENT 4_1_mux
    GENERIC(out_delay : time := 5 ns);-- this defines
    PORT(     data1:    IN   std_logic;-- a default
              data2:    IN   std_logic;-- delay of 5ns
              data3:    IN   std_logic;-- if no MAP is
              data4:    IN   std_logic;-- used
              addr:     IN   std_logic_vector(1 DOWNTO
                             0);
              output:   OUT  std_logic);
  END COMPONENT;
BEGIN
  u1: 4_1_mux
    GENERIC MAP(3.5 ns) -- a little slower
    PORT MAP( data1 => data1,
              data2 => data2,
              data3 => data3,
              data4 => data4,
              addr => address,
              output => out1);
  u2: 4_1_mux
    GENERIC MAP(5.5 ns) -- a little faster
    PORT MAP( data1 => data5,
              data2 => data6,
              data3 => data7,
              data4 => data8,
              addr => address,
              output => out2);
  BEGIN               -- out2
    CASE addr3 IS
         WHEN '0' =>
              output <= out1;
         WHEN '1' =>
              output <= out2;
         WHEN OTHERS =>
              ASSERT FALSE
                   REPORT "address error"
                        SEVERITY ERROR;
    END CASE;
```

```
   END PROCESS;
END behav8;
```

Listing 36. 8-1 MUX Architecture

A GENERIC MAP is used to specify different delays for components than the delays specified in the components. This allows an easy way of changing parameters passed into a component. The example above uses generics and generic maps to provide timing information. A GENERIC MAP will always override any previous GENERIC clauses. A generic can be thought of as a default value, and a generic map overides the default.

5.2.3 Configuration

The following code comprises the configuration for the 8-1 MUX.

```
CONFIGURATION config OF 4_1_mux IS
   FOR behav8
     FOR u1: 4_1_mux
          USE CONFIGURATION config4;
     END FOR;
     FOR u2: 4_1_mux
          USE CONFIGURATION config4;
     END FOR;
   END FOR;
END config;
```

Listing 37. 8-1 MUX Configuration

5.3 VHDL Test benches

After the actual VHDL code is written, the important task of testing the code comes. Simulators usually offer a built in test capability, but often it is more desirable to actually construct a VHDL test process. Once the final behavioral or functional code is complete, we can instantiate the top-

level entity in a test bench and write a process which toggles the inputs and monitors the outputs. The test bench can be as simple as a state machine, or as complex as a process which reads inputs from a file and compares the outputs to a pre-generated file.

Testing can be done on a variety of levels. Simulators usually have their own built in simulation packages which consist of "forcing functions" to simulate your code. This has its limitations. For one, using VHDL it is possible to write complicated test benches comprised of multiple files or stimulus written in other languages and saved as a text file to be implemented using TEXTIO functions. These functions allow you to read and write text files.

5.3.1 8-1 MUX Test bench

In order to effectively test the MUX, we must write a test bench which will toggle the inputs. We can do this quite effectively with a VHDL test bench.

```
ENTITY testbench IS -- we must declare all signals to
                       look at
  PORT(  data1:    IN   std_logic;
         data2:    IN   std_logic;
         data3:    IN   std_logic;
         data4:    IN   std_logic;
         data5:    IN   std_logic;
         data6:    IN   std_logic;
         data7:    IN   std_logic;
         data8:    IN   std_logic;
         addr:     IN   std_logic_vector(3 DOWNTO 0);
         output:   OUT  std_logic);
END testbench;

ARCHITECTURE behavior OF testbench IS
  COMPONENT 8_1_mux
    GENERIC(out_delay : time := 5 ns);
    PORT(     data1:    IN   std_logic;
              data2:    IN   std_logic;
              data3:    IN   std_logic;
```

```
                data4:    IN   std_logic;
                data5:    IN   std_logic;
                data6:    IN   std_logic;
                data7:    IN   std_logic;
                data8:    IN   std_logic;
                addr:     IN   std_logic_vector(3 DOWNTO
                               0);
                output:   OUT  std_logic);
  END COMPONENT;
BEGIN
  u1: 8_1_mux
    PORT MAP( data1 => data1,
              data2 => data2,
              data3 => data3,
              data4 => data4,
              data5 => data5,
              data6 => data6,
              data7 => data7,
              data8 => data8,
              addr => addr,
              output => output);
  tester:
  PROCESS
    state : integer := 0;
  BEGIN
    WAIT UNTIL rising_edge(clock);
    CASE state IS
         WHEN 0 => -- initialize
              data1 <= '0';
              data2 <= '0';
              data3 <= '0';
              data4 <= '0';
              data5 <= '0';
              data6 <= '0';
              data7 <= '0';
              data8 <= '0';
              addr <= "000";
              state := 1;
         WHEN 1 => -- test addressing
              data1 <= '1';
              state := 2;
         WHEN 2 =>
              data1 <= '0';
              data2 <= '1';
              state := 3;
         WHEN 3 => -- test the second mux
              data2 <= '0';
              data5 <= '1';
              addr <= "100";
              state := 4;
         WHEN 4 =>
              data5 <= '0';
              state := 5;
         WHEN 5 => -- done
```

```
                ASSERT FALSE
                     REPORT "test complete"
                          SEVERITY note;
    END CASE;
  END PROCESS;
END behavior;

CONFIGURATION config OF testbench IS
  FOR behavior
    FOR u1: 8_1_mux
         USE CONFIGURATION config;
    END FOR;
  END FOR;
END config;
```

Listing 38. The complete 8-1 MUX VHDL

CHAPTER 6: AM2900 VHDL EXAMPLE

6.0 The AM2900 Bit- Sliced Pipelined Processor

The AM2900 micro controller is an interesting project to implement in VHDL. The processor itself poses somewhat of a challenge in VHDL, because VHDL is a clock based language, in which tasks are divided into clock cycles and code is written to implement them. The AM2900 completes one operation per clock cycle. This poses a problem in that all functions and procedures must execute immediately. This also limits us to using variables internal to the microprocessor, we can only use signals in the beginning and end of the process. Appendix A contains the entire listing of the AM2900 VHDL source code.

6.1 I/O

First thing that must be determined are the inputs and outputs to the system. For inputs, we will need:

data_in:	a 32 bit input to the AM2900
address:	The address in which to store the data_in
mem_load:	a toggle switch to determine whether to load the memory or execute the program
clock:	The clock input (12.5 Mhz is used)

For Outputs we will need:

y:	the 4 bit display on the AM2900
pc_out:	the program counter

For storage we'll need:

inst_mem: 16 32-bit words used to store the instructions

memory: 16 4-bit words used for register storage

pipe_reg: The 16 bit pipeline register

Internal Storage will require:

stack: The stack used for address storage

r, s, q: Internal registers

status: The status word from the calculation of f

The internal storage functions are comprised of mostly variables. These variables will be used for asynchronous operations, so they must be declared as variables. This has the disadvantage of not allowing us to display the outputs on the trace window. We can monitor variables during execution, but only signals can be displayed on the trace window.

6.2 Execution

Next we must determine the execution order for the AM2900. By looking at the inputs, we see that the AM2900 has two modes of operation, data entry and execution. We can therefore construct the outer case statement as:

```
CASE mem_load IS
WHEN '0' =>
-- execute code related to loading the memory
WHEN '1' =>
-- execute the instructions in memory
END CASE;
```

Listing 39. AM2900 Execution

6.2.1 Memory Load

The memory load by default resets the pc and sp. We do this so that the machine starts clean upon execution. Next, we assign the data_in to the location in the instruction memory pointed to by the supplied address. All of this is accomplished with the following code:

```
pc := 0;
sp := 0;
tempy := "0000";
inst_mem(address) <= data_in;
pipe_reg <= inst_mem(0);
pc_out <= pc;
```

Listing 40. AM2900 Memory Load

6.2.2 Execution

The other mode of operation is the micro program execution section. This section requires careful thought for the timing. First we must look at the instruction format. From this we see that there is a definite order in which the instruction is decoded. By looking at the instruction, we can determine the following:

1. The next micro-instruction control depends on the value of f and the status register (also depends on f)
2. The MUX control is only used for four of the Load operations
3. Load is based upon the ALU operation
4. The ALU is dependent on the source select.

Therefore, we will need to perform operations in the following order:

Source Select

ALU

Destination Control and MUX (where appropriate)

Next Instruction Control

6.2.2.1 Source Select

The code for the source select appears below:

```
CASE ss IS                          --  load registers
  WHEN "000" =>                     --  r <= a, s <= q
    int_address := bvtoi(a);
    r := memory(int_address);
    s := q;
  WHEN "001" =>                     --  r <= a, s <= b
    int_address := bvtoi(a);
    r := memory(int_address);
    int_address := bvtoi(b);
    s := memory(int_address);
  WHEN "010" =>                     --  r <= 0, s <= q
    r := "0000";
    s := q;
  WHEN "011" =>                     --  r <= 0, s <= b
    r := "0000";
    int_address := bvtoi(b);
    s := memory(int_address);
  WHEN "100" =>                     --  r <= 0, s <= a
    r := "0000";
    int_address := bvtoi(a);
    s := memory(int_address);
  WHEN "101" =>                     --  r <= d, s <= a
    r := d;
    int_address := bvtoi(a);
    s := memory(int_address);
  WHEN "110" =>                     --  r <= d, s <= q
    r := d;
    s := q;
  WHEN "111" =>                     --  r <= d, s <= 0;
    r := d;
    s := "0000";
  WHEN OTHERS =>
END CASE;                           --  ss
```

Listing 41. AM2900 Source Select

This section of code is pretty straight forward. We assign registers r and s the values pointed to by the registers or switches. We do call a function bvtoi which takes a bit vector as an input and returns an integer. We use this integer to look up the value in the memory. This code is included in a library supplied by Synopsis, reproduced in the appendices.

6.2.2.2 ALU

The ALU code determines the ALU operation to perform on the registers. The code appears below:

```
    CASE alu IS
      WHEN "000" =>                  --  f = r + s
        add_sub('0', r, s, cn, f, status);
      WHEN "001" =>                  --  f = s - r
        add_sub('1', s, r, cn, f, status);
      WHEN "010" =>                  --  f = r - s
        add_sub('1', r, s, cn, f, status);
      WHEN "011" =>                  --  f = r OR s
        f := r OR s;
      WHEN "100" =>                  --  f = r AND s
        f := r AND s;
      WHEN "101" =>                  --  f = (NOT r) AND s
        f := NOT(r) AND s;
      WHEN "110" =>                  --  f = r XOR s
        f := r XOR s;
      WHEN "111" =>                  --  f = NOT(r XOR s)
        f := NOT(r XOR s);
      WHEN OTHERS =>
    END CASE;                        --  alu
```

Listing 42. AM2900 ALU

Most of the operations can be performed with VHDL primitives. For the addition and subtraction, we define a procedure, add_sub:

```
  PROCEDURE add_sub(sub : IN bit;
                    r : IN bit_vector(3 DOWNTO 0);
                    s : IN bit_vector(3 DOWNTO 0);
                    cn : IN bit;
                    f : OUT bit_vector(3 DOWNTO 0);
                    status : OUT bit_vector(3 DOWNTO 0)
                    ) IS
    VARIABLE carry : bit_vector(3 DOWNTO 0) := "0000";
    VARIABLE int_sub : bit_vector(3 DOWNTO 0);
    VARIABLE final : bit_vector(3 DOWNTO 0);

    BEGIN

--------------------------------------------------------------
-------------------
--  if subtraction, take twos compliment
--------------------------------------------------------------
-------------------
      IF (sub = '1') THEN
```

```
        int_sub(0) := sub XOR s(0) XOR '1';
        carry(0) := sub AND s(0);
        int_sub(1) := sub XOR s(1) XOR carry(0);
        carry(1) := (sub AND s(1)) OR (carry(0) AND (sub XOR
s(1)));
        int_sub(2) := sub XOR s(2) XOR carry(1);
        carry(2) := (sub AND s(2)) OR (carry(1) AND (sub XOR
s(2)));
        int_sub(3) := sub XOR s(3) XOR carry(2);
      END IF;
------------------------------------------------------------
------------------
--  now add the two numbers
------------------------------------------------------------
------------------

      final(0) := int_sub(0) XOR r(0) XOR cn;
      carry(0) := (int_sub(0) AND r(0)) OR (cn AND
(int_sub(0) XOR r(0)));
      final(1) := int_sub(1) XOR r(1) XOR carry(0);
      carry(1) := (int_sub(1) AND r(1)) OR (carry(0) AND
(int_sub(1) XOR r(1)));
      final(2) := int_sub(2) XOR r(2) XOR carry(1);
      carry(2) := (int_sub(2) AND r(2)) OR (carry(1) AND
(int_sub(2) XOR r(2)));
      final(3) := int_sub(3) XOR r(3) XOR carry(2);
      carry(3) := (int_sub(3) AND r(3)) OR (carry(2) AND
(int_sub(3) XOR r(3)));
      IF (final = "0000") THEN
        status(2) := '0';
      END IF;
      status(0) := final(3);
      status(1) := carry(3);
      status(3) := carry(3) XOR carry(2);
      f := final;
    END add_sub;
```

Listing 43. AM2900 Add/Subtract Routine

The add_sub routine above takes in the two register, a bit which determines whether to add or subtract. It returns the result and sets the bits in the status register. Since it is only a four bit operation, no loops or recursion is used, and we explicitly spell out the operation.

6.2.2.3 Destination Control Word

The destination control word determines where the final value is stored and any final shifting operations are performed. The code appears below:

```
          mux(1) := mux1;                    --  form the mux
control word
          mux(0) := mux0;                    --  used in the
load command

------------------------------------------------------------
-------------------
--  decode the destination control word, y, the output, is
--  normally equal to f except for case "110"
------------------------------------------------------------
-------------------
          CASE dc IS
            WHEN "000" =>                    --  q <= f
              q := f;
              tempy := f;
            WHEN "001" =>                    --  NOP
              tempy := f;
            WHEN "010" =>                    --  memory(b) <= f
                                             --  y=a?
              tempy := a;
              int_address := bvtoi(b);
              memory(int_address) <= f;
            WHEN "011" =>                    --  memory(b) <= f
              tempy := f;
              int_address := bvtoi(b);
              memory(int_address) <= f;
            WHEN "100" =>                    --  b <= f/2,
                                             -- q <= q/2
              tempy := f;
              CASE mux IS
                WHEN "00" =>                 --  zero
                  f(3) := f(2);
                  f(2) := f(1);
                  f(1) := f(0);
                  f(0) := '0';
                  q(3) := q(2);
                  q(2) := q(1);
                  q(1) := q(0);
                  q(0) := '0';
                WHEN "01" =>                 --  rotate
                  temp := f(0);
                  f(0) := f(1);
                  f(1) := f(2);
                  f(2) := f(3);
                  f(3) := temp;
                  temp := q(0);
```

```
      q(0) := q(1);
      q(1) := q(2);
      q(2) := q(3);
      q(3) := temp;
    WHEN "10" =>              --  rotate double
      temp := q(3);
      q(3) := f(0);
      f(0) := f(1);
      f(1) := f(2);
      f(2) := f(3);
      f(3) := q(0);
      q(0) := q(1);
      q(1) := q(2);
      q(2) := temp;
    WHEN "11" =>              --  arithmetic
                              -- double
      q(0) := q(1);
      q(1) := q(2);
      q(2) := q(3);
      q(3) := f(0);
      f(0) := f(1);
      f(1) := f(2);
      f(2) := f(3);
  END CASE;                   --  mux
  int_address := bvtoi(b);
  memory(int_address) <= f;
WHEN "101" =>                 --  b <= f/2
  tempy := f;
  CASE mux IS
    WHEN "00" =>              --  zero
      f(3) := f(2);
      f(2) := f(1);
      f(1) := f(0);
      f(0) := '0';
    WHEN "01" =>              --  rotate
      temp := f(0);
      f(0) := f(1);
      f(1) := f(2);
      f(2) := f(3);
      f(3) := temp;
    WHEN "10" =>              --  rotate double
      f(0) := f(1);
      f(1) := f(2);
      f(2) := f(3);
      f(3) := q(0);
    WHEN "11" =>              --  arithmetic
                              --  double
      f(0) := f(1);
      f(1) := f(2);
      f(2) := f(3);
  END CASE;                   --  mux
WHEN "110" =>                 --  b <= 2f,
                              -- q <= 2q
  tempy := f;
```

```
  CASE mux IS
    WHEN "00" =>                -- zero
      f(3) := f(2);
      f(2) := f(1);
      f(1) := f(0);
      f(0) := '0';
      q(3) := q(2);
      q(2) := q(1);
      q(1) := q(0);
      q(0) := '0';
    WHEN "01" =>                -- rotate
      temp := f(3);
      f(3) := f(2);
      f(2) := f(1);
      f(1) := f(0);
      f(0) := temp;
      temp := q(3);
      q(3) := q(2);
      q(2) := q(1);
      q(1) := q(0);
      q(0) := temp;
    WHEN "10" =>                -- rotate double
      temp := f(0);
      f(0) := q(3);
      q(3) := q(2);
      q(2) := q(1);
      q(1) := q(0);
      q(0) := f(3);
      f(3) := f(2);
      f(2) := f(1);
      f(1) := temp;
    WHEN "11" =>                -- arithmetic
                                -- double
      f(3) := f(2);
      f(2) := f(1);
      f(1) := f(0);
      f(0) := q(3);
      q(3) := q(2);
      q(2) := q(1);
      q(1) := q(0);
      q(0) := '0';
  END CASE;                     -- mux
  int_address := bvtoi(b);
  memory(int_address) <= f;
WHEN "111" =>                   -- b <= 2f
  tempy := f;
  CASE mux IS
    WHEN "00" =>                -- zero
      f(3) := f(2);
      f(2) := f(1);
      f(1) := f(0);
      f(0) := '0';
    WHEN "01" =>                -- rotate
      temp := f(3);
```

```
              f(3) := f(2);
              f(2) := f(1);
              f(1) := f(0);
              f(0) := temp;
            WHEN "10" =>              --  rotate double
              f(3) := f(2);
              f(2) := f(1);
              f(1) := f(0);
              f(0) := q(3);
            WHEN "11" =>              --  arithmetic
                                      --  double
              f(3) := f(2);
              f(2) := f(1);
              f(1) := f(0);
              f(0) := q(3);
          END CASE;                   --  mux
          int_address := bvtoi(b);
          memory(int_address) <= f;
      END CASE;                       --  dc
      y <= tempy;
```

Listing 44. AM2900 Destination Control Word

6.2.2.4 The next instruction control

The final operation is the next instruction control. This operation controls the execution flow of the AM2900. It relies on all the previous operations performed. The code appears below:

```
------------------------------------------------------------
------------------
--  decode instruction and execute
------------------------------------------------------------
------------------
      CASE ni IS
        WHEN "0000" =>                --  branch if F != 0
          IF (status(2) = '0') THEN
            pc := bvtoi(ba);
          ELSE
            pc := pc + 1;
          END IF;
        WHEN "0001" =>                --  branch
          pc := bvtoi(ba);
        WHEN "0010" =>                --  continue
          pc := pc + 1;
        WHEN "0011" =>                --  branch map
          pc := bvtoi(d);
        WHEN "0100" =>                --  JSR if f != 0
          IF ((status(2) = '0')) THEN
            IF (sp > 15) THEN
```

```
      ASSERT FALSE
        REPORT "ERROR"
          SEVERITY ERROR;
    END IF;
    stack(sp) := pc;
    sp := sp + 1;
    pc := bvtoi(ba);
  ELSE
    pc := pc + 1;
  END IF;
WHEN "0101" =>                -- JSR
  IF (sp > 15) THEN
    ASSERT FALSE
      REPORT "error stack overflow"
        SEVERITY ERROR;
  END IF;
  stack(sp) := pc;
  sp := sp + 1;
  pc := bvtoi(ba);
WHEN "0110" =>                -- RTS
  IF (sp = 0) THEN
    ASSERT FALSE
      REPORT "error, stack underflow"
        SEVERITY ERROR;
  END IF;
  pc := stack(sp);
  sp := sp - 1;
WHEN "0111" =>                -- file reference
                              -- the stack
  pc := stack(sp);
WHEN "1000" =>                -- end loop and
                              -- pop if f = 0
  IF (status(2) = '0') THEN
    IF (sp = 0) THEN
      ASSERT FALSE
        REPORT "error stack underflow"
          SEVERITY ERROR;
    END IF;
    pc := stack(sp);
    sp := sp + 1;
  ELSE
    pc := pc + 1;
  END IF;
WHEN "1001" =>                -- push and
                              -- continue
  IF (sp > 15) THEN
    ASSERT FALSE
      REPORT "error stack overflow"
        SEVERITY ERROR;
  END IF;
  stack(sp) := bvtoi(ba);
  sp := sp + 1;
  pc := pc + 1;
WHEN "1010" =>                -- pop and
```

```
                                -- continue
    IF (sp = 0) THEN
      ASSERT FALSE
        REPORT "error stack underflow"
          SEVERITY ERROR;
    END IF;
    sp := sp - 1;
    pc := pc + 1;
  WHEN "1011" =>                 --  end loop and
                                -- pop if cout
    IF (status(1) = '1') THEN
      IF (sp = 0) THEN
        ASSERT FALSE
          REPORT "error stack underflow"
            SEVERITY ERROR;
      END IF;
      pc := stack(sp);
      sp := sp - 1;
    ELSE
      pc := pc + 1;
    END IF;
  WHEN "1100" =>             --  branch reg if f = 0
    IF (status(2) = '1') THEN
      pc := bvtoi(ba);
    ELSE
      pc := pc + 1;
    END IF;
  WHEN "1101" =>               --  branch reg if f3
    IF (status(0) = '1') THEN
      pc := bvtoi(ba);
    ELSE
      pc := pc + 1;
    END IF;
  WHEN "1110" =>              --  branch reg if ovr
    IF (status(3) = '1') THEN
      pc := bvtoi(ba);
    ELSE
      pc := pc + 1;
    END IF;
  WHEN "1111" =>              --  branch reg if cout
    IF (status(1) = '1') THEN
      pc := bvtoi(ba);
    ELSE
      pc := pc + 1;
    END IF;
END CASE;                       --  ni
pipe_reg <= inst_mem(pc);
pc_out <= pc;
```

Listing 45. AM2900 Next Instruction Control

CHAPTER 7: TESTING THE AM2900

7.0 The AM2900 Test benches

The AM2900 VHDL description has been written, but now we must test the execution to make sure that it performs properly. To ensure that the execution performs properly, we must run some VHDL test benches. The test benches which I present here contain two parts.

The first part is a simple clock generator, and the second is the actual test code. The test code is further broken up into multiple data load steps and two steps to perform an infinite execution. The simulation run length depends on the simulator settings. I ran the test benches for 2 μSeconds, but a longer or shorter time can be picked.

The actual code for the test benches can be found in the appendices, but a short description of each follows.

7.1 Load the AM2900 Ram

This test bench corresponds to loading three of the registers with values selected by the D switches. The listing appears in Appendix B along with the simulation results.

```
------------------------------------------------------------
------------------
--  AM2900 Testbench #1: Loading the AM2901 RAM
--  corresponds to Exercise #1 from professor Chang
--  instruction memory is as follows:
--  0: XXXX XXXX X011 X111 X011 XXXX 0000 0010
--  1: XXXX XXXX X001 X011 X011 XXXX 0000 XXXX
--  2: XXXX XXXX X011 X111 X011 XXXX 0001 0100
--  3: XXXX XXXX X001 X011 X011 XXXX 0001 XXXX
--  4: XXXX XXXX X011 X111 X011 XXXX 1001 0101
--  5: XXXX XXXX X001 X011 X011 XXXX 1001 XXXX
--  6: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  7: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  8: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
```

```
--  9: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  10:XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  11:XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  12:XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  13:XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  14:XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  15:XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
------------------------------------------------------------
------------------

library work;
USE work.am2900_pkg.ALL;

ENTITY am2900_test1 IS
END am2900_test1;

ARCHITECTURE tester1 OF am2900_test1 IS
COMPONENT am2900
  PORT( inst_mem     : BUFFER bit_16_array(15 DOWNTO 0);
        data_in      : IN     bit_vector(31 DOWNTO 0);
        address      : IN     integer;
        mem_load     : IN     bit;
        clock        : IN     bit;
        run_load     : IN     bit;
        y            : OUT    bit_vector(3 DOWNTO 0);
        memory       : BUFFER bit_4_array(15 DOWNTO 0);
        pipe_reg     : BUFFER bit_vector(31 DOWNTO 0);
        pc_out       : OUT integer);
END COMPONENT;

  SIGNAL inst_mem : bit_16_array(15 DOWNTO 0);
  SIGNAL data_in : bit_vector(31 DOWNTO 0);
  SIGNAL address : integer;
  SIGNAL mem_load : bit;
  SIGNAL clock : bit;
  SIGNAL run_load : bit;
  SIGNAL y : bit_vector(3 DOWNTO 0);
  SIGNAL memory : bit_4_array(15 DOWNTO 0);
  SIGNAL pipe_reg : bit_vector(31 DOWNTO 0);
  SIGNAL pc_out : integer;

BEGIN  --  tester
  u1: am2900
    PORT MAP( inst_mem => inst_mem,
              data_in => data_in,
              address => address,
              mem_load => mem_load,
              clock => clock,
              run_load => run_load,
              y => y,
              memory => memory,
              pipe_reg => pipe_reg,
              pc_out => pc_out);
```

```
-------------------------------------------------------------
------------------
--  We will use a 15.5Mhz clock to drive the circuit, by
using a clock
--  circuit, we give up the single stepping capability, but
since we
--  will get out a sample over time, this is not a problem.
-------------------------------------------------------------
------------------
clock_gen_12_5:
PROCESS
BEGIN

        clock <= '0';
        WAIT FOR 40 ns;
        clock <= '1';
        WAIT FOR 40 ns;
END PROCESS;

tester:
PROCESS
  VARIABLE state : integer := 0;
  BEGIN
    WAIT UNTIL (clock'event AND clock ='1');
    CASE state IS

-------------------------------------------------------------
------------------
--  load the memory with the above program specified in the
--  initial comment section
-------------------------------------------------------------
------------------
      WHEN 0 =>
        mem_load <= '0';
        address <= 0;
        data_in <= "00000010001101110011000000000010";
        state := 1;
      WHEN 1 =>
        address <= 1;
        data_in <= "00000010000100110011000000000000";
        state := 2;
      WHEN 2 =>
        address <= 2;
        data_in <= "00000010001101110011000000010100";
        state := 3;
      WHEN 3 =>
        address <= 3;
        data_in <= "00000010000100110011000000010000";
        state := 4;
      WHEN 4 =>
        address <= 4;
        data_in <= "00000010001101110011000010010101";
        state := 5;
      WHEN 5 =>
```

```
        address <= 5;
        data_in <= "00000010000100110011000010010000";
        state := 6;
      WHEN 6 =>
        mem_load <= '1';
        state := 7;
      WHEN OTHERS =>
    END CASE;                              --  state
  END PROCESS;
END tester1;

CONFIGURATION config1 OF am2900_test1 IS
  FOR tester1
    FOR u1: am2900
      USE CONFIGURATION work.am2900_con;
    END FOR;
  END FOR;
END config1;
```

Listing 46. AM2900 Test bench #1

7.2 Branch And Continue

This program corresponds to loading a program into memory which contains multiple branches and continue instructions. The listing appears in Appendix C along with the simulation results.

```
------------------------------------------------------------
-------------------
--  AM2900 Testbench #4: Continue and Branch
--  corresponds to Exercise #4 from professor Chang
--  instruction memory is as follows:
--  0: 1001 0001 XXXX XXXX XXXX XXXX XXXX XXXX
--  1: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  2: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  3: 0000 0001 XXXX XXXX XXXX XXXX XXXX XXXX
--  4: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  5: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  6: 1010 0001 XXXX XXXX XXXX XXXX XXXX XXXX
--  7: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  8: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
--  9: 0110 0001 XXXX XXXX XXXX XXXX XXXX XXXX
--  10:XXXX 0010 XXXX XXXX XXXX XXXX XXXX XXXX
--  11:XXXX 0010 XXXX XXXX XXXX XXXX XXXX XXXX
--  12:XXXX 0010 XXXX XXXX XXXX XXXX XXXX XXXX
--  13:XXXX 0010 XXXX XXXX XXXX XXXX XXXX XXXX
--  14:XXXX 0010 XXXX XXXX XXXX XXXX XXXX XXXX
--  15:0011 0001 XXXX XXXX XXXX XXXX XXXX XXXX
```

```
------------------------------------------------------------
------------------

library work;
USE work.am2900_pkg.ALL;

ENTITY am2900_test IS
END am2900_test;

ARCHITECTURE tester OF am2900_test IS
COMPONENT am2900
  PORT( inst_mem     : BUFFER bit_16_array(15 DOWNTO 0);
        data_in      : IN     bit_vector(31 DOWNTO 0);
        address      : IN     integer;
        mem_load     : IN     bit;
        clock        : IN     bit;
        run_load     : IN     bit;
        y            : OUT    bit_vector(3 DOWNTO 0);
        memory       : BUFFER bit_4_array(15 DOWNTO 0);
        pipe_reg     : BUFFER bit_vector(31 DOWNTO 0);
        pc_out       : OUT integer);
END COMPONENT;

  SIGNAL inst_mem : bit_16_array(15 DOWNTO 0);
  SIGNAL data_in : bit_vector(31 DOWNTO 0);
  SIGNAL address : integer;
  SIGNAL mem_load : bit;
  SIGNAL clock : bit;
  SIGNAL run_load : bit;
  SIGNAL y : bit_vector(3 DOWNTO 0);
  SIGNAL memory : bit_4_array(15 DOWNTO 0);
  SIGNAL pipe_reg : bit_vector(31 DOWNTO 0);
  SIGNAL pc_out : integer;

BEGIN  --  tester
  u1: am2900
    PORT MAP( inst_mem => inst_mem,
              data_in => data_in,
              address => address,
              mem_load => mem_load,
              clock => clock,
              run_load => run_load,
              y => y,
              memory => memory,
              pipe_reg => pipe_reg,
              pc_out => pc_out);

------------------------------------------------------------
------------------
--  We will use a 15.5Mhz clock to drive the circuit, by
using a clock
--  circuit, we give up the single stepping capability, but
since we
--  will get out a sample over time, this is not a problem.
```

```
------------------------------------------------------------
------------------
clock_gen_12_5:
PROCESS
BEGIN

        clock <= '0';
        WAIT FOR 40 ns;
        clock <= '1';
        WAIT FOR 40 ns;
END PROCESS;

tester:
PROCESS
  VARIABLE state : integer := 0;
  BEGIN
    WAIT UNTIL (clock'event AND clock ='1');
    CASE state IS

------------------------------------------------------------
------------------
--  load the memory with the above program specified in the
--  initial comment section
------------------------------------------------------------
------------------
      WHEN 0 =>
        mem_load <= '0';
        address <= 0;
        data_in <= "10010001000000000000000000000000";
        state := 1;
      WHEN 1 =>
        address <= 3;
        data_in <= "00000001000000000000000000000000";
        state := 2;
      WHEN 2 =>
        address <= 6;
        data_in <= "10100001000000000000000000000000";
        state := 3;
      WHEN 3 =>
        address <= 9;
        data_in <= "01100001000000000000000000000000";
        state := 4;
      WHEN 4 =>
        address <= 10;
        data_in <= "00000010000000000000000000000000";
        state := 5;
      WHEN 5 =>
        address <= 11;
        data_in <= "00000010000000000000000000000000";
        state := 6;
      WHEN 6 =>
        address <= 12;
        data_in <= "00000010000000000000000000000000";
        state := 7;
```

```
      WHEN 7 =>
        address <= 13;
        data_in <= "00000010000000000000000000000000";
        state := 8;
      WHEN 8 =>
        address <= 14;
        data_in <= "00000010000000000000000000000000";
        state := 9;
      WHEN 9 =>
        address <= 15;
        data_in <= "00110001000000000000000000000000";
        state := 10;
      WHEN 10 =>
        mem_load <= '1';
        state := 11;
      WHEN OTHERS =>
    END CASE;                              --  state
  END PROCESS;
END tester;

CONFIGURATION config OF am2900_test IS
  FOR tester
    FOR u1: am2900
      USE CONFIGURATION work.am2900_con;
    END FOR;
  END FOR;
END config;
```

Listing 47. AM2900 Branch and Continue

CHAPTER 8: AM2900 SYNTHESIS

8.0 Coding the AM2900 for Synthesis

Appendix E contains the actual code which was put through the Synopsys Design Architect to produce the VHDL output found on the accompanying disk. The output code is not included in printed form, as it is 228 pages long of structural VHDL. The main differences between this code and the previous AM2900 code, is that it is optimized for synthesis. This is very important to keep gate counts down and duplicity to a minimum. The major changes are:

8.1 Multiple Processes

This is accomplished by splitting VHDL into logical units, processes, which are either clock driven or event driven. All event driven processes are combinatorial and all clock driven processes are sequential.

8.2 Reduced Duplicity

We must reduce all blocks of code which will be synthesized into multiple functional blocks. This includes the add_sub routine, as well as the arithmetic operations.

8.3 Variable Management

Variables are a powerful part of VHDL, however it is possible to create segments of code which simulate properly, but have no meaning in hardware. An example would be using

```
WHEN "01" =>                -- rotate
  temp := f(3);
  f(3) := f(2);
  f(2) := f(1);
  f(1) := f(0);
  f(0) := temp;
  temp := q(3);
  q(3) := q(2);
  q(2) := q(1);
  q(1) := q(0);
  q(0) := temp;
```

In this code segment, the variable *temp* is used twice in a row. In hardware, this is impossible. A signal wire **can not** have two different values simultaneously. Synthesis tools may be able to correct such a problem, however it is bad coding practice to do this.

8.4 The Result

The Result can be seen in appendix G, and is functionally equivalent to the AM2900 behavioral in all respects. The test benches and results are not duplicated, however all simulation results were the same.

CHAPTER 9 CONCLUSIONS

VHDL programming is a rapidly growing field in which there is a high demand. VHDL is also an evolving environment. VHDL '92 will be finalized soon, expanding upon the original VHDL definition and correcting problems in the language. It is a standard in military hardware, but it is fast becoming a standard in commercial hardware also. The purpose of this paper was to introduce Tufts students to VHDL. This paper should provide a basis to understand and write simple VHDL. With further study, students should be able to handle complex tasks using VHDL.

REFERENCES

[1] IEEE 1076 Language Reference Manual (LRM)
IEEE 201-981-1391; 1988

[2] The VHDL Handbook
Dave Coelho
Kluwer Academic Publishers; 1989

[3] VHDL: Hardware Description and Design
Carey Ussery
Kluwer Academic Publishers; 1989

[4] VHDL
Douglas Perry
McGraw-Hill; 1990

[5] Chip Level Modeling Using VHDL
Dr. James Armstrong, Virginia Polytechnical Institute
Prentiss-Hall; 1988

[6] Applications of VHDL to Circuit Design
Randolph E. Harr, Alec G. Stanculescu
Kluwer Academic Publishers; 1991

[7] AM2900 Reference Materials
American Micro Devices