The VHDL Cookbook

First Edition

Peter J. Ashenden

The VHDL Cookbook

First Edition

July, 1990

Peter J. Ashenden

Dept. Computer Science
University of Adelaide
South Australia

Contents

Contents (cont'd)

Contents (cont'd)

1. Introduction

VHDL is a language for describing digital electronic systems. It arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US.

VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

The purpose of this booklet is to give you a quick introduction to VHDL. This is done by informally describing the facilities provided by the language, and using examples to illustrate them. This booklet does not fully describe every aspect of the language. For such fine details, you should consult the *IEEE Standard VHDL Language Reference Manual*. However, be warned: the standard is like a legal document, and is very difficult to read unless you are already familiar with the language. This booklet does cover enough of the language for substantial model writing. It assumes you know how to write computer programs using a conventional programming language such as Pascal, C or Ada.

The remaining chapters of this booklet describe the various aspects of VHDL in a bottom-up manner. Chapter2 describes the facilities of VHDL which most resemble normal sequential programming languages. These include data types, variables, expressions, sequential statements and subprograms. Chapter3 then examines the facilities for describing the structure of a module and how it it decomposed into sub-modules. Chapter4 covers aspects of VHDL that integrate the programming language features with a discrete event timing model to allow simulation of behaviour. Chapter5 is a key chapter that shows how all these facilities are combined to form a complete model of a system. Then Chapter6 is a pot-pourri of more advanced features which you may find useful for modeling more complex systems.

Throughout this booklet, the syntax of language features is presented in Backus-Naur Form (BNF). The syntax specifications are drawn from the IEEE VHDL Standard. Concrete examples are also given to illustrate the language features. In some cases, some alternatives are omitted from BNF

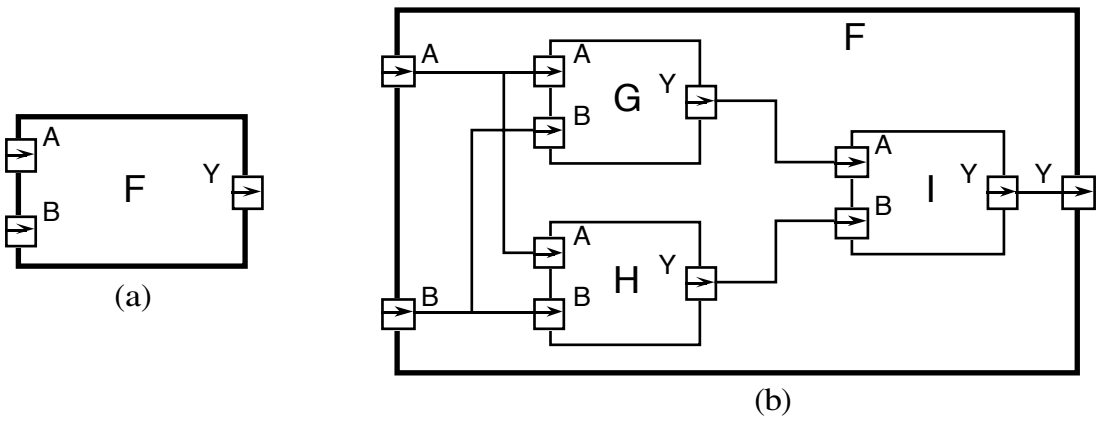


Figure 1-1. Example of a structural description.

productions where they are not directly relevant to the context. For this reason, the full syntax is included in AppendixA, and should be consulted as a reference.

1.1. Describing Structure

A digital electronic system can be described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the inputs. Figure1-1(a) shows an example of this view of a digital system. The module F has two inputs, A and B, and an output Y. Using VHDL terminology, we call the module F a design *entity*, and the inputs and outputs are called *ports*.

One way of describing the function of a module is to describe how it is composed of sub-modules. Each of the sub-modules is an *instance* of some entity, and the ports of the instances are connected using *signals*. Figure1-1(b) shows how the entity F might be composed of instances of entities G, H and I. This kind of description is called a *structural* description. Note that each of the entities G, H and I might also have a structural description.

1.2. Describing Behaviour

In many cases, it is not appropriate to describe a module structurally. One such case is a module which is at the bottom of the hierarchy of some other structural description. For example, if you are designing a system using IC packages bought from an IC shop, you do not need to describe the internal structure of an IC. In such cases, a description of the function performed by the module is required, without reference to its actual internal structure. Such a description is called a *functional* or *behavioural* description.

To illustrate this, suppose that the function of the entity F in Figure1-1(a) is the exclusive-or function. Then a behavioural description of F could be the Boolean function

$$Y = \overline{A} \,.\, B + A \,.\, \overline{B}$$

More complex behaviours cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. VHDL solves this problem by allowing description of behaviour in the form

of an executable program. Chapters2 and4 describe the programming language facilities.

1.3. Discrete Event Time Model

Once the structure and behaviour of a module have been specified, it is possible to simulate the module by executing its bevioural description. This is done by simulating the passage of time in discrete steps. At some simulation time, a module input may be stimulated by changing the value on an input port. The module reacts by running the code of its behavioural description and scheduling new values to be placed on the signals connected to its output ports at some later simulated time. This is called scheduling a *transaction* on that signal. If the new value is different from the previous value on the signal, an *event* occurs, and other modules with input ports connected to the signal may be activated.

The simulation starts with an *initialisation phase*, and then proceeds by repeating a two-stage *simulation cycle*. In the initialisation phase, all signals are given initial values, the simulation time is set to zero, and each module's behaviour program is executed. This usually results in transactions being scheduled on output signals for some later time.

In the first stage of a simulation cycle, the simulated time is advanced to the earliest time at which a transaction has been scheduled. All transactions scheduled for that time are executed, and this may cause events to occur on some signals.

In the second stage, all modules which react to events occurring in the first stage have their behaviour program executed. These programs will usually schedule further transactions on their output signals. When all of the behaviour programs have finished executing, the simulation cycle repeats. If there are no more scheduled transactions, the whole simulation is completed.

The purpose of the simulation is to gather information about the changes in system state over time. This can be done by running the simulation under the control of a *simulation monitor*. The monitor allows signals and other state information to be viewed or stored in a trace file for later analysis. It may also allow interactive stepping of the simulation process, much like an interactive program debugger.

1.4. A Quick Example

In this section we will look at a small example of a VHDL description of a two-bit counter to give you a feel for the language and how it is used. We start the description of an entity by specifying its external interface, which includes a description of its ports. So the counter might be defined as:

```
entity count2 is
    generic (prop_delay : Time := 10 ns);
    port (clock : in bit;
          q1, q0 : out bit);
end count2;
```

This specifies that the entity count2 has one input and two outputs, all of which are bit values, that is, they can take on the values '0' or '1'. It also defines a generic constant called prop_delay which can be used to control the operation of the entity (in this case its propagation delay). If no value is

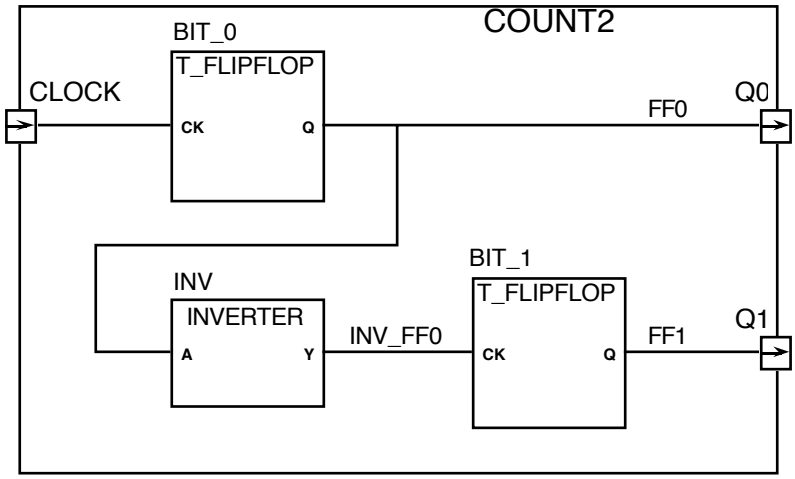


Figure1-2. Structure of count2.

explicitly given for this value when the entity is used in a design, the default value of 10ns will be used.

An implementation of the entity is described in an architecture body. There may be more than one architecture body corresponding to a single entity specification, each of which describes a different view of the entity. For example, a behavioural description of the counter could be written as:

```
architecture behaviour 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 behaviour;
```

In this description of the counter, the behaviour is implemented by a process called count_up, which is sensitive to the input clock. A process is a body of code which is executed whenever any of the signals it is sensitive to changes value. This process has a variable called count_value to store the current state of the counter. The variable is initialized to zero at the start of simulation, and retains its value between activations of the process. When the clock input changes from '0' to '1', the state variable is incremented, and transactions are scheduled on the two output ports based on the new value. The assignments use the generic constant prop_delay to determine how long after the clock change the transaction should be scheduled. When control reaches the end of the process body, the process is suspended until another change occurs on clock.

The two-bit counter might also be described as a circuit composed of two T-flip-flops and an inverter, as shown in Figure1-2. This can be written in VHDL as:

```
architecture structure of count2 is

    component t_flipflop
        port (ck : in bit;  q : out bit);
    end  component;

    component inverter
        port (a : in bit;  y : out bit);
    end  component;

    signal ff0, ff1, inv_ff0 : bit;

begin

    bit_0 : t_flipflop port map (ck => clock, q => ff0);

    inv : inverter port map (a => ff0, y => inv_ff0);

    bit_1 : t_flipflop port map (ck => inv_ff0, q => ff1);

    q0 <= ff0;
    q1 <= ff1;

end  structure;
```

In this architecture, two component types are declared, t_flipflop and inverter, and three internal signals are declared. Each of the components is then instantiated, and the ports of the instances are mapped onto signals and ports of the entity. For example, bit_0 is an instance of the t_flipflop component, with its ck port connected to the clock port of the count2 entity, and its q port connected to the internal signal ff0. The last two signal assignments update the entity ports whenever the values on the internal signals change.

2. VHDL is Like a Programming Language

As mentioned in Section 1.2, the behaviour of a module may be described in programming language form. This chapter describes the facilities in VHDL which are drawn from the familiar programming language repertoire. If you are familiar with the Ada programming language, you will notice the similarity with that language. This is both a convenience and a nuisance. The convenience is that you don't have much to learn to use these VHDL facilities. The problem is that the facilities are not as comprehensive as those of Ada, though they are certainly adequate for most modeling purposes.

2.1. Lexical Elements

2.1.1. Comments

Comments in VHDL start with two adjacent hyphens ('--') and extend to the end of the line. They have no part in the meaning of a VHDL description.

2.1.2. Identifiers

Identifiers in VHDL are used as reserved words and as programmer defined names. They must conform to the rule:

```
identifier ::= letter { [ underline ] letter_or_digit }
```

Note that case of letters is not considered significant, so the identifiers cat and Cat are the same. Underline characters in identifiers are significant, so This_Name and ThisName are different identifiers.

2.1.3. Numbers

Literal numbers may be expressed either in decimal or in a base between two and sixteen. If the literal includes a point, it represents a real number, otherwise it represents an integer. Decimal literals are defined by:

```
decimal_literal ::= integer [ . integer ] [ exponent ]
integer ::= digit { [ underline ] digit }
exponent ::= E [ + ] integer | E - integer
```

Some examples are:

```
0       1       123_456_789       987E6       -- integer literals
0.0     0.5     2.718_28          12.4E-9     -- real literals
```

Based literal numbers are defined by:

```
based_literal ::= base # based_integer [ . based_integer ] # [ exponent ]
base ::= integer
based_integer ::= extended_digit { [ underline ] extended_digit }
```

```
extended_digit ::= digit | letter
```

The base and the exponent are expressed in decimal. The exponent indicates the power of the base by which the literal is multiplied. The letters A to F (upper or lower case) are used as extended digits to represent 10 to 15. Some examples:

```
2#1100_0100#     16#C4#     4#301#E1       -- the integer 196
2#1.1111_1111_111#E+11     16#F.FF#E2     -- the real number 4095.0
```

2.1.4. Characters

Literal characters are formed by enclosing an ASCII character in single-quote marks. For example:

```
'A'      '*'      '"'      ' '
```

2.1.5. Strings

Literal strings of characters are formed by enclosing the characters in double-quote marks. To include a double-quote mark itself in a string, a pair of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters. Examples of strings:

```
"A string"
""                                          -- empty string
"A string in a string: ""A string"". "      -- contains quote marks
```

2.1.6. Bit Strings

VHDL provides a convenient way of specifying literal values for arrays of type bit ('0's and '1's, see Section 2.2.5). The syntax is:

```
bit_string_literal ::= base_specifier " bit_value "
base_specifier ::= B | O | X
bit_value ::= extended_digit { [ underline ] extended_digit }
```

Base specifier B stands for binary, O for octal and X for hexadecimal. Some examples:

```
B"1010110"    -- length is 7
O"126"        -- length is 9, equivalent to B"001_010_110"
X"56"         -- length is 8, equivalent to B"0101_0110"
```

2.2. Data Types and Objects

VHDL provides a number of basic, or *scalar*, types, and a means of forming *composite* types. The scalar types include numbers, physical quantities, and enumerations (including enumerations of characters), and there are a number of standard predefined basic types. The composite types provided are arrays and records. VHDL also provides *access* types (pointers) and *files*, although these will not be fully described in this booklet.

A data type can be defined by a type declaration:

```
full_type_declaration ::= type identifier is type_definition ;
type_definition ::=
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
scalar_type_definition ::=
    enumeration_type_definition | integer_type_definition
    | floating_type_definition | physical_type_definition
```

```
composite_type_definition ::=
    array_type_definition
    | record_type_definition
```

Examples of different kinds of type declarations are given in the following sections.

2.2.1. Integer Types

An integer type is a range of integer values within a specified range. The syntax for specifying integer types is:

```
integer_type_definition ::= range_constraint
range_constraint ::= range range
range ::= simple_expression direction simple_expression
direction ::= to | downto
```

The expressions that specify the range must of course evaluate to integer numbers. Types declared with the keyword **to** are called *ascending* ranges, and those declared with the keyword **downto** are called *descending* ranges. The VHDL standard allows an implementation to restrict the range, but requires that it must at least allow the range –2147483647 to +2147483647.

Some examples of integer type declarations:

```
type byte_int is range 0 to 255;
type signed_word_int is range –32768 to 32767;
type bit_index is range 31 downto 0;
```

There is a predefined integer type called integer. The range of this type is implementation defined, though it is guaranteed to include –2147483647 to +2147483647.

2.2.2. Physical Types

A physical type is a numeric type for representing some physical quantity, such as mass, length, time or voltage. The declaration of a physical type includes the specification of a base unit, and possibly a number of secondary units, being multiples of the base unit. The syntax for declaring physical types is:

```
physical_type_definition ::=
    range_constraint
        units
            base_unit_declaration
            { secondary_unit_declaration }
        end units
base_unit_declaration ::= identifier ;
secondary_unit_declaration ::= identifier = physical_literal ;
physical_literal ::= [ abstract_literal ] unit_name
```

Some examples of physical type declarations:

```
type length is range 0 to 1E9
    units
        um;
        mm = 1000 um;
        cm = 10 mm;
        m = 1000 mm;
        in = 25.4 mm;
        ft = 12 in;
        yd = 3 ft;
        rod = 198 in;
        chain = 22 yd;
        furlong = 10 chain;
    end units;

type resistance is range 0 to 1E8
    units
        ohms;
        kohms = 1000 ohms;
        Mohms = 1E6 ohms;
    end units;
```

The predefined physical type time is important in VHDL, as it is used extensively to specify delays in simulations. Its definition is:

```
type time is range implementation_defined
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
    end units;
```

To write a value of some physical type, you write the number followed by the unit. For example:

```
10 mm   1 rod   1200 ohm   23 ns
```

2.2.3. Floating Point Types

A floating point type is a discrete approximation to the set of real numbers in a specified range. The precision of the approximation is not defined by the VHDL language standard, but must be at least six decimal digits. The range must include at least –1E38 to +1E38. A floating point type is declared using the syntax:

```
floating_type_definition := range_constraint
```

Some examples are:

```
type signal_level is range –10.00 to +10.00;

type probability is range 0.0 to 1.0;
```

There is a predefined floating point type called real. The range of this type is implementation defined, though it is guaranteed to include –1E38 to +1E38.

2.2.4. Enumeration Types

An enumeration type is an ordered set of identifiers or characters. The identifiers and characters within a single enumeration type must be distinct, however they may be reused in several different enumeration types.

The syntax for declaring an enumeration type is:

enumeration_type_definition ::= (enumeration_literal { , enumeration_literal })

enumeration_literal ::= identifier | character_literal

Some examples are:

type logic_level **is** (unknown, low, undriven, high);

type alu_function **is** (disable, pass, add, subtract, multiply, divide);

type octal_digit **is** ('0', '1', '2', '3', '4', '5', '6', '7');

There are a number of predefined enumeration types, defined as follows:

type severity_level **is** (note, warning, error, failure);

type boolean **is** (false, true);

type bit **is** ('0', '1');

type character **is** (

NUL,	SOH,	STX,	ETX,	EOT,	ENQ,	ACK,	BEL,
BS,	HT,	LF,	VT,	FF,	CR,	SO,	SI,
DLE,	DC1,	DC2,	DC3,	DC4,	NAK,	SYN,	ETB,
CAN,	EM,	SUB,	ESC,	FSP,	GSP,	RSP,	USP,
' ',	'!',	'"',	'#',	'$',	'%',	'&',	''',
'(',	')',	'*',	'+',	',',	'-',	'.',	'/',
'0',	'1',	'2',	'3',	'4',	'5',	'6',	'7',
'8',	'9',	':',	';',	'<',	'=',	'>',	'?',
'@',	'A',	'B',	'C',	'D',	'E',	'F',	'G',
'H',	'I',	'J',	'K',	'L',	'M',	'N',	'O',
'P',	'Q',	'R',	'S',	'T',	'U',	'V',	'W',
'X',	'Y',	'Z',	'[',	'\',	']',	'^',	'_',
'`',	'a',	'b',	'c',	'd',	'e',	'f',	'g',
'h',	'i',	'j',	'k',	'l',	'm',	'n',	'o',
'p',	'q',	'r',	's',	't',	'u',	'v',	'w',
'x',	'y',	'z',	'{',	'|',	'}',	'~',	DEL);

Note that type character is an example of an enumeration type containing a mixture of identifiers and characters. Also, the characters '0' and '1' are members of both bit and character . Where '0' or '1' occur in a program, the context will be used to determine which type is being used.

2.2.5. Arrays

An array in VHDL is an indexed collection of elements all of the same type. Arrays may be one-dimensional (with one index) or multi-dimensional (with a number of indices). In addition, an array type may be constrained, in which the bounds for an index are established when the type is defined, or unconstrained, in which the bounds are established subsequently.

The syntax for declaring an array type is:

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
    array (index_subtype_definition { , index_subtype_definition })
        of *element*_subtype_indication

constrained_array_definition ::=
    array index_constraint of *element*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

index_constraint ::= (discrete_range { , discrete_range })

discrete_range ::= *discrete*_subtype_indication | range

Subtypes, referred to in this syntax specification, will be discussed in detail in Section2.2.7.

Some examples of constrained array type declarations:

```
type word is array (31 downto 0) of bit;

type memory is array (address) of word;

type transform is array (1 to 4, 1 to 4) of real;

type register_bank is array (byte range 0 to 132) of integer;
```

An example of an unconstrained array type declaration:

```
type vector is array (integer range <>) of real;
```

The symbol '<>' (called a box) can be thought of as a place-holder for the index range, which will be filled in later when the array type is used. For example, an object might be declared to be a vector of 20 elements by giving its type as:

```
vector(1 to 20)
```

There are two predefined array types, both of which are unconstrained. They are defined as:

```
type string is array (positive range <>) of character;

type bit_vector is array (natural range <>) of bit;
```

The types positive and natural are subtypes of integer, defined in Section2.2.7 below. The type bit_vector is particularly useful in modeling binary coded representations of values in simulations of digital systems.

An element of an array object can referred to by indexing the name of the object. For example, suppose a and b are one- and two-dimensional array objects respectively. Then the indexed names a(1) and b(1, 1) refer to elements of these arrays. Furthermore, a contiguous slice of a one-dimensional array can be referred to by using a range as an index. For example a(8 **to** 15) is an eight-element array which is part of the array a.

Sometimes you may need to write a literal value of an array type. This can be done using an array aggregate, which is a list of element values. Suppose we have an array type declared as:

```
type a is array (1 to 4) of character;
```

and we want to write a value of this type containing the elements 'f', 'o', 'o', 'd' in that order. We could write an aggregate with *positional* association as follows:

```
('f', 'o', 'o', 'd')
```

in which the elements are listed in the order of the index range, starting with the left bound of the range. Alternatively, we could write an aggregate with *named* association:

```
(1 => 'f', 3 => 'o', 4 => 'd', 2 => 'o')
```

In this case, the index for each element is explicitly given, so the elements can be in any order. Positional and named association can be mixed within an aggregate, provided all the positional associations come first. Also, the word **others** can be used in place of an index in a named association, indicating a value to be used for all elements not explicitly mentioned. For example, the same value as above could be written as:

```
('f', 4 => 'd', others => 'o')
```

2.2.6. Records

VHDL provides basic facilities for records, which are collections of named elements of possibly different types. The syntax for declaring record types is:

```
record_type_definition ::=
    record
        element_declaration
        { element_declaration }
    end record
element_declaration ::= identifier_list : element_subtype_definition ;
identifier_list ::= identifier { , identifier )
element_subtype_definition ::= subtype_indication
```

An example record type declaration:

```
type instruction is
    record
        op_code : processor_op;
        address_mode : mode;
        operand1, operand2: integer range 0 to 15;
    end  record;
```

When you need to refer to a field of a record object, you use a selected name. For example, suppose that r is a record object containing a field called f. Then the name r.f refers to that field.

As for arrays, aggregates can be used to write literal values for records. Both positional and named association can be used, and the same rules apply, with record field names being used in place of array index names.

2.2.7. Subtypes

The use of a subtype allows the values taken on by an object to be restricted or constrained subset of some base type. The syntax for declaring a subtype is:

```
subtype_declaration ::= subtype identifier is subtype_indication ;
subtype_indication ::= [ resolution_function_name ] type_mark [ constraint ]
type_mark ::= type_name | subtype_name
constraint ::= range_constraint | index_constraint
```

There are two cases of subtypes. Firstly a subtype may constrain values from a scalar type to be within a specified range (a range constraint). For example:

```
subtype pin_count is integer range 0 to 400;
subtype digits is character range '0' to '9';
```

Secondly, a subtype may constrain an otherwise unconstrained array type by specifying bounds for the indices. For example:

```
subtype id is string(1 to 20);
subtype word is bit_vector(31 downto 0);
```

There are two predefined numeric subtypes, defined as:

```
subtype natural is integer range 0 to highest_integer
subtype positive is integer range 1 to highest_integer
```

2.2.8. Object Declarations

An object is a named item in a VHDL description which has a value of a specified type. There are three classes of objects: constants, variables and signals. Only the first two will be discusses in this section; signals will be covered in Section3.2.1. Declaration and use of constants and variables is very much like their use in programming languages.

A constant is an object which is initialised to a specified value when it is created, and which may not be subsequently modified. The syntax of a constant declaration is:

```
constant_declaration ::=
    constant identifier_list : subtype_indication [ := expression ] ;
```

Constant declarations with the initialising expression missing are called deferred constants, and may only appear in package declarations (see Section2.5.3). The initial value must be given in the corresponding package body. Some examples:

```
constant e : real := 2.71828;

constant delay : Time := 5 ns;

constant max_size : natural;
```

A variable is an object whose value may be changed after it is created. The syntax for declaring variables is:

```
variable_declaration ::=
    variable identifier_list : subtype_indication [ := expression ] ;
```

The initial value expression, if present, is evaluated and assigned to the variable when it is created. If the expression is absent, a default value is assigned when the variable is created. The default value for scalar types is the leftmost value for the type, that is the first in the list of an enumeration type, the lowest in an ascending range, or the highest in a descending range. If the variable is a composite type, the default value is the composition of the default values for each element, based on the element types.

Some examples of variable declarations:

```
variable count : natural := 0;

variable trace : trace_array;
```

Assuming the type trace_array is an array of boolean, then the initial value of the variable trace is an array with all elements having the value false.

Given an existing object, it is possible to give an alternate name to the object or part of it. This is done using and alias declaration. The syntax is:

```
alias_declaration ::= alias identifier : subtype_indication is name ;
```

A reference to an alias is interpreted as a reference to the object or part corresponding to the alias. For example:

```
variable instr : bit_vector(31 downto 0);

alias op_code : bit_vector(7 downto 0) is instr(31 downto 24);
```

declares the name op_code to be an alias for the left-most eight bits of instr.

2.2.9. Attributes

Types and objects declared in a VHDL description can have additional information, called attributes, associated with them. There are a number of standard pre-defined attributes, and some of those for types and arrays

are discussed here. An attribute is referenced using the ' notation. For example,

```
thing'attr
```

refers to the attribute attr of the type or object thing.

Firstly, for any scalar type or subtype T, the following attributes can be used:

Attribute	Result
T'left	Left bound of T
T'right	Right bound of T
T'low	Lower bound of T
T'high	Upper bound of T

For an ascending range, T'left = T'low, and T'right = T'high. For a descending range, T'left = T'high, and T'right = T'low.

Secondly, for any discrete or physical type or subtype T, X a member of T, and N an integer, the following attributes can be used:

Attribute	Result
T'pos(X)	Position number of X in T
T'val(N)	Value at position N in T
T'leftof(X)	Value in T which is one position left from X
T'rightof(X)	Value in T which is one position right from X
T'pred(X)	Value in T which is one position lower than X
T'succ(X)	Value in T which is one position higher than X

For an ascending range, T'leftof(X) = T'pred(X), and T'rightof(X) = T'succ(X). For a descending range, T'leftof(X) = T'succ(X), and T'rightof(X) = T'pred(X).

Thirdly, for any array type or object A, and N an integer between 1 and the number of dimensions of A, the following attributes can be used:

Attribute	Result
A'left(N)	Left bound of index range of dim'n N of A
A'right(N)	Right bound of index range of dim'n N of A
A'low(N)	Lower bound of index range of dim'n N of A
A'high(N)	Upper bound of index range of dim'n N of A
A'range(N)	Index range of dim'n N of A
A'reverse_range(N)	Reverse of index range of dim'n N of A
A'length(N)	Length of index range of dim'n N of A

2.3. Expressions and Operators

Expressions in VHDL are much like expressions in other programming languages. An expression is a formula combining primaries with operators. Primaries include names of objects, literals, function calls and parenthesized expressions. Operators are listed in Table 2-1 in order of decreasing precedence.

The logical operators **and**, **or**, **nand**, **nor**, **xor** and **not** operate on values of type bit or boolean, and also on one-dimensional arrays of these types. For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result. For bit and

Highest precedence:	**	**abs**	**not**			
	*	/	**mod**	**rem**		
	+ (sign)	– (sign)				
	+	–	&			
	=	/=	<	<=	>	>=
Lowest precedence:	**and**	**or**	**nand**	**nor**	**xor**	

Table 7-1. Operators and precedence.

boolean operands, **and**, **or**, **nand**, and **nor** are 'short-circuit' operators, that is they only evaluate their right operand if the left operand does not determine the result. So **and** and **nand** only evaluate the right operand if the left operand is true or '1', and **or** and **nor** only evaluate the right operand if the left operand is false or '0'.

The relational operators =, /=, <, <=, > and >= must have both operands of the same type, and yield boolean results. The equality operators (= and /=) can have operands of any type. For composite types, two values are equal if all of their corresponding elements are equal. The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

The sign operators (+ and –) and the addition (+) and subtraction (–) operators have their usual meaning on numeric operands. The concatenation operator (&) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand. It can also concatenate a single new element to an array, or two individual elements to form an array. The concatenation operator is most commonly used with strings.

The multiplication (*) and division (/) operators work on integer, floating point and physical types types. The modulus (**mod**) and remainder (**rem**) operators only work on integer types. The absolute value (**abs**) operator works on any numeric type. Finally, the exponentiation (**) operator can have an integer or floating point left operand, but must have an integer right operand. A negative right operand is only allowed if the left operand is a floating point number.

2.4. Sequential Statements

VHDL contains a number of facilities for modifying the state of objects and controlling the flow of execution of models. These are discussed in this section.

2.4.1. Variable Assignment

As in other programming languages, a variable is given a new value using an assignment statement. The syntax is:

```
variable_assignment_statement ::= target := expression ;
target ::= name | aggregate
```

In the simplest case, the target of the assignment is an object name, and the value of the expression is given to the named object. The object and the value must have the same base type.

If the target of the assignment is an aggregate, then the elements listed must be object names, and the value of the expression must be a composite value of the same type as the aggregate. Firstly, all the names in the aggregate are evaluated, then the expression is evaluated, and lastly the components of the expression value are assigned to the named variables. This is effectively a parallel assignment. For example, if a variable r is a record with two fields a and b, then they could be exchanged by writing

```
(a => r.b, b => r.a) := r
```

(Note that this is an example to illustrate how such an assignment works; it is not an example of good programming practice!)

2.4.2. If Statement

The if statement allows selection of statements to execute depending on one or more conditions. The syntax is:

```
if_statement ::=
    if condition then
        sequence_of_statements
    { elsif condition then
        sequence_of_statements }
    [ else
        sequence_of_statements ]
    end if ;
```

The conditions are expressions resulting in boolean values. The conditions are evaluated successively until one found that yields the value true. In that case the corresponding statement list is executed. Otherwise, if the else clause is present, its statement list is executed.

2.4.3. Case Statement

The case statement allows selection of statements to execute depending on the value of a selection expression. The syntax is:

```
case_statement ::=
    case expression is
        case_statement_alternative
        { case_statement_alternative }
    end case ;

case_statement_alternative ::=
    when choices =>
        sequence_of_statements

choices ::= choice { | choice }

choice ::=
    simple_expression
    | discrete_range
    | element_simple_name
    | others
```

The selection expression must result in either a discrete type, or a one-dimensional array of characters. The alternative whose choice list includes the value of the expression is selected and the statement list executed. Note that all the choices must be distinct, that is, no value may be duplicated. Furthermore, all values must be represented in the choice lists, or the special choice **others** must be included as the last alternative. If no choice list includes the value of the expression, the others alternative is selected. If the expression results in an array, then the choices may be strings or bit strings.

Some examples of case statements:

```
case element_colour of
    when red =>
        statements for red;
    when green | blue =>
        statements for green or blue;
    when orange to turquoise =>
        statements for these colours;
end case;
```

```
case opcode of
    when X"00" => perform_add;
    when X"01" => perform_subtract;
    when others => signal_illegal_opcode;
end case;
```

2.4.4. Loop Statements

VHDL has a basic loop statement, which can be augmented to form the usual while and for loops seen in other programming languages. The syntax of the loop statement is:

```
loop_statement ::=
    [ loop_label : ]
        [ iteration_scheme ] loop
            sequence_of_statements
        end loop [ loop_label ] ;

iteration_scheme ::=
    while condition
    | for loop_parameter_specification

parameter_specification ::=
    identifier in discrete_range
```

If the iteration scheme is omitted, we get a loop which will repeat the enclosed statements indefinitely. An example of such a basic loop is:

```
loop
    do_something;
end loop;
```

The while iteration scheme allows a test condition to be evaluated before each iteration. The iteration only proceeds if the test evaluates to true. If the test is false, the loop statement terminates. An example:

```
while index < length and str(index) /= ' ' loop
    index := index + 1;
end loop;
```

The for iteration scheme allows a specified number of iterations. The loop parameter specification declares an object which takes on successive values from the given range for each iteration of the loop. Within the statements enclosed in the loop, the object is treated as a constant, and so may not be assigned to. The object does not exist beyond execution of the loop statement. An example:

```
for item in 1 to last_item loop
    table(item) := 0;
end loop;
```

There are two additional statements which can be used inside a loop to modify the basic pattern of iteration. The 'next' statement terminates execution of the current iteration and starts the subsequent iteration. The

'exit' statement terminates execution of the current iteration and terminates the loop. The syntax of these statements is:

```
next_statement ::= next [ loop_label ] [ when condition ] ;
exit_statement ::= exit [ loop_label ] [ when condition ] ;
```

If the loop label is omitted, the statement applies to the inner-most enclosing loop, otherwise it applies to the named loop. If the when clause is present but the condition is false, the iteration continues normally. Some examples:

```
for i in 1 to max_str_len loop
    a(i) := buf(i);
    exit when buf(i) = NUL;
end loop;

outer_loop : loop
    inner_loop : loop
        do_something;
        next outer_loop when temp = 0;
        do_something_else;
    end loop inner_loop;
end loop outer_loop;
```

2.4.5. Null Statement

The null statement has no effect. It may be used to explicitly show that no action is required in certain cases. It is most often used in case statements, where all possible values of the selection expression must be listed as choices, but for some choices no action is required. For example:

```
case controller_command is
    when forward => engage_motor_forward;
    when reverse => engage_motor_reverse;
    when idle => null;
end case;
```

2.4.6. Assertions

An assertion statement is used to verify a specified condition and to report if the condition is violated. The syntax is:

```
assertion_statement ::=
    assert condition
        [ report expression ]
        [ severity expression ] ;
```

If the report clause is present, the result of the expression must be a string. This is a message which will be reported if the condition is false. If it is omitted, the default message is "Assertion violation". If the severity clause is present the expression must be of the type severity_level. If it is omitted, the default is error. A simulator may terminate execution if an assertion violation occurs and the severity value is greater than some implementation dependent threshold. Usually the threshold will be under user control.

2.5. Subprograms and Packages

Like other programming languages, VHDL provides subprogram facilities in the form of procedures and functions. VHDL also provided a package facility for collecting declarations and objects into modular units. Packages also provide a measure of data abstraction and information hiding.

2.5.1. Procedures and Functions

Procedure and function subprograms are declared using the syntax:

```
subprogram_declaration ::= subprogram_specification ;
subprogram_specification ::=
    procedure designator [ ( formal_parameter_list ) ]
    | function designator [ ( formal_parameter_list ) ] return type_mark
```

A subprogram declaration in this form simply names the subprogram and specifies the parameters required. The body of statements defining the behaviour of the subprogram is deferred. For function subprograms, the declaration also specifies the type of the result returned when the function is called. This form of subprogram declaration is typically used in package specifications (see Section 2.5.3), where the subprogram body is given in the package body, or to define mutually recursive procedures.

The syntax for specifying the formal parameters of a subprogram is:

```
formal_parameter_list ::= parameter_interface_list
interface_list ::= interface_element { ; interface_element }
interface_element ::= interface_declaration
interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration
interface_constant_declaration ::=
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]
interface_variable_declaration ::=
    [ variable ] identifier_list : [ mode ] subtype_indication [ := static_expression ]
```

For now we will only consider constant and variable parameters, although signals can also be used(see Chapter3). Some examples will clarify this syntax. Firstly, a simple example of a procedure with no parameters:

```
procedure reset;
```

This simply defines reset as a procedure with no parameters, whose statement body will be given subsequently in the VHDL program. A procedure call to reset would be:

```
reset;
```

Secondly, here is a declaration of a procedure with some parameters:

```
procedure increment_reg(variable reg : inout word_32;
                        constant incr : in integer := 1);
```

In this example, the procedure increment_reg has two parameters, the first called reg and the second called incr. Reg is a variable parameter, which means that in the subprogram body, it is treated as a variable object and may be assigned to. This means that when the procedure is called, the actual parameter associated with reg must itself be a variable. The mode of reg is **inout**, which means that reg can be both read and assigned to. Other possible modes for subprogram parameters are **in**, which means that the parameter may only be read, and **out**, which means that the parameter may only be assigned to. If the mode is **inout** or **out**, then the word **variable** can be omitted and is assumed.

The second parameter, incr, is a constant parameter, which means that it is treated as a constant object in the subprogram statement body, and may not be assigned to. The actual parameter associated with incr when the procedure is called must be an expression. Given the mode of the

parameter, **in**, the word **constant** could be omitted and assumed. The expression after the assignment operator is a default expression, which is used if no actual parameter is associated with incr in a call to the procedure.

A call to a subprogram includes a list of actual parameters to be associated with the formal parameters. This association list can be position, named, or a combination of both. (Compare this with the format of aggregates for values of composite types.) A call with positional association lists the actual parameters in the same order as the formals. For example:

```
increment_reg(index_reg, offset–2);     -- add value to index_reg
increment_reg(prog_counter);            -- add 1 (default) to prog_counter
```

A call with named association explicitly gives the formal parameter name to be associated with each actual parameter, so the parameters can be in any order. For example:

```
increment_reg(incr => offset–2, reg => index_reg);
increment_reg(reg => prog_counter);
```

Note that the second call in each example does not give a value for the formal parameter incr, so the default value is used.

Thirdly, here is an example of function subprogram declaration:

```
function byte_to_int(byte : word_8) return integer;
```

The function has one parameter. For functions, the parameter mode must be **in**, and this is assumed if not explicitly specified. If the parameter class is not specified it is assumed to be **constant**. The value returned by the body of this function must be an integer.

When the body of a subprogram is specified, the syntax used is:

```
subprogram_body ::=
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ designator ] ;
subprogram_declarative_part ::= { subprogram_declarative_item }
subprogram_statement_part ::= { sequential_statement }
subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
```

The declarative items listed after the subprogram specification declare things which are to be used locally within the subprogram body. The names of these items are not visible outside of the subprogram, but are visible inside locally declared subprograms. Furthermore, these items shadow any things with the same names declared outside the subprogram.

When the subprogram is called, the statements in the body are executed until either the end of the statement list is encountered, or a return statement is executed. The syntax of a return statement is:

```
return_statement ::= return [ expression ] ;
```

If a return statement occurs in a procedure body, it must not include an expression. There must be at least one return statement in a function body, it must have an expression, and the function must complete by executing a return statement. The value of the expression is the valued returned to the function call.

Another point to note about function subprograms is that they may not have any side-effects. This means that no visible variable declared outside the function body may be assigned to or altered by the function. This includes passing a non-local variable to a procedure as a variable parameter with mode **out** or **inout**. The important result of this rule is that functions can be called without them having any effect on the environment of the call.

An example of a function body:

```
function byte_to_int(byte : word_8) return integer is
    variable result : integer := 0;
begin
    for index in 0 to 7 loop
        result := result*2 + bit'pos(byte(index));
    end loop;
    return result;
end byte_to_int;
```

2.5.2. Overloading

VHDL allows two subprograms to have the same name, provided the number or base types of parameters differs. The subprogram name is then said to be overloaded. When a subprogram call is made using an overloaded name, the number of actual parameters, their order, their base types and the corresponding formal parameter names (if named association is used) are used to determine which subprogram is meant. If the call is a function call, the result type is also used. For example, suppose we declared the two subprograms:

```
function check_limit(value : integer) return boolean;

function check_limit(value : word_32) return boolean;
```

Then which of the two functions is called depends on whether a value of type integer or word_8 is used as the actual parameter. So

```
test := check_limit(4095)
```

would call the first function, and

```
test := check_limit(X"0000_0FFF")
```

would call the second function.

The designator used to define a subprogram can be either an identifier or a string representing any of the operator symbols listed in Section2.3. The latter case allows extra operand types to be defined for those operators. For example, the addition operator might be overloaded to add word_32 operands by declaring a function:

```
function "+" (a, b : word_32) return word_32 is
begin
    return int_to_word_32( word_32_to_int(a) + word_32_to_int(b) );
end "+";
```

Within the body of this function, the addition operator is used to add integers, since its operands are both integers. However, in the expression:

```
X"1000_0010" + X"0000_FFD0"
```

the newly declared function is called, since the operands to the addition operator are both of type word_32. Note that it is also possible to call operators using the prefix notation used for ordinary subprogram calls, for example:

```
"+" (X"1000_0010", X"0000_FFD0")
```

2.5.3. Package and Package Body Declarations

A package is a collection of types, constants, subprograms and possibly other things, usually intended to implement some particular service or to isolate a group of related items. In particular, the details of constant values and subprogram bodies can be hidden from users of a package, with only their interfaces made visible.

A package may be split into two parts: a package declaration, which defines its interface, and a package body, which defines the deferred details. The body part may be omitted if there are no deferred details. The syntax of a package declaration is:

```
package_declaration ::=
    package identifier is
        package_declarative_part
    end [ package_simple_name ] ;

package_declarative_part ::= { package_declarative_item }

package_declarative_item ::=
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | alias_declaration
    | use_clause
```

The declarations define things which are to be visible to users of the package, and which are also visible inside the package body. (There are also other kinds of declarations which can be included, but they are not discussed here.)

An example of a package declaration:

```
package data_types is
    subtype address is bit_vector(24 downto 0);
    subtype data is bit_vector(15 downto 0);
    constant vector_table_loc : address;
    function data_to_int(value : data) return integer;
    function int_to_data(value : integer) return data;
end data_types;
```

In this example, the value of the constant vector_table_loc and the bodies of the two functions are deferred, so a package body needs to be given.

The syntax for a package body is:

```
package_body ::=
    package body package_simple_name is
        package_body_declarative_part
    end [ package_simple_name ] ;

package_body_declarative_part ::= { package_body_declarative_item }
```

```
package_body_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | alias_declaration
  | use_clause
```

Note that subprogram bodies may be included in a package body, whereas only subprogram interface declarations may be included in the package interface declaration.

The body for the package data_types shown above might be written as:

```
package body data_types is

    constant vector_table_loc : address := X"FFFF00";

    function data_to_int(value : data) return integer is
        body of data_to_int
    end data_to_int;

    function int_to_data(value : integer) return data is
        body of int_to_data
    end int_to_data;

end data_types;
```

In this package body, the value for the constant is specified, and the function bodies are given. The subtype declarations are not repeated, as those in the package declarations are visible in the package body.

2.5.4. Package Use and Name Visibility

Once a package has been declared, items declared within it can be used by prefixing their names with the package name. For example, given the package declaration in Section2.4.3 above, the items declared might be used as follows:

```
variable PC : data_types.address;

int_vector_loc := data_types.vector_table_loc + 4*int_level;

offset := data_types.data_to_int(offset_reg);
```

Often it is convenient to be able to refer to names from a package without having to qualify each use with the package name. This may be done using a use clause in a declaration region. The syntax is:

```
use_clause ::= use selected_name { , selected_name } ;

selected_name ::= prefix . suffix
```

The effect of the use clause is that all of the listed names can subsequently be used without having to prefix them. If all of the declared names in a package are to be used in this way, you can use the special suffix **all**, for example:

```
use data_types.all;
```

3. VHDL Describes Structure

In Section 1.1 we introduced some terminology for describing the structure of a digital system. In this chapter, we will look at how structure is described in VHDL.

3.1. Entity Declarations

A digital system is usually designed as a hierarchical collection of modules. Each module has a set of ports which constitute its interface to the outside world. In VHDL, an *entity* is such a module which may be used as a component in a design, or which may be the top level module of the design.

The syntax for declaring an entity is:

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity_simple_name ] ;

entity_header ::=
    [ formal_generic_clause ]
    [ formal_port_clause ]

generic_clause ::= generic ( generic_list ) ;

generic_list ::= generic_interface_list

port_clause ::= port ( port_list ) ;

port_list ::= port_interface_list

entity_declarative_part ::= { entity_declarative_item }
```

The entity declarative part may be used to declare items which are to be used in the implementation of the entity. Usually such declarations will be included in the implementation itself, so they are only mentioned here for completeness. Also, the optional statements in the entity declaration may be used to define some special behaviour for monitoring operation of the entity. Discussion of these will be deferred until Section6.5.

The entity header is the most important part of the entity declaration. It may include specification of *generic constants*, which can be used to control the structure and behaviour of the entity, and *ports*, which channel information into and out of the entity.

The generic constants are specified using an interface list similar to that of a subprogram declaration. All of the items must be of class constant. As a reminder, the syntax of an interface constant declaration is:

```
interface_constant_declaration ::=
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]
```

The actual value for each generic constant is passed in when the entity is used as a component in a design.

The entity ports are also specified using an interface list, but the items in the list must all be of class signal. This is a new kind of interface item not previously discussed. The syntax is:

```
interface_signal_declaration ::=
    [ signal ] identifier_list : [ mode ] subtype_indication [ bus ]
        [ := static_expression ]
```

Since the class must be signal, the word **signal** can be omitted and is assumed. The word **bus** may be used if the port is to be connected to more than one output (see Sections 6.1 and 6.2). As with generic constants the actual signals to be connected to the ports are specified when the entity is used as a component in a design.

To clarify this discussion, here are some examples of entity declarations:

```
entity processor is
    generic (max_clock_freq : frequency := 30 MHz);
    port (clock : in bit;
            address : out integer;
            data : inout word_32;
            control : out proc_control;
            ready : in bit);
end processor;
```

In this case, the generic constant max_clock_freq is used to specify the timing behaviour of the entity. The code describing the entity's behaviour would use this value to determine delays in changing signal values.

Next, an example showing how generic parameters can be used to specify a class of entities with varying structure:

```
entity ROM is
    generic (width, depth : positive);
    port (enable : in bit;
            address : in bit_vector(depth–1 downto 0);
            data : out bit_vector(width–1 downto 0) );
end ROM;
```

Here, the two generic constants are used to specify the number of data bits and address bits respectively for the read-only memory. Note that no default value is given for either of these constants. This means that when the entity is used as a component, actual values must be supplied for them.

Finally an example of an entity declaration with no generic constants or

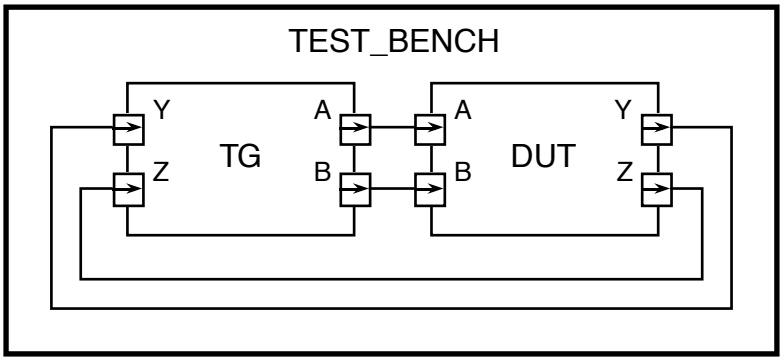


Figure 3-1. Test bench circuit.

ports:

```
entity test_bench is
end test_bench;
```

Though this might at first seem to be a pointless example, in fact it illustrates a common use of entities, shown in Figure3-1. A top-level entity for a design under test (DUT) is used as a component in a test bench circuit with another entity (TG) whose purpose is to generate test values. The values on signals can be traced using a simulation monitor, or checked directly by the test generator. No external connections from the test bench are needed, hence it has no ports.

3.2. Architecture Declarations

Once an entity has had its interface specified in an entity declaration, one or more implementations of the entity can be described in *architecture* bodies. Each architecture body can describe a different view of the entity. For example, one architecture body may purely describe the behaviour using the facilities covered in Chapters 2 and 4, whereas others may describe the structure of the entity as a hierarchically composed collection of components. In this section, we will only cover structural descriptions, deferring behaviour descriptions until Chapter4.

An architecture body is declared using the syntax:

```
architecture_body ::=
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture_simple_name ] ;
architecture_declarative_part ::= { block_declarative_item }
architecture_statement_part ::= { concurrent_statement }
block_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | alias_declaration
    | component_declaration
    | configuration_specification
    | use_clause
concurrent_statement ::=
    block_statement
    | component_instantiation_statement
```

The declarations in the architecture body define items that will be used to construct the design description. In particular, signals and components may be declared here and used to construct a structural description in terms of component instances, as illustrated in Section1.4. These are discussed in more detail in the next sections.

3.2.1. Signal Declarations

Signals are used to connect submodules in a design. They are declared using the syntax:

```
signal_declaration ::=
    signal identifier_list : subtype_indication [ signal_kind ] [ := expression ] ;
signal_kind ::= register | bus
```

Use of the signal kind specification is covered in Section6.2. Omitting the signal kind results in an ordinary signal of the subtype specified. The expression in the declaration is used to give the signal an initial value during the initialization phase of simulation. If the expression is omitted, a default initial value will be assigned.

One important point to note is that ports of an object are treated exactly as signals within that object.

3.2.2. Blocks

The submodules in an architecture body can be described as blocks. A block is a unit of module structure, with its own interface, connected to other blocks or ports by signals. A block is specified using the syntax:

```
block_statement ::=
    block_label :
        block [ ( guard_expression ) ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_label ] ;
block_header ::=
    [ generic_clause
    [ generic_map_aspect ; ] ]
    [ port_clause
    [ port_map_aspect ; ] ]
generic_map_aspect ::= generic map ( generic_association_list )
port_map_aspect ::= port map ( port_association_list )
block_declarative_part ::= { block_declarative_item }
block_statement_part ::= { concurrent_statement }
```

The guard expression is not covered in this booklet, and may be omitted. The block header defines the interface to the block in much the same way as an entity header defines the interface to an entity. The generic association list specifies values for the generic constants, evaluated in the context of the enclosing block or architecture body. The port map association list specifies which actual signals or ports from the enclosing block or architecture body are connected to the block's ports. Note that a block statement part may also contain block statements, so a design can be composed of a hierarchy of blocks, with behavioural descriptions at the bottom level of the hierarchy.

As an example, suppose we want to describe a structural architecture of the processor entity example in Section3.1. If we separate the processor into a control unit and a data path section, we can write a description as a pair of interconnected blocks, as shown in Figure3-2.

The control unit block has ports clk, bus_control and bus_ready, which are connected to the processor entity ports. It also has an output port for controlling the data path, which is connected to a signal declared in the architecture. That signal is also connected to a control port on the data path block. The address and data ports of the data path block are connected to the corresponding entity ports. The advantage of this modular decomposition is that each of the blocks can then be developed

```
architecture block_structure of processor is

    type data_path_control is ... ;

    signal internal_control : data_path_control;

begin

    control_unit : block
        port (clk : in bit;
                bus_control : out proc_control;
                bus_ready : in bit;
                control : out data_path_control);
        port map (clk => clock,
                        bus_control => control, bus_ready => ready;
                        control => internal_control);
        declarations for control_unit
    begin
        statements for control_unit
    end block control_unit;

    data_path : block
        port (address : out integer;
                data : inout word_32;
                control : in data_path_control);
        port map (address => address, data => data,
                        control => internal_control);
        declarations for data_path
    begin
        statements for data_path
    end block data_path;

end block_structure;
```

Figure3-2. Structural architecture of processor example.

independently, with the only effects on other blocks being well defined through their interfaces.

3.2.3. Component Declarations

An architecture body can also make use of other entities described separately and placed in design libraries. In order to do this, the architecture must declare a component, which can be thought of as a template defining a virtual design entity, to be instantiated within the architecture. Later, a configuration specification (see Section3.3) can be used to specify a matching library entity to use. The syntax of a component declaration is:

```
component_declaration ::=
    component identifier
        [ local_generic_clause ]
        [ local_port_clause ]
    end component ;
```

Some examples of component declarations:

```
component nand3
    generic (Tpd : Time := 1 ns);
    port (a, b, c : in logic_level;
            y : out logic_level);
end component;
```

```
component read_only_memory
    generic (data_bits, addr_bits : positive);
    port (en : in bit;
            addr : in bit_vector(depth–1 downto 0);
            data : out bit_vector(width–1 downto 0) );
end component;
```

The first example declares a three-input gate with a generic parameter specifying its propagation delay. Different instances can later be used with possibly different propagation delays. The second example declares a read-only memory component with address depth and data width dependent on generic constants. This component could act as a template for the ROM entity described in Section3.1.

3.2.4. Component Instantiation

A component defined in an architecture may be instantiated using the syntax:

```
component_instantiation_statement ::=
    instantiation_label :
        component_name
        [ generic_map_aspect ]
        [ port_map_aspect ] ;
```

This indicates that the architecture contains an instance of the named component, with actual values specified for generic constants, and with the component ports connected to actual signals or entity ports.

The example components declared in the previous section might be instantiated as:

```
enable_gate: nand3
    port map (a => en1, b => en2, c => int_req, y => interrupt);

parameter_rom: read_only_memory
    generic map (data_bits => 16, addr_bits => 8);
    port map (en => rom_sel, data => param, addr => a(7 downto 0);
```

In the first instance, no generic map specification is given, so the default value for the generic constant Tpd is used. In the second instance, values are specified for the address and data port sizes. Note that the actual signal associated with the port addr is a slice of an array signal. This illustrates that a port which is an array can be connected to part of a signal which is a larger array, a very common practice with bus signals.

4. VHDL Describes Behaviour

In Section 1.2 we stated that the behaviour of a digital system could be described in terms of programming language notation. The familiar sequential programming language aspects of VHDL were covered in detail in Chapter 2. In this chapter, we describe how these are extended to include statements for modifying values on signals, and means of responding to the changing signal values.

4.1. Signal Assignment

A signal assignment schedules one or more transactions to a signal (or port). The syntax of a signal assignment is:

signal_assignment_statement ::= target <= [**transport**] waveform ;

target ::= name | aggregate

waveform ::= waveform_element { , waveform_element }

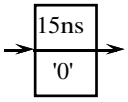
waveform_element ::=
*value*_expression [**after** *time*_expression]
| **null** [after *time*_expression]

The target must represent a signal, or be an aggregate of signals (see also variable assignments, Section 2.4.1). If the time expression for the delay is omitted, it defaults to 0 fs. This means that the transaction will be scheduled for the same time as the assignment is executed, but during the next simulation cycle.

Each signal has associated with it a *projected output waveform*, which is a list of transactions giving future values for the signal. A signal assignment adds transactions to this waveform. So, for example, the signal assignment:

```
s <= '0' after 10 ns;
```

will cause the signal enable to assume the value true 10 ns after the assignment is executed. We can represent the projected output waveform graphically by showing the transactions along a time axis. So if the above assignment were executed at time 5 ns, the projected waveform would be:

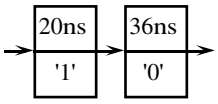


When simulation time reaches 15 ns, this transaction will be processed and the signal updated.

Suppose then at time 16 ns, the assignment:

```
s <= '1' after 4 ns, '0' after 20 ns;
```

were executed. The two new transactions are added to the projected output waveform:

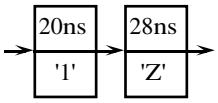


Note that when multiple transactions are listed in a signal assignment, the delay times specified must be in ascending order.

If a signal assignment is executed, and there are already old transactions from a previous assignmenton the projected output waveform, then some of the old transactions may be deleted. The way this is done depends on whether the word **transport** is included in the new assignment. If it is included, the assignment is said to use *transport delay*. In this case, all old transactions scheduled to occur after the first new transaction are deleted before the new transactions are added. It is as though the new transactions supercede the old ones. So given the projected output waveform shown immediately above, if the assignment:

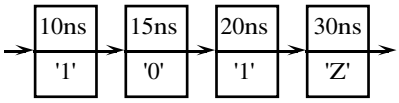
```
s <= transport 'Z' after 10 ns;
```

were executed at time 18 ns, then the transaction scheduled for 36 ns would be deleted, and the projected output waveform would become:



The second kind of delay, *inertial delay*, is used to model devices which do not respond to input pulses shorter than their output delay. An intertial delay is specified by omitting the word **transport** from the signal assignment. When an inertial delay transaction is added to a projected output waveform, firstly all old transactions scheduled to occur after the new transaction are deleted, and the new transaction is added, as in the case of transport delay. Next, all old transactions scheduled to occur before the new transaction are examined. If there are any with a different value from the new transaction, then all transactions up to the last one with a different value are deleted. The remaining transactions with the same value are left.

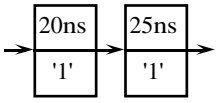
To illustrate this, suppose the projected output waveform at time 0 ns is:



and the assignment:

```
s <= '1' after 25 ns;
```

is executed also at 0 ns. Then the new projected ouptut waveform is:



When a signal assignment with multiple waveform elements is specified with intertial delay, only the first transaction uses inertial delay; the rest are treated as being transport delay transactions.

4.2. Processes and the Wait Statement

The primary unit of behavioural description in VHDL is the *process*. A process is a sequential body of code which can be activated in response to changes in state. When more than one process is activated at the same

time, they execute concurrently. A process is specified in a process statement, with the syntax:

```
process_statement ::=
    [ process_label : ]
        process [ ( sensitivity_list ) ]
            process_declarative_part
        begin
            process_statement_part
        end process [ process_label ] ;
process_declarative_part ::= { process_declarative_item }
process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | use_clause
process_statement_part ::= { sequential_statement }
sequential_statement ::=
    wait_statement
    | assertion_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
```

A process statement is a concurrent statement which can be used in an architecture body or block. The declarations define items which can be used locally within the process. Note that variables may be defined here and used to store state in a model.

A process may contain a number of signal assignment statements for a given signal, which together form a *driver* for the signal. Normally there may only be one driver for a signal, and so the code which determines a signals value is confined to one process.

A process is activated initially during the initialisation phase of simulation. It executes all of the sequential statements, and then repeats, starting again with the first statement. A process may suspended itself by executing a wait statement. This is of the form:

```
wait_statement ::=
    wait [ sensitivity_clause ] [ condition_clause ] [ timeout_clause ] ;
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name { , signal_name }
condition_clause ::= until condition
timeout_clause ::= for time_expression
```

The sensitivity list of the wait statement specifies a set of signals to which the process is sensitive while it is suspended. When an event occurs

on any of these signals (that is, the value of the signal changes), the process resumes and evaluates the condition. If it is true or if the condition is omitted, execution procedes with the next statement, otherwise the process resuspends. If the sensitivity clause is omitted, then the process is sensitive to all of the signals mentioned in the condition expression. The timeout expression must evaluate to a positive duration, and indicates the maximum time for which the process will wait. If it is omitted, the process may wait indefinitely.

If a sensitivity list is included in the header of a process statement, then the process is assumed to have an implicit wait statement at the end of its statement part. The sensitivity list of this implicit wait statement is the same as that in the process header. In this case the process may not contain any explicit wait statements.

An example of a process statements with a sensitivity list:

```
process (reset, clock)
    variable state : bit := false;
begin
    if reset then
        state := false;
    elsif clock = true then
        state := not state;
    end if;
    q <= state after prop_delay;
    -- implicit wait on reset, clock
end process;
```

During the initialization phase of simulation, the process is activated and assigns the initial value of state to the signal q. It then suspends at the implicit wait statement indicated in the comment. When either reset or clock change value, the process is resumed, and execution repeats from the beginning.

The next example describes the behaviour of a synchronization device called a Muller-C element used to construct asynchronous logic. The output of the device starts at the value '0', and stays at this value until both inputs are '1', at which time the output changes to '1'. The output then stays '1' until both inputs are '0', at which time the output changes back to '0'.

```
muller_c_2 : process
begin
    wait until a = '1' and b = '1';
    q <= '1';
    wait until a = '0' and b = '0';
    q <= '0';
end process muller_c_2 ;
```

This process does not include a sensitivity list, so explicit wait statements are used to control the suspension and activation of the process. In both wait statements, the sensitivity list is the set of signals a and b, determined from the condition expression.

4.3. Concurrent Signal Assignment Statements

Often a process describing a driver for a signal contains only one signal assignment statement. VHDL provides a convenient short-hand notation, called a concurrent signal assignment statement, for expressing such processes. The syntax is:

```
concurrent_signal_assignment_statement ::=
    [ label : ] conditional_signal_assignment
    | [ label : ] selected_signal_assignment
```

For each kind of concurrent signal assignment, there is a corresponding process statement with the same meaning.

4.3.1. Conditional Signal Assignment

A conditional signal assignment statement is a shorthand for a process containing signal assignments in an if statement. The syntax is:

```
conditional_signal_assignment ::= target <= options conditional_waveforms ;
options ::= [ guarded ] [ transport ]
conditional_waveforms ::=
    { waveform when condition else }
    waveform
```

Use of the word **guarded** is not covered in this booklet. If the word **transport** is included, then the signal assignments in the equivalent process use transport delay.

Suppose we have a conditional signal assignment:

```
s <= waveform_1 when condition_1 else
        waveform_2 when condition_2 else
        ...
        waveform_n;
```

Then the equivalent process is:

```
process
    if condition_1 then
        s <= waveform_1;
    elsif condition_2 then
        s <= waveform_2;
    elsif ...

    else
        s <= waveform_n;
    wait [ sensitivity_clause ];
end process;
```

If none of the waveform value expressions or conditions contains a reference to a signal, then the wait statement at the end of the equivalent process has no sensitivity clause. This means that after the assignment is made, the process suspends indefinitely. For example, the conditional assignment:

```
reset <= '1', '0' after 10 ns when short_pulse_required else
             '1', '0' after 50 ns;
```

schedules two transactions on the signal reset, then suspends for the rest of the simulation.

On the other hand, if there are references to signals in the waveform value expressions or conditions, then the wait statement has a sensitivity list consisting of all of the signals referenced. So the conditional assignment:

```
mux_out <= 'Z' after Tpd when en = '0' else
                in_0 after Tpd when sel = '0' else
                in_1 after Tpd;
```

is sensitive to the signals en and sel. The process is activated during the initialization phase, and thereafter whenever either of en or sel changes value.

The degenerate case of a conditional signal assignment, containing no conditional parts, is equivalent to a process containing just a signal assignment statement. So:

```
s <= waveform;
```

is equivalent to:

```
process
    s <= waveform;
    wait [ sensitivity_clause ];
end process;
```

4.3.2. Selected Signal Assignment

A selected signal assignment statement is a shorthand for a process containing signal assignments in a case statement. The syntax is:

```
selected_signal_assignment ::=
    with expression select
        target <= options selected_waveforms ;
selected_waveforms ::=
    { waveform when choices , }
    waveform when choices
choices ::= choice { | choice }
```

The options part is the same as for a conditional signal assignment. So if the word **transport** is included, then the signal assignments in the equivalent process use transport delay.

Suppose we have a selected signal assignment:

```
with expression select
    s <= waveform_1 when choice_list_1,
         waveform_2 when choice_list_2,
         ...
         waveform_n when choice_list_n;
```

Then the equivalent process is:

```
process
    case expression is
        when choice_list_1=>
            s <= waveform_1;
        when choice_list_2=>
            s <= waveform_2;
        ...
        when choice_list_n=>
            s <= waveform_n;
    end case;
    wait [ sensitivity_clause ];
end process;
```

The sensitivity list for the wait statement is determined in the same way as for a conditional signal assignment. That is, if no signals are referenced in the selected signal assignment expression or waveforms, the wait statement has no sensitivity clause. Otherwise the sensitivity clause contains all the signals referenced in the expression and waveforms.

An example of a selected signal assignment statement:

```
with alu_function select
    alu_result <= op1 + op2 when alu_add | alu_incr,
                  op1 – op2 when alu_subtract,
                  op1 and op2 when alu_and,
                  op1 or op2 when alu_or,
                  op1 and not op2 when alu_mask;
```

In this example, the value of the signal alu_function is used to select which signal assignment to alu_result to execute. The statement is sensitive to the signals alu_function, op1 and op2, so whenever any of these change value, the selected signal assignment is resumed.

5. Model Organisation

The previous chapters have described the various facilities of VHDL somewhat in isolation. The purpose of this chapter is to show how they are all tied together to form a complete VHDL description of a digital system.

5.1. Design Units and Libraries

When you write VHDL descriptions, you write them in a *design file*, then invoke a compiler to analyse them and insert them into a *design library*. A number of VHDL constructs may be separately analysed for inclusion in a design library. These constructs are called *library units*. The *primary* library units are entity declarations, package declarations and configuration declarations (see Section 5.2). The *secondary* library units are architecture bodies and package bodies. These library units depend on the specification of their interface in a corresponding primary library unit, so the primary unit must be analysed before any corresponding secondary unit.

A design file may contain a number of library units. The structure of a design file can be specified by the syntax:

```
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
context_clause ::= { context_item }
context_item ::= library_clause | use_clause
library_clause ::= library logical_name_list ;
logical_name_list ::= logical_name { , logical_name }
library_unit ::= primary_unit | secondary_unit
primary_unit ::=
    entity_declaration | configuration_declaration | package_declaration
secondary_unit ::= architecture_body | package_body
```

Libraries are referred to using identifiers called logical names. This name must be translated by the host operating system into an implementation dependent storage name. For example, design libraries may be implemented as database files, and the logical name might be used to determine the database file name. Library units in a given library can be referred to by prefixing their name with the library logical name. So for example, ttl_lib.ttl_10 would refer to the unit ttl_10 in library ttl_lib.

The context clause preceding each library unit specifies which other libraries it references and which packages it uses. The scope of the names made visible by the context clause extends until the end of the design unit.

There are two special libraries which are implicitly available to all design units, and so do not need to be named in a library clause. The first of these is called work, and refers to the working design library into which the

current design units will be placed by the analyser. Hence in a design unit, the previously analysed design units in a design file can be referred to using the library name work.

The second special libary is called std, and contains the packages standard and textio. Standard contains all of the predefined types and functions. All of the items in this package are implicitly visible, so no use clause is necessary to access them.

5.2. Configurations

In Sections 3.2.3 and 3.2.4 we showed how a structural description can declare a component specification and create instances of components. We mentioned that a component declared can be thought of as a template for a design entity. The binding of an entity to this template is achieved through a configuration declaration. This declaration can also be used to specify actual generic constants for components and blocks. So the configuration declaration plays a pivotal role in organising a design description in preparation for simulation or other processing.

The syntax of a configuration declaration is:

```
configuration_declaration ::=
    configuration identifier of entity_name is
        configuration_declarative_part
        block_configuration
    end [ configuration_simple_name ] ;

configuration_declarative_part ::= { configuration_declarative_item }

configuration_declarative_item ::= use_clause

block_configuration ::=
    for block_specification
        { use_clause }
        { configuration_item }
    end for ;

block_specification ::= architecture_name | block_statement_label

configuration_item ::= block_configuration | component_configuration

component_configuration ::=
    for component_specification
        [ use binding_indication ; ]
        [ block_configuration ]
    end for ;

component_specification ::= instantiation_list : component_name

instantiation_list ::=
    instantiation_label { , instantiation_label )
    | others
    | all

binding_indication ::=
    entity_aspect
    [ generic_map_aspect ]
    [ port_map_aspect ]

entity_aspect ::=
    entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
    | open

generic_map_aspect ::= generic map ( generic_association_list )
```

```
entity processor is
    generic (max_clock_speed : frequency := 30 MHz);
    port ( port list );
end processor;

architecture block_structure of processor is

    declarations

begin

    control_unit : block
        port ( port list );
        port map ( association list );
        declarations for control_unit
    begin
        statements for control_unit
    end block control_unit;

    data_path : block
        port ( port list );
        port map ( association list );
        declarations for data_path
    begin
        statements for data_path
    end block data_path;

end block_structure;
```

Figure 5-1. Example processor entity and architecture body.

port_map_aspect ::= **port map** (*port*_association_list)

The declarative part of the configuration declaration allows the configuration to use items from libraries and packages. The outermost block configuration in the configuration declaration defines the configuration for an architecture of the named entity. For example, in Chapter 3 we had an example of a processor entity and architecture, outlined again in Figure5-1. The overall structure of a configuration declaration for this architecture might be:

```
configuration test_config of processor is

    use work.processor_types.all

    for block_structure
        configuration items
    end for;

end test_config;
```

In this example, the contents of a package called processor_types in the current working library are made visible, and the block configuration refers to the architecture block_structure of the entity processor.

Within the block configuration for the architecture, the submodules of the architecture may be configured. These submodules include blocks and component instances. A block is configured with a nested block configuration. For example, the blocks in the above architecture can be configured as shown in Figure5-2.

Where a submodule is an instance of a component, a component configuration is used to bind an entity to the component instance. To illustrate, suppose the data_path block in the above example contained an

```
configuration test_config of processor is
    use work.processor_types.all
    for block_structure
        for control_unit
            configuration items
        end for;
        for data_path
            configuration items
        end for;
    end for;
end test_config;
```

Figure5-2. Configuration of processor example.

```
    data_path : block
        port ( port list );
        port map ( association list );
        component alu
            port (function : in alu_function;
                  op1, op2 : in bit_vector_32;
                  result : out bit_vector_32);
        end component;
        other declarations for data_path
    begin
        data_alu : alu
            port map (function => alu_fn, op1 => b1, op2 => b2, result => alu_r);
        other statements for data_path
    end block data_path;
```

Figure5-3. Structure of processor data-path block.

instance of the component alu, declared as shown in Figure5-3. Suppose also that a library project_cells contains an entity called alu_cell defined as:

```
entity alu_cell is
    generic (width : positive);
    port (function_code : in alu_function;
          operand1, operand2 : in bit_vector(width-1 downto 0);
          result : out bit_vector(width-1 downto 0);
          flags : out alu_flags);
end alu_cell;
```

with an architecture called behaviour. This entity matches the alu component template, since its operand and result ports can be constrained to match those of the component, and the flags port can be left unconnected. A block configuration for data_path could be specified as shown in Figure5-4.

Alternatively, if the library also contained a configuration called alu_struct for an architecture structure of the entity alu_cell, then the block configuration could use this, as shown in Figure5-5.

```
for data_path
    for data_alu : alu
        use entity project_cells.alu_cell(behaviour)
            generic map (width => 32)
            port map (function_code => function, operand1 => op1, operand2 => op2,
                      result => result, flags => open);
    end for;
    other configuration items
end for;
```

Figure5-4. Block configuration using library entity.

```
for data_path
    for data_alu : alu
        use configuration project_cells.alu_struct
            generic map (width => 32)
            port map (function_code => function, operand1 => op1, operand2 => op2,
                      result => result, flags => open);
    end for;
    other configuration items
end for;
```

Figure5-5. Block configuration using another configuration.

5.3. Complete Design Example

To illustrate the overall structure of a design description, a complete design file for the example in Section1.4 is shown in Figure5-6. The design file contains a number of design units which are analysed in order. The first design unit is the entity declaration of count2. Following it are two secondary units, architectures of the count2 entity. These must follow the entity declaration, as they are dependent on it. Next is another entity declaration, this being a test bench for the counter. It is followed by a secondary unit dependent on it, a structural description of the test bench. Following this is a configuration declaration for the test bench. It refers to the previously defined library units in the working library, so no library clause is needed. Notice that the count2 entity is referred to in the configuration as work.count2, using the library name. Lastly, there is a configuration declaration for the test bench using the structural architecture of count2. It uses two library units from a separate reference library, misc. Hence a library clause is included before the configuration declaration. The library units from this library are referred to in the configuration as misc.t_flipflop and misc.inverter.

This design description includes all of the design units in one file. It is equally possible to separate them into a number of files, with the opposite extreme being one design unit per file. If multiple files are used, you need to take care that you compile the files in the correct order, and re-compile dependent files if changes are made to one design unit. Source code control systems can be of use in automating this process.

```
-- primary unit: entity declaration of count2

entity count2 is
    generic (prop_delay : Time := 10 ns);
    port (clock : in bit;
          q1, q0 : out bit);
end count2;

-- secondary unit: a behavioural architecture body of count2

architecture behaviour 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 behaviour;

-- secondary unit: a structural architecture body of count2

architecture structure of count2 is

component t_flipflop
        port (ck : in bit;  q : out bit);
    end component;

    component inverter
        port (a : in bit;  y : out bit);
    end component;

    signal ff0, ff1, inv_ff0 : bit;

begin

    bit_0 : t_flipflop port map (ck => clock, q => ff0);

    inv : inverter port map (a => ff0, y => inv_ff0);

    bit_1 : t_flipflop port map (ck => inv_ff0, q => ff1);

    q0 <= ff0;
    q1 <= ff1;

end structure;
```

Figure5-6. Complete design file.

```
-- primary unit: entity declaration of test bench

entity test_count2 is
end test_count2;

-- secondary unit: structural architecture body of test bench

architecture structure of test_count2 is

    signal clock, q0, q1 : bit;

    component count2
        port (clock : in bit;
                q1, q0 : out bit);
    end component;

begin

    counter : count2
        port map (clock => clock, q0 => q0, q1 => q1);

    clock_driver : process
    begin
        clock <= '0', '1' after 50 ns;
        wait for 100 ns;
    end process clock_driver;

end structure;


-- primary unit: configuration using behavioural architecture

configuration test_count2_behaviour of test_count2 is

    for structure     -- of test_count2
        for counter : count2
            use entity work.count2(behaviour);
        end for;
    end for;

end test_count2_behaviour;


-- primary unit: configuration using structural architecture

library misc;

configuration test_count2_structure of test_count2 is

    for structure     -- of test_count2
        for counter : count2
            use entity work.count2(structure);
            for structure     -- of count_2
                for all : t_flipflop
                    use entity misc.t_flipflop(behaviour);
                end for;
                for all : inverter
                    use entity misc.inverter(behaviour);
                end for;
            end for;
        end for;
    end for;

end test_count2_structure;
```

Figure5-6 (continued).

6. Advanced VHDL

This chapter describes some more advanced facilities offered in VHDL. Although you can write many models using just the parts of the language covered in the previous chapters, you will find the features described here will significantly extend your model writing abilities.

6.1. Signal Resolution and Buses

In many digital sytems, buses are used to connect a number of output drivers to a common signal. For example, if open-collector or open-drain output drivers are used with a pull-up load on a signal, the signal can be pulled low by any driver, and is only pulled high by the load when all drivers are off. This is called a *wired-or* or *wired-and* connection. On the other hand, if tri-state drivers are used, at most one driver may be active at a time, and it determines the signal value.

VHDL normally allows only one driver for a signal. (Recall that a driver is defined by the signal assignments in a process.) In order to model signals with multiple drivers, VHDL uses the notion of *resolved types* for signals. A resolved type includes in its definition a resolution function, which takes the values of all the drivers contributing to a signal, and combines them to determine the final signal value.

A resolved type for a signal is declared using the syntax for a subtype:

subtype_indication ::= [*resolution_function*_name] type_mark [constraint]

The resolution function name is the name of a function previously defined. The function must take a parameter which is an unconstrained array of values of the signal subtype, and must return a result of that subtype. To illustrate, consider the declarations:

```
type logic_level is (L, Z, H);
type logic_array is array (integer range <>) of logic_level;

function resolve_logic (drivers : in logic_array) return logic_level;

subtype resolved_level is resolve_logic logic_level;
```

In this example, the type logic_level represents three possible states for a digital signal: low (L), high-impedance (Z) and high (H). The subtype resolved_level can be used to declare a resolved signal of this type. The resolution function might be implemented as shown in Figure6-1.

This function iterates over the array of drivers, and if any is found to have the value L, the function returns L. Otherwise the function returns H, since all drivers are either Z or H. This models a wired-or signal with a pull-up. Note that in some cases a resolution function may be called with an empty array as the parameter, and should handle that case appropriately. The example above handles it by returning the value H, the pulled-up value.

```
function resolve_logic (drivers : in logic_array) return logic_level;
begin
    for index in drivers'range loop
        if drivers(index) = L then
            return L;
        end if;
    end loop;
    return H;
end resolve_logic;
```

Figure 7-1. Resolution function for three-state logic

6.2. Null Transactions

VHDL provides a facility to model outputs which may be turned off (for example tri-state drivers). A signal assignment may specify that no value is to be assigned to a resolved signal, that is, that the driver should be disconnected. This is done with a null waveform element. Recall that the syntax for a waveform element is:

```
waveform_element ::=
    value_expression [ after time_expression ]
    | null [ after time_expression ]
```

So an example of such a signal assignment is:

```
d_out <= null after Toz;
```

If all of the drivers of a resolved signal are disconnected, the question of the resulting signal value arises. There are two possibilities, depending on whether the signal was declared with signal kind **register** or **bus**. For register kind signals, the most recently determined value remains on the signal. This can be used to model charge storage nodes in MOS logic families. For bus kind signals, the resolution function must determine the value for the signal when no drivers are contributing to it. This is how tri-state, open-collector and open-drain buses would typically be modeled.

6.3. Generate Statements

VHDL has an additional concurrent statement which can be used in architecture bodies to describe regular structures, such as arrays of blocks, component instances or processes. The syntax is:

```
generate_statement ::=
    generate_label :
        generation_scheme generate
            { concurrent_statement }
        end generate [ generate_label ] ;
generation_scheme ::=
    for generate_parameter_specification
    | if condition
```

The for generation scheme describes structures which have a repeating pattern. The if generation scheme is usually used to handle exception cases within the structure, such as occur at the boundaries. This is best illustrated by example. Suppose we want to describe the structure of an

```
adder : for i in 0 to width-1 generate

    ls_bit : if i = 0 generate
        ls_cell : half_adder port map (a(0), b(0), sum(0), c_in(1));
    end generate lsbit;

    middle_bit : if i > 0 and i < width-1 generate
        middle_cell : full_adder port map (a(i), b(i), c_in(i), sum(i), c_in(i+1));
    end generate middle_bit;

    ms_bit : if i = width-1 generate
        ms_cell : full_adder port map (a(i), b(i), c_in(i), sum(i), carry);
    end generate ms_bit;

end generate adder;
```

Figure6-2. Generate statement for adder.

adder constructed out of full-adder cells, with the exception of the least significant bit, which is consists of a half-adder. A generate statement to achieve this is shown in Figure6-2.

The outer generate statement iterates with i taking on values from 0 to width-1. For the least significant bit (i=0), an instance of a half adder component is generated. The input bits are connected to the least significant bits of a and b, the output bit is connected to the least significant bit of sum, and the carry bit is connectected to the carry in of the next stage. For intermediate bits, an instance of a full adder component is generated with inputs and outputs connected similarly to the first stage. For the most significant bit (i=width-1), an instance of the half adder is also generated, but its carry output bit is connected to the signal carry.

6.4. Concurrent Assertions and Procedure Calls

There are two kinds of concurrent statement which were not covered in previous chapters: concurrent assertions and concurrent procedure calls. A concurrent assertion statement is equivalent to a process containing only an assertion statement followed by a wait statement. The syntax is:

concurrent_assertion_statement ::= [label :] assertion_statement

The concurrent signal assertion:

L : **assert** *condition* **report** *error_string* **severity** *severity_value*;

is equivalent to the process:

```
L : process
begin
    assert condition report error_string severity severity_value;
    wait [ sensitivity_clause ] ;
end process L;
```

The sensitivity clause includes all the signals which are referred to in the condition expression. If no signals are referenced, the process is activated once at simulation initialisation, checks the condition, and then suspends indefinitely.

The other concurrent statement, the concurrent procedure call, is equivalent to a process containing only a procedure call followed by a wait statement. The syntax is:

```
concurrent_procedure_call ::= [ label : ] procedure_call_statement
```

The procedure may not have any formal parameters of class **variable**, since it is not possible for a variable to be visible at any place where a concurrent statement may be used. The sensitivity list of the wait statement in the process includes all the signals which are actual parameters of mode **in** or **inout** in the procedure call. These are the only signals which can be read by the called procedure.

Concurrent procedure calls are useful for defining process behaviour that may be reused in several places or in different models. For example, suppose a package bit_vect_arith declares the procedure:

```
procedure add(signal a, b : in bit_vector; signal result : out bit_vector);
```

Then an example of a concurrent procedure call using this procedure is:

```
adder : bit_vect_arith.add (sample, old_accum, new_accum);
```

This would be equivalent to the process:

```
adder : process
begin
    bit_vect_arith.add (sample, old_accum, new_accum);
    wait on sample, old_accum;
end process adder;
```

6.5. Entity Statements

In Section3.1, it was mentioned that an entity declaration may include statements for monitoring operation of the entity. Recall that the syntax for an entity declaration is:

```
entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity_simple_name ] ;
```

The syntax for the statement part is:

```
entity_statement_part ::= { entity_statement }
entity_statement ::=
    concurrent_assertion_statement
    | passive_concurrent_procedure_call
    | passive_process_statement
```

The concurrent statement that are allowed in an entity declaration must be *passive*, that is, they may not contain any signal assignments. (This includes signal assignments inside nested procedures of a process.) A result of this rule is that such processes cannot modify the state of the entity, or any circuit the entity may be used in. However, they can fully monitor the state, and so may be used to report erroneous operating conditions, or to trace the behavior of the design.

7. Sample Models: The DP32 Processor

This chapter contains an extended example, a description of a hypothetical processor called the DP32. The processor instruction set and bus architectures are first described, and then a behavioural description is given. A test bench model is constructed, and the model checked with a small test program. Next, the processor is decomposed into components at the register transfer level. A number of components are described, and a structural description of the processor is constructed using these components. The same test bench is used, but this time with the structural architecture.

7.1. Instruction Set Architecture

The DP32 is a 32-bit processor with a simple instruction set. It has a number of registers, shown in Figure 7-1. There are 256 general purpose registers (R0–R255), a program counter (PC) and a condition code register (CC). The general purpose registers are addressable by software, whereas the PC and CC registers are not.

On reset, the PC is initialised to zero, and all other registers are undefined. By convention, R0 is read-only and contains zero. This is not enforced by hardware, and the zero value must be loaded by software after reset.

The memory accessible to the DP32 consists of 32-bit words, addressed by a 32-bit word-address. Instructions are all multiples of 32-bit words, and are stored in this memory. The PC register contains the address of the next instruction to be executed. After each instruction word is fetched, the PC is incremented by one to point to the next word.

The three CC register bits are updated after each arithmetic or logical instruction. The Z (zero) bit is set if the result is zero. The N (negative) bit is set if the result of an arithmetic instruction is negative, and is undefined after logical instructions. The V(overflow) bit is set if the result of an arithmetic instruction exceeds the bounds of representable integers, and is

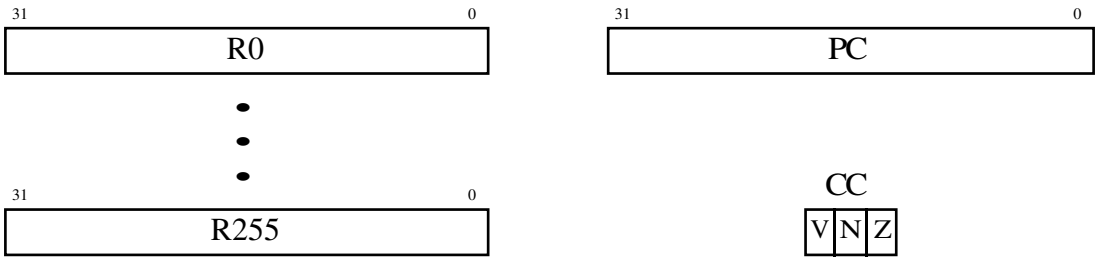


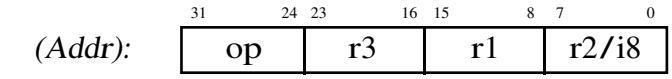
Figure 7-1. DP32 registers.

Instruction	Name	Function	opcode
Add	add	r3 ← r1 + r2	X"00"
Sub	subtract	r3 ← r1 – r2	X"01"
Mul	multiply	r3 ← r1 × r2	X"02"
Div	divide	r3 ← r1 ÷ r2	X"03"
Addq	add quick	r3 ← r1 + i8	X"10"
Subq	subtract quick	r3 ← r1 – i8	X"11"
Mulq	multiply quick	r3 ← r1 × i8	X"12"
Divq	divide quick	r3 ← r1 ÷ i8	X"13"
Land	logical and	r3 ← r1 & r2	X"04"
Lor	logical or	r3 ← r1 | r2	X"05"
Lxor	logical exclusive or	r3 ← r1 ⊕ r2	X"06"
Lmask	logical mask	r3 ← r1 & ~r2	X"07"

Table 7-1. DP32 arithmetic and logic instructions.

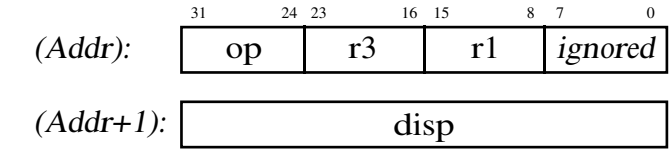
undefined after logical instructions.

The DP32 instruction set is divided into a number of encoding formats. Firstly, arithmetic and logical instructions are all one 32-bit word long, formatted as follows:

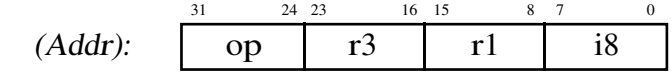


The op field is the op-code, r3 is the destination register address, r1 and r2 are source register addresses, and i8 is an immediate two-compliment integer operand. The arithmetic and logical instructions are listed in Table7-1.

Memory load and store instructions have two formats, depending on whether a long or short displacement value is used. The format for a long displacement is:



The format for a short displacement is:



The op field is the op-code, r3 specifies the register to be loaded or stored, r1 is used as an index register, disp is a long immediate displacement, and i8 is a short immediate displacement. The load and store instructions are listed in Table7-2.

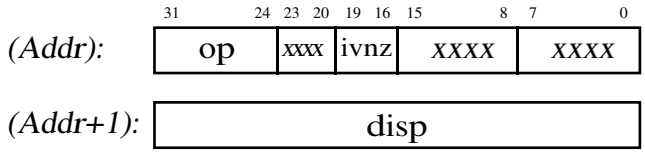
Instruction	Name	Function	opcode
Ld	load	r3 ← M[r1 + disp32]	X“20”
St	store	M[r1 + disp32] ← r3	X“21”
Ldq	load quick	r3 ← M[r1 + i8]	X“30”
Stq	store quick	M[r1 + i8] ← r3	X“31”

Table7-2. DP32 load and store instructions.

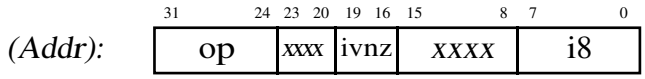
Instruction	Name	Function	opcode
Br-*ivnz*	branch	if *cond* then PC ← PC + disp32	X“40”
Brq-*ivnz*	branch quick	if *cond* then PC ← PC + i8	X“51”
Bi-*ivnz*	branch indexed	if *cond* then PC ← r1 + disp32	X“41”
Biq-*ivnz*	branch indexed quick	if *cond* then PC ← r1 + i8	X“51”

Table7-3. DP32 load and store instructions.

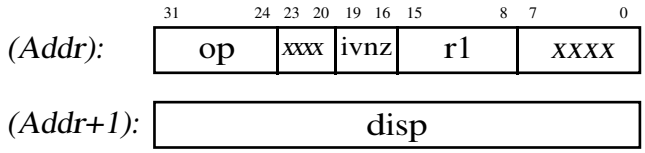
Finally, there are four branch instructions, listed in Table7-3, each with a slightly different format. The format of the ordinary brach is:



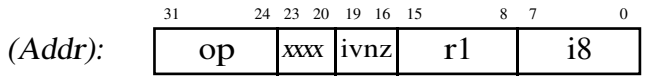
The format of a quick branch is:



The format of an indexed branch



The format of a quick indexed branch



The op field is the op-code, disp is a long immediate displacement, i8 is a short immediate displacement, r1 is used as an index register, and ivnz is a the condition mask. The branch is taken if

cond ≡ ((V & v) | (N & n) | (Z & z)) = i.

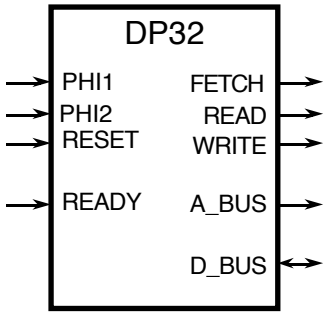


Figure7-2. DP32 port diagram.

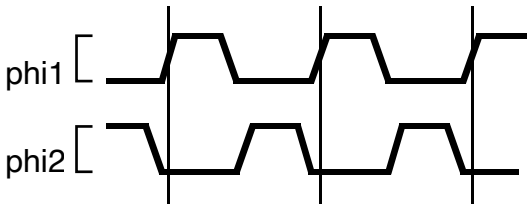


Figure7-3. DP32 clock waveforms.

7.2. Bus Architecture

The DP32 processor communicates with its memory over synchronous 32-bit address and data buses. The external ports of the DP32 are shown in Figure7-2.

The two clock inputs, phi1 and phi2, provide a two-phase non-overlapping clock for the processor. The clock waveforms are shown in Figure7-3. Each cycle of the phi1 clock defines a bus state, one of Ti (idle), T1 or T2. Bus transactions consist of a T1 state followed by one or more T2 states, with Ti states between transactions.

The port a_bus is a 32-bit address bus, and d_bus is a 32-bit bidirection data bus. The read and write ports control bus read and write transactions. The fetch port is a status signal indicating that a bus read in progress is an instruction fetch. The ready input is used by a memory device to indicate that read data is available or write data has been accepted.

The timing for a bus read transaction is show in Figure7-4. During an idle state, Ti, the processor places the memory address on the address bus to start the transaction. The next state is a T1 state. After the leading edge of the phi1 clock, the processor asserts the read control signal, indicating that the address is valid and the memory should start the read transaction. The processor also asserts the fetch signal if it is reading instructions. It always leaves the write signal negated during read transactions. During the T1 state and the following T2 state, the memory accesses the requested data, and places it on the data bus. If it has completed the data access by the end of the T2 state, it asserts ready. The processor accepts the data, and completes the transaction. On the other hand, if the memory has not yet supplied the data by the end of the T2 state, it leaves ready false. The processor then repeats T2 states until it detects ready true. By this means, a slow memory can extend the transaction until it has read the data. At the end of the transaction, the processor returns its control outputs to their default values, and the memory negates ready and removes the data from the data bus. The processor continues with idle states until the next transaction is required.

The timing for a bus write transaction is show in Figure7-5. Here also, the transaction starts with the processor placing the address on the address bus during a Ti state. After the leading edge of phi1 during the subsequent T1 state, the processor negates fetch and asserts write. The read signal remains false for the whole transaction. During the T1 state, the processor also makes the data to be written available on the data bus. The memory

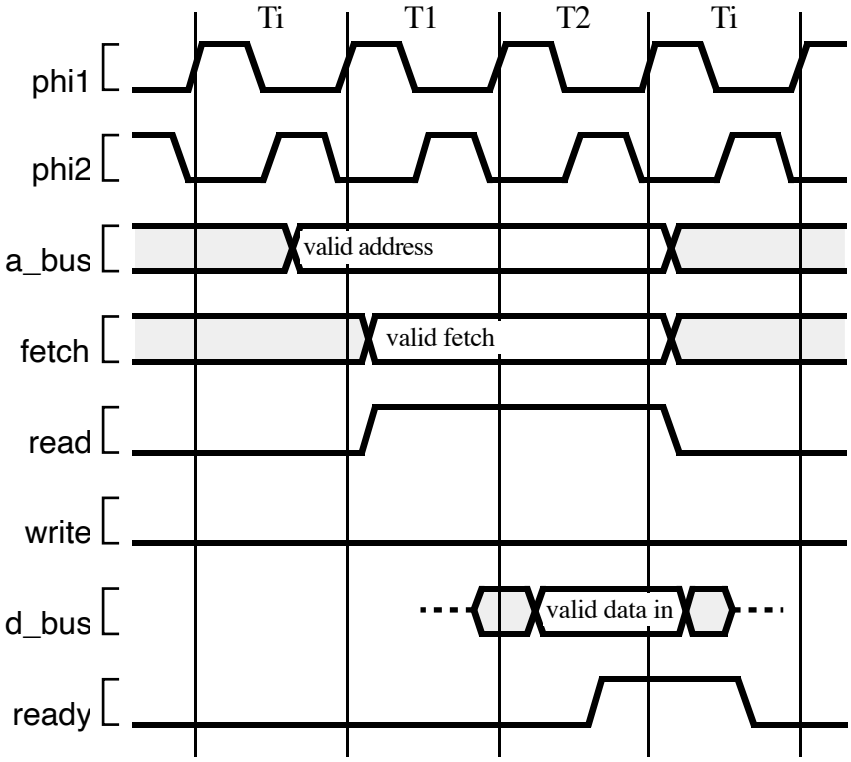


Figure7-4. DP32 bus read transaction.

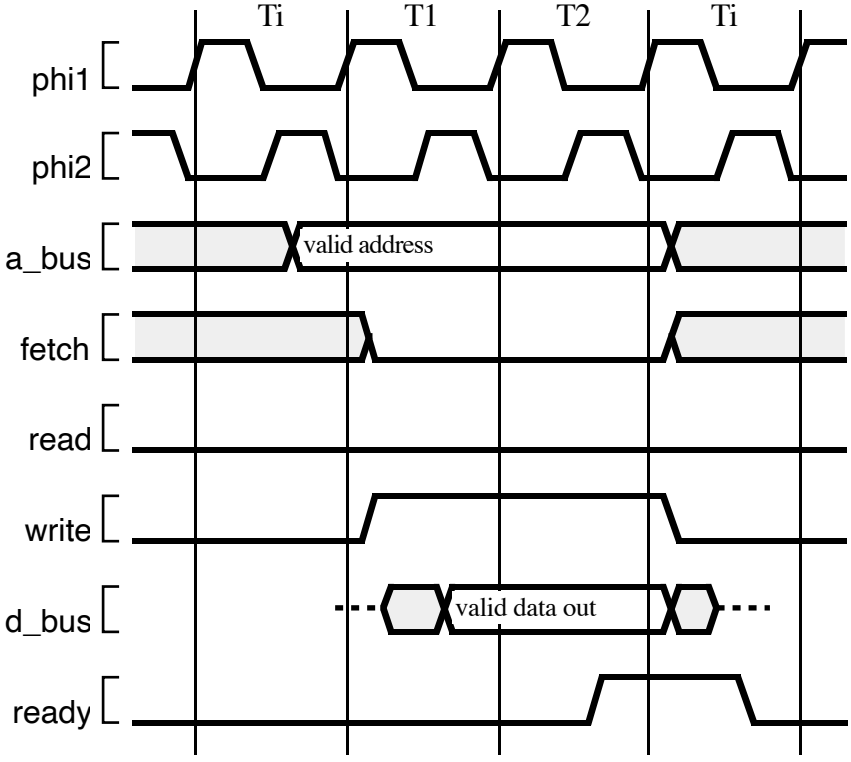


Figure7-5. DP32 bus write transaction.

can accept this data during the T1 and subsequent T2 states. If it has completed the write by the end of the T2 state, it asserts ready. The processor then completes the transaction and continutes with Ti states, and the memory removes the data from the data bus and negates ready. If the memory has not had time to complete the write by the end of the T2 state, it leaves ready false. The processor will then repeat T2 states until it detects ready true.

7.3. Types and Entity

We start the description of the DP32 processor by defining a package containing the data types to be used in the model, and some useful operations on those types. The package declaration of dp32_types is listed in Figure7-6.

```
package dp32_types is

    constant unit_delay : Time := 1 ns;

    type bool_to_bit_table is array (boolean) of bit;
    constant bool_to_bit : bool_to_bit_table;

    subtype bit_32 is bit_vector(31 downto 0);
    type bit_32_array is array (integer range <>) of bit_32;
    function resolve_bit_32 (driver : in bit_32_array) return bit_32;
    subtype bus_bit_32 is resolve_bit_32 bit_32;

    subtype bit_8 is bit_vector(7 downto 0);

    subtype CC_bits is bit_vector(2 downto 0);
    subtype cm_bits is bit_vector(3 downto 0);

    constant op_add :       bit_8 := X"00";
    constant op_sub :       bit_8 := X"01";
    constant op_mul :    bit_8 := X"02";
    constant op_div :    bit_8 := X"03";
    constant op_addq :   bit_8 := X"10";
    constant op_subq :   bit_8 := X"11";
    constant op_mulq :   bit_8 := X"12";
    constant op_divq :   bit_8 := X"13";
    constant op_land :   bit_8 := X"04";
    constant op_lor :    bit_8 := X"05";
    constant op_lxor :       bit_8 := X"06";
    constant op_lmask :  bit_8 := X"07";
    constant op_ld :     bit_8 := X"20";
    constant op_st :     bit_8 := X"21";
    constant op_ldq :        bit_8 := X"30";
    constant op_stq :        bit_8 := X"31";
    constant op_br :     bit_8 := X"40";
    constant op_brq :        bit_8 := X"50";
    constant op_bi :     bit_8 := X"41";
    constant op_biq :        bit_8 := X"51";

    function bits_to_int (bits : in bit_vector) return integer;
    function bits_to_natural (bits : in bit_vector) return natural;
    procedure int_to_bits (int : in integer; bits : out bit_vector);

end dp32_types;
```

Figure7-6. Package declaration for dp32_types.

The constant unit_delay is used as the default delay time through-out the DP32 description. This approach is common when writing models to describe the function of a digital system, before developing a detailed timing model.

The constant bool_to_bit is a lookup table for converting between boolean conditions and the type bit. Examples of its use will be seen later. Note that it is a deferred constant, so its value will be given in the package body.

The next declarations define the basic 32-bit word used in the DP32 model. The function resolve_bit_32 is a resolution function used to determine the value on a 32-bit bus with multiple drivers. Such a bus is declared with the subtype bus_bit_32, a resolved type.

The subtype bit_8 is part of a 32-bit word used as an op-code or register address. CC_bits is the type for condition codes, and cm_bits is the type for the condition mask in a branch op-code.

The next set of constant declarations define the op-code bit patterns for valid op-codes. These symbolic names are used as a matter of good coding style, enabling the op-code values to be changed without having to modify the model code in numerous places.

Finally, a collection of conversion functions between bit-vector values and numeric values is defined. The bodies for these subprograms are hidden in the package body.

The body of the dp32_types package is listed in Figure7-7. Firstly the value for the deferred constant bool_to_bit is given: false translates to '0' and true translates to '1'. An example of the use of this table is:

```
flag_bit <= bool_to_bit(flag_condition);
```

Next, the body of the resolution function for 32-bit buses is defined. The function takes as its parameter an unconstrained array of bit_32 values, and produces as a result the bit-wide logical-or of the values. Note that the function cannot assume that the length of the array will be greater than one. If no drivers are active on the bus, an empty array will be passed to the resolution function. In this case, the default value of all '0' bits (float_value) is used as the result.

```
package body dp32_types is

    constant bool_to_bit : bool_to_bit_table :=
        (false => '0', true => '1');

    function resolve_bit_32 (driver : in bit_32_array) return bit_32 is

        constant float_value : bit_32 := X"0000_0000";
        variable result : bit_32 := float_value;

    begin
        for i in driver'range loop
            result := result or driver(i);
        end loop;
        return result;
    end resolve_bit_32;
```

Figure7-7. Package body for dp32_types.

The function bits_to_int converts a bit vector representing a twos-compliment signed integer into an integer type value. The local variable temp is declared to be a bit vector of the same size and index range as the parameter bits. The variable result is initialised to zero when the function is invoked, and subsequently used to accumulate the weighted bit values in

```
    function bits_to_int (bits : in bit_vector) return integer is

        variable temp : bit_vector(bits'range);
        variable result : integer := 0;

    begin
        if bits(bits'left) = '1' then          -- negative number
            temp := not bits;
        else
            temp := bits;
        end if;
        for index in bits'range loop           -- sign bit of temp = '0'
            result := result * 2 + bit'pos(temp(index));
        end loop;
        if bits(bits'left) = '1' then
            result := (-result) - 1;
        end if;
        return result;
    end bits_to_int;

    function bits_to_natural (bits : in bit_vector) return natural is

        variable result : natural := 0;

    begin
        for index in bits'range loop
            result := result * 2 + bit'pos(bits(index));
        end loop;
        return result;
    end bits_to_natural;

    procedure int_to_bits (int : in integer; bits : out bit_vector) is

        variable temp : integer;
        variable result : bit_vector(bits'range);

    begin
        if int < 0 then
            temp := -(int+1);
        else
            temp := int;
        end if;
        for index in bits'reverse_range loop
            result(index) := bit'val(temp rem 2);
            temp := temp / 2;
        end loop;
        if int < 0 then
            result := not result;
            result(bits'left) := '1';
        end if;
        bits := result;
    end int_to_bits;

end dp32_types;
```

Figure7-7 (continued).

```
use work.dp32_types.all;
entity dp32 is
    generic (Tpd : Time := unit_delay);
    port (d_bus : inout bus_bit_32 bus;
          a_bus : out bit_32;
          read, write : out bit;
          fetch : out bit;
          ready : in bit;
          phi1, phi2 : in bit;
          reset : in bit);
end dp32;
```

Figure7-8. Entity declaration for dp32.

the for loop. The function bits_to_natural performs a similar function to bits_to_int, but does not need to do any special processing for negative numbers. Finally, the function int_to_bits performs the inverse of bits_to_int.

The entity declaration of the DP32 processor is shown in Figure7-8. The library unit is preceded by a use clause referencing all the items in the package dp32_types. The entity has a generic constant Tpd used to specify the propagation delays between input events and output signal changes. The default value is the unit delay specified in the dp32_types package. There are a number of ports corresponding to those shown in Figure7-2. The reset, clocks, and bus control signals are represented by values of type bit. The address bus output is a simple bit-vector type, as the processor is the only module driving that bus. On the other hand, the data bus is a resolved bit-vector type, as it may be driven by both the processor and a memory module. The word **bus** in the port declaration indicates that all drivers for the data bus may be disconnected at the same time (ie, none of them is driving the bus).

7.4. Behavioural Description

In this section a behavioural model of the DP32 processor will be presented. This model can be used to run test programs in the DP32 instruction set by connecting it to a simulated memory model. The architecture body for the behavioural description is listed in Figure7-9.

The declaration section for the architecture body contains the declaration for the DP32 register file type, and array of 32-bit words, indexed by a natural number constrained to be in the range 0 to 255.

The architecture body contains only one concurrent statement, namely an anonymous process which implements the behaviour as a sequential algorithm. This process declares a number of variables which represent the internal state of the processor: the register file (reg), the program counter (PC), and the current instruction register (current_instr). A number of working variables and aliases are also declared.

The procedure memory_read implements the behavioural model of a memory read transaction. The parameters are the memory address to read from, a flag indicating whether the read is an instruction fetch, and a result parameter returning the data read. The procedure refers to the

entity ports, which are visible because they are declared in the parent of the procedure.

The memory_read model firstly drives the address and fetch bit ports, and then waits until the next leading edge of phi1, indicating the start of the next clock cycle. (The wait statement is sensitive to a change from '0' to '1' on phi1.) When that event occurs, the model checks the state of the reset input port, and if it is set, immediately returns without further action. If reset is clear, the model starts a T1 state by asserting the read bit port a propagation delay time after the clock edge. It then waits again until the next phi1 leading edge, indicating the start of the next clock cycle. Again, it checks reset and discontinues if reset is set. The model then starts a loop executing T2 states. It waits until phi2 changes from '1' to '0' (at the end of the cycle), and then checks reset again, returning if it is set. Otherwise it checks the ready bit input port, and if set, accepts the data from the data bus port and exits the loop. If ready is not set, the loop repeats, adding another T2 state to the transaction. After the loop, the model waits for the next clock edge indicating the start of the Ti state at the end of the transaction. After checking reset again, the model clears ready to complete the transaction, and returns to the parent process.

The procedure memory_write is similar, implementing the model for a memory write transaction. The parameters are simply the memory address to write to, and the data to write. The model similarly has reset checks after each wait point. One difference is that at the end of the transaction, there is a null signal assignment to the data bus port. This models the bahaviour of the processor disconnecting from the data bus, that is, at this point it stops driving the port.

```
use work.dp32_types.all;

architecture behaviour of dp32 is

    subtype reg_addr is natural range 0 to 255;
    type reg_array is array (reg_addr) of bit_32;

begin -- behaviour of dp32

    process

        variable reg : reg_array;
        variable PC : bit_32;
        variable current_instr : bit_32;
        variable op: bit_8;
        variable r3, r1, r2 : reg_addr;
        variable i8 : integer;
        alias cm_i : bit is current_instr(19);
        alias cm_V : bit is current_instr(18);
        alias cm_N : bit is current_instr(17);
        alias cm_Z : bit is current_instr(16);
        variable cc_V, cc_N, cc_Z : bit;
        variable temp_V, temp_N, temp_Z : bit;
        variable displacement, effective_addr : bit_32;
```

Figure7-9. Behavioural architecture body for dp32.

```
procedure memory_read (addr : in bit_32;
                       fetch_cycle : in boolean;
                       result : out bit_32) is
begin
    -- start bus cycle with address output
    a_bus <= addr after Tpd;
    fetch <= bool_to_bit(fetch_cycle) after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T1 phase
    --
    read <= '1' after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T2 phase
    --
    loop
        wait until phi2 = '0';
        if reset = '1' then
            return;
        end if;
        -- end of T2
        if ready = '1' then
            result := d_bus;
            exit;
        end if;
    end loop;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- Ti phase at end of cycle
    --
    read <= '0' after Tpd;
end memory_read;
```

Figure7-9 (continued).

```
procedure memory_write (addr : in bit_32;
                        data : in bit_32) is
begin
    -- start bus cycle with address output
    a_bus <= addr after Tpd;
    fetch <= '0' after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T1 phase
    --
    write <= '1' after Tpd;
    wait until phi2 = '1';
    d_bus <= data after Tpd;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- T2 phase
    --
    loop
        wait until phi2 = '0';
        if reset = '1' then
            return;
        end if;
        -- end of T2
        exit when ready = '1';
    end loop;
    wait until phi1 = '1';
    if reset = '1' then
        return;
    end if;
    --
    -- Ti phase at end of cycle
    --
    write <= '0' after Tpd;
    d_bus <= null after Tpd;
end memory_write;
```

Figure7-9 (continued).

The next four procedures, add, subtract, multiply and divide, implement the arithmetic operations on 32-bit words representing twos-complement signed integers. They each take two integer operands, and produce a 32-bit word result and the three condition code flags V (overflow), N (negative) and Z (zero). The result parameter is of mode **inout** because the test for negative and zero results read its value after it has been written. Each procedure is carefully coded to avoid causing an integer overflow on the host machine executing the model (assuming that machine uses 32-bit integers). The add and subtract procedures wrap around if overflow occurs, and multiply and divide return the largest or smallest integer.

Following these procedures is the body of the process which implements the DP32 behavioural model. This process is activated during the initialisation phase of a simulation. It consists of three sections which are repeated sequentially: reset processing, instruction fetch, and instruction execution.

```
procedure add (result : inout bit_32;
                       op1, op2 : in integer;
                       V, N, Z : out bit) is
begin
    if op2 > 0 and op1 > integer'high-op2 then      -- positive overflow
        int_to_bits(((integer'low+op1)+op2)-integer'high-1, result);
        V := '1';
    elsif op2 < 0 and op1 < integer'low-op2 then  -- negative overflow
        int_to_bits(((integer'high+op1)+op2)-integer'low+1, result);
        V := '1';
    else
        int_to_bits(op1 + op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end add;

procedure subtract (result : inout bit_32;
                           op1, op2 : in integer;
                           V, N, Z : out bit) is
begin
    if op2 < 0 and op1 > integer'high+op2 then     -- positive overflow
        int_to_bits(((integer'low+op1)-op2)-integer'high-1, result);
        V := '1';
    elsif op2 > 0 and op1 < integer'low+op2 then  -- negative overflow
        int_to_bits(((integer'high+op1)-op2)-integer'low+1, result);
        V := '1';
    else
        int_to_bits(op1 - op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end subtract;
```

Figure7-9 (continued).

When the reset input is asserted, all of the control ports are returned to their initial states, the data bus driver is disconnected, and the PC register is cleared. The model then waits until reset is negated before proceeding. Throughout the rest of the model, the reset input is checked after each bus transaction. If the transaction was aborted by reset being asserted, no further action is taken in fetching or executing an instruction, and control falls through to the reset handling code.

The instruction fetch part is simply a call to the memory read procedure. The PC register is used to provide the address, the fetch flag is true, and the result is returned into the current instruction register. The PC register is then incremented by one using the arithmetic procedure previously defined.

The fetched instruction is next decoded into its component parts: the op-code, the source and destination register addresses and an immediate constant field. The op-code is then used as the selector for a case statement

```
procedure multiply (result : inout bit_32;
                    op1, op2 : in integer;
                    V, N, Z : out bit) is
begin
    if ((op1>0 and op2>0) or (op1<0 and op2<0))  -- result positive
            and (abs op1 > integer'high / abs op2) then      -- positive overflow
        int_to_bits(integer'high, result);
        V := '1';
    elsif ((op1>0 and op2<0) or (op1<0 and op2>0))   -- result negative
            and ((- abs op1) < integer'low / abs op2) then    -- negative overflow
        int_to_bits(integer'low, result);
        V := '1';
    else
        int_to_bits(op1 * op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end multiply;

procedure divide (result : inout bit_32;
                  op1, op2 : in integer;
                  V, N, Z : out bit) is
begin
    if op2=0 then
        if op1>=0 then                -- positive overflow
            int_to_bits(integer'high, result);
        else
            int_to_bits(integer'low, result);
        end if;
        V := '1';
    else
        int_to_bits(op1 / op2, result);
        V := '0';
    end if;
    N := result(31);
    Z := bool_to_bit(result = X"0000_0000");
end divide;
```

Figure7-9 (continued).

which codes the instruction execution. For the arithmetic instructions (including the quick forms), the arithmetic procedures previously defined are invoked. For the logical instructions, the register bit-vector values are used in VHDL logical expressions to determine the bit-vector result. The condition code Z flag is set if the result is a bit-vector of all '0' bits.

The model executes a load instruction by firstly reading the displacement from memory and incrementing the PC register. The displacement is added to the value of the index register to form the effective address. This is then used in a memory read to load the data into the result register. A quick load is executed similarly, except that no memory read is needed to fetch the displacement; the variable i8 decoded from the instruction is used. The store and quick store instructions parallel the load instructions, with the memory data read being replaced by a memory data write.

Execution of a branch instruction starts with a memory read to fetch the displacement, and an add to increment the PC register by one. The displacement is added to the value of the PC register to form the effective address. Next, the condition expression is evaluated, comparing the condition code bits with the condition mask in the instruction, to determine whether the branch is taken. If it is, the PC register takes on the effective address value. The branch indexed instruction is similar, with the index register value replacing the PC value to form the effective address. The quick branch forms are also similar, with the immediate constant being used for the displacement instead of a value fetched from memory.

```
begin
    --
    -- check for reset active
    --
    if reset = '1' then
        read <= '0' after Tpd;
        write <= '0' after Tpd;
        fetch <= '0' after Tpd;
        d_bus <= null after Tpd;
        PC := X"0000_0000";
        wait until reset = '0';
    end if;
    --
    -- fetch next instruction
    --
    memory_read(PC, true, current_instr);
    if reset /= '1' then
        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
        --
        -- decode & execute
        --
        op := current_instr(31 downto 24);
        r3 := bits_to_natural(current_instr(23 downto 16));
        r1 := bits_to_natural(current_instr(15 downto 8));
        r2 := bits_to_natural(current_instr(7 downto 0));
        i8 := bits_to_int(current_instr(7 downto 0));
```

Figure7-9 (continued).

```
case op is
    when op_add =>
        add(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_addq =>
        add(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_sub =>
        subtract(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_subq =>
        subtract(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_mul =>
        multiply(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_mulq =>
        multiply(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_div =>
        divide(reg(r3), bits_to_int(reg(r1)), bits_to_int(reg(r2)),
                cc_V, cc_N, cc_Z);
    when op_divq =>
        divide(reg(r3), bits_to_int(reg(r1)), i8, cc_V, cc_N, cc_Z);
    when op_land =>
        reg(r3) := reg(r1) and reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_lor =>
        reg(r3) := reg(r1) or reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_lxor =>
        reg(r3) := reg(r1) xor reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_lmask =>
        reg(r3) := reg(r1) and not reg(r2);
        cc_Z := bool_to_bit(reg(r3) = X"0000_0000");
    when op_ld =>
        memory_read(PC, true, displacement);
        if reset /= '1' then
            add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
            add(effective_addr,
                    bits_to_int(reg(r1)), bits_to_int(displacement),
                    temp_V, temp_N, temp_Z);
            memory_read(effective_addr, false, reg(r3));
        end if;
    when op_ldq =>
        add(effective_addr,
                bits_to_int(reg(r1)), i8,
                temp_V, temp_N, temp_Z);
        memory_read(effective_addr, false, reg(r3));
    when op_st =>
        memory_read(PC, true, displacement);
        if reset /= '1' then
            add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
            add(effective_addr,
                    bits_to_int(reg(r1)), bits_to_int(displacement),
                    temp_V, temp_N, temp_Z);
            memory_write(effective_addr, reg(r3));
        end if;
```

Figure7-9 (continued).

```
                when op_stq =>
                    add(effective_addr,
                            bits_to_int(reg(r1)), i8,
                            temp_V, temp_N, temp_Z);
                    memory_write(effective_addr, reg(r3));
                when op_br =>
                    memory_read(PC, true, displacement);
                    if reset /= '1' then
                        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
                        add(effective_addr,
                                bits_to_int(PC), bits_to_int(displacement),
                                temp_V, temp_N, temp_Z);
                        if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                                = cm_i then
                            PC := effective_addr;
                        end if;
                    end if;
                when op_bi =>
                    memory_read(PC, true, displacement);
                    if reset /= '1' then
                        add(PC, bits_to_int(PC), 1, temp_V, temp_N, temp_Z);
                        add(effective_addr,
                                bits_to_int(reg(r1)), bits_to_int(displacement),
                                temp_V, temp_N, temp_Z);
                        if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                                = cm_i then
                            PC := effective_addr;
                        end if;
                    end if;
                when op_brq =>
                    add(effective_addr,
                            bits_to_int(PC), i8,
                            temp_V, temp_N, temp_Z);
                    if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                            = cm_i then
                        PC := effective_addr;
                    end if;
                when op_biq =>
                    add(effective_addr,
                            bits_to_int(reg(r1)), i8,
                            temp_V, temp_N, temp_Z);
                    if ((cm_V and cc_V) or (cm_N and cc_N) or (cm_Z and cc_Z))
                            = cm_i then
                        PC := effective_addr;
                    end if;
                when others =>
                    assert false report "illegal instruction" severity warning;
            end case;
        end if;     -- reset /= '1'
    end process;

end behaviour;
```

Figure7-9 (continued).

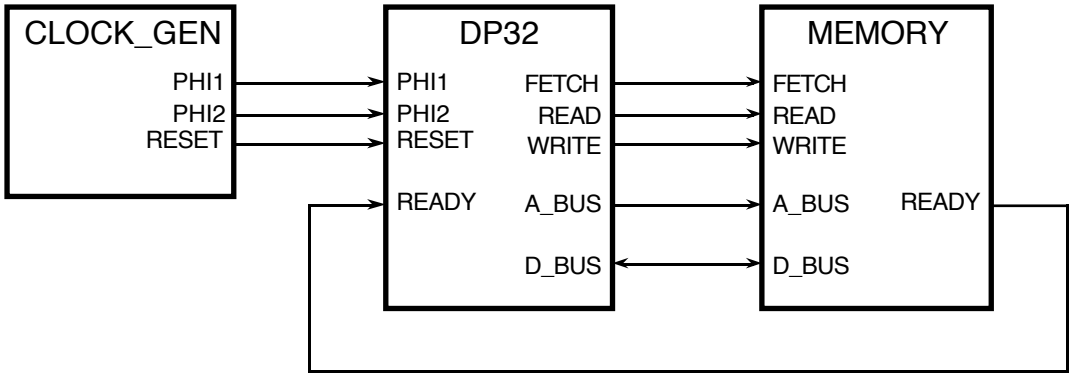


Figure7-10. Test bench circuit for DP32.

```
use work.dp32_types.all;

entity clock_gen is
    generic (Tpw : Time;                -- clock pulse width
             Tps : Time);               -- pulse separation between phases
    port (phi1, phi2 : out bit;
          reset : out bit);
end clock_gen;

architecture behaviour of clock_gen is

    constant clock_period : Time := 2*(Tpw+Tps);

begin

    reset_driver :
        reset <= '1', '0' after 2*clock_period+Tpw;

    clock_driver : process
    begin
        phi1 <=  '1', '0' after Tpw;
        phi2 <= '1' after Tpw+Tps, '0' after Tpw+Tps+Tpw;
        wait for clock_period;
    end process clock_driver;

end behaviour;
```

Figure7-11. Description of clock_gen driver.

7.5. Test Bench

One way of testing the behavioural model of the DP32 processor is to connect it in a test bench circuit, shown in Figure7-10. The clock_gen component generates the two-phase clock and the reset signal to drive the processor. The memory stores a test program and data. We write behavioural models for these two components, and connect them in a structural description of the test bench.

Figure7-11 lists the entity declaration and behavioural architecture of the clock generator. The clock_gen entity has two formal generic constants. Tpw is the pulse width for each of phi1 and phi2, that is, the time for which each clock is '1'. Tps is the pulse separation, that is, the time between one clock signal changing to '0' and the other clock signal changing to '1'.

Based on these values, the clock period is twice the sum of the pulse width and the separation.

The architecture of the clock generator consists of two concurrent statements, one to drive the reset signal and the other to drive the clock signals. The reset driver schedules a '1' value on reset when it is activated at simulation initialisation, followed by a '0' a little after two clock periods later. This concurrent statement is never subsequently reactivated, since its waveform list does not refer to any signals. The clock driver process, when activated, schedules a pulse on phi1 immediately, followed by a pulse on phi2, and then suspends for a clock period. When it resumes, it repeats, scheduling the next clock cycle.

The entity declaration and behavioural architecture of the memory module are shown in Figure7-12. The architecture body consists of one process to implement the behaviour. The process contains an array variable to represent the storage of the memory. When the process is activated, it places the output ports in an initial state: the data bus disconnected and the ready bit negated. It then waits for either a read or write command. When one of these occurs, the address is sampled and converted from a bit-vector to a number. If it is within the address bounds of the memory, the command is acted upon.

For a write command, the ready bit is asserted after a delay representing the write access time of the memory, and then the model waits until the end of the write cycle. At that time, the value on the data bus from a propagation delay beforehand is sampled and written into the memory array. The use of this delayed value models the fact that memory devices actually store the data that was valid a setup-time before the triggering edge of the command bit.

For a read command, the data from the memory array is accessed and placed on the data bus after a delay. This delay represents the read access time of the memory. The ready bit is also asserted after the delay, indicating that the processor may continue. The memory then waits until the end of the read cycle.

At the end of a memory cycle, the process repeats, setting the data bus and ready bit drivers to their initial state, and waiting for the next command.

Figure7-13 shows the entity declaration and structural architecture of the test bench circuit. The entity contains no ports, since there are no external connections to the test bench. The architecture body contains component declarations for the clock driver, the memory and the processor. The ports in these component declarations correspond exactly to those of the entity declarations. There are no formal generic constants, so the actuals for the generics in the entity declarations will be specified in a configuration. The architecture body next declares the signals which are used to connect the components together. These signals may be traced by a simulation monitor when the simulation is run. The concurrent statements of the architecture body consist of the three component instances.

```
use work.dp32_types.all;

entity memory is
    generic (Tpd : Time := unit_delay);
    port (d_bus : inout bus_bit_32 bus;
          a_bus : in bit_32;
          read, write : in bit;
          ready : out bit);
end memory;

architecture behaviour of memory is
begin

    process

        constant low_address : integer := 0;
        constant high_address : integer := 65535;
        type memory_array is
            array (integer range low_address to high_address) of bit_32;
        variable mem : memory_array;
        variable address : integer;

    begin
        --
        -- put d_bus and reply into initial state
        --
        d_bus <= null after Tpd;
        ready <= '0' after Tpd;
        --
        -- wait for a command
        --
        wait until (read = '1') or (write = '1');
        --
        -- dispatch read or write cycle
        --
        address := bits_to_int(a_bus);
        if address >= low_address and address <= high_address then
            -- address match for this memory
            if write = '1' then
                ready <= '1' after Tpd;
                wait until write = '0';              -- wait until end of write cycle
                mem(address) := d_bus'delayed(Tpd);  -- sample data from Tpd ago
            else -- read = '1'
                d_bus <= mem(address) after Tpd;          -- fetch data
                ready <= '1' after Tpd;
                wait until read = '0';                    -- hold for read cycle
            end if;
        end if;
    end process;

end behaviour;
```

Figure7-12. Description of memory module.

```
use work.dp32_types.all;

entity dp32_test is
end dp32_test;

architecture structure of dp32_test is

    component clock_gen
        port (phi1, phi2 : out bit;
              reset : out bit);
    end component;

    component dp32
        port (d_bus : inout bus_bit_32 bus;
              a_bus : out bit_32;
              read, write : out bit;
              fetch : out bit;
              ready : in bit;
              phi1, phi2 : in bit;
              reset : in bit);
    end component;

    component memory
        port (d_bus : inout bus_bit_32 bus;
                  a_bus : in bit_32;
                  read, write : in bit;
                  ready : out bit);
    end component;

    signal d_bus : bus_bit_32 bus;
    signal a_bus : bit_32;
    signal read, write : bit;
    signal fetch : bit;
    signal ready : bit;
    signal phi1, phi2 : bit;
    signal reset : bit;

begin

    cg : clock_gen
        port map (phi1 => phi1, phi2 => phi2, reset => reset);

    proc : dp32
        port map (d_bus => d_bus, a_bus => a_bus,
                  read => read, write => write, fetch => fetch,
                  ready => ready,
                  phi1 => phi1, phi2 => phi2, reset => reset);

    mem : memory
        port map (d_bus => d_bus, a_bus => a_bus,
                  read => read, write => write, ready => ready);

end structure;
```

Figure7-13. Description of test bench circuit.

```
configuration dp32_behaviour_test of dp32_test is
    for structure
        for cg : clock_gen
            use entity work.clock_gen(behaviour)
                generic map (Tpw => 8 ns, Tps => 2 ns);
        end for;
        for mem : memory
            use entity work.memory(behaviour);
        end for;
        for proc : dp32
            use entity work.dp32(behaviour);
        end for;
    end for;
end dp32_behaviour_test;
```

Figure7-14. Configuration of test bench using behaviour of DP32.

Lastly, a configuration for the test bench, using the behavioural description of the DP32 processor, is listed in Figure7-14. The configuration specifies that each of the components in the structure architecture of the test bench should use the behaviour architecture of the corresponding entity. Actual generic constants are specified for the clock generator, giving a clock period of 20ns. The default values for the generic constants of the other entities are used.

In order to run the test bench model, a simulation monitor is invoked and a test program loaded into the array variable in the memory model. The author used the Zycad System VHDL™ simulation system for this purpose. Figure7-15 is an extract from the listing produced by an assembler created for the DP32 processor. The test program initializes R0 to zero (the assembler macro `initr0` generates an `lmask` instruction), and then loops incrementing a counter in memory. The values in parentheses are the instruction addresses, and the hexadecimal values in square brackets are the assembled instructions.

™ Zycad System VHDL is a trademark of Zycad Corporation.

```
 1.                                 include dp32.inc $
 2.
 3.                                 !!!   conventions:
 4.                                 !!!       r0 = 0
 5.                                 !!!       r1 scratch
 6.
 7.                                 begin
 8.   (     0) [07000000         ]    initr0
 9.                                 start:
10.   (     1) [10020000         ]    addq(r2, r0, 0)   ! r2 := 0
11.                                 loop:
12.   (     2) [21020000 00000008]    sta(r2, counter)  ! counter := r2
13.   (     4) [10020201         ]    addq(r2, r2, 1)   ! increment r2
14.   (     5) [1101020A         ]    subq(r1, r2, 10)  ! if r2 = 10 then
15.   (     6) [500900FA         ]    brzq(start)       !   restart
16.   (     7) [500000FA         ]    braq(loop)        ! else next loop
17.
18.                                 counter:
19.   (     8) [00000000         ]    data(0)
20.                                 end
```

Figure7-15. Assembler listing of a test program.

7.6. Register Transfer Architecture

The previous descriptions of the DP32 specified its behaviour without reference to the internal structure of the processor. Such a description is invaluable, as it allows the computer architect to evaluate the instruction set and compare it with alternatives before commiting expensive resources to detailed design and implementation.

Once this abstract architecture has been settled on, the next level of architecture can be designed. Figure7-16 is a block diagram of a simple architecture to implement the DP32 instrcuction set. (Most control signals are not shown.) It consists mainly of a collection of registers and an arithmetic and logic unit (ALU), connected by a number of buses. There are also buffers for interfacing to the processor-memory bus, and a control unit for sequencing operation of the processor.

The software addressable registers are implemented using a three-port register file. Ports1 and2 supply source operands onto the op1 and op2 buses respectively. The address for port2 is normally taken from the r2 field of the current instruction, but a multiplexor is included to allow the r3 field to be used when a store instruction is executed. The op1 and op2 buses

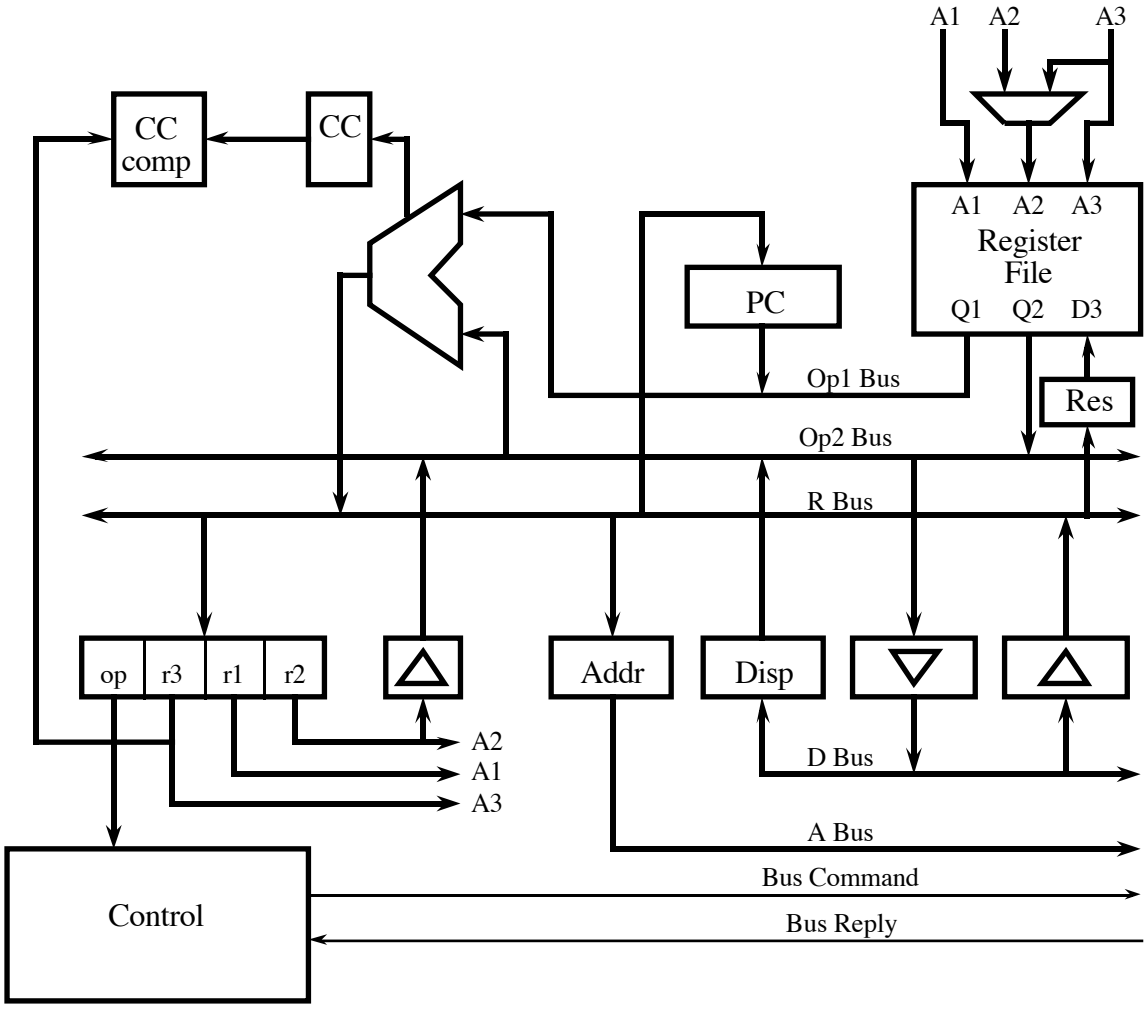


Figure7-16. DP32 data paths block diagram.

are connected to the ALU inputs, and the ALU output drives the result bus. The result can be latched for writing back to the register file using port3. The program counter (PC) register also supplies the op1 bus, and can be loaded from the result bus. The ALU condition flags are latched into the condition code (CC) register, and from there can be compared with the condition mask from the current instruction. The memory bus interface includes an address latch to drive the address bus, a data output buffer driven from the op2 bus, a data input buffer driving the result bus, and a displacement latch driving the op2 bus. An instruction fetched from memory is stored in current instruction register. The r1, r2 and r3 fields are used as register file addresses. The r2 field is also used as an immediate constant and may be sign extended onto the op2 bus. Four bits from the r3 field are used as the condition mask, and the opcode field is used by the control unit.

In this section, descriptions will be given for each of the sub-modules in this architecture, and then they will be used in a structural architecture body of the DP32 entity.

7.6.1. Multiplexor

An entity declaration and architecture body for a 2-input multiplexor is listed in Figure7-17. The entity has a select input bit, two bit-vector inputs i0 and i1, and a bit-vector output y. The size of the bit-vector ports is determined by the generic constant width, which must be specified when the entity is used in a structural description. The architecture body contains a concurrent selected signal assignment, which uses the value of the select input to determine which of the two bit-vector inputs is passed through to the output. The assignment is sensitive to all of the input signals, so when any of them changes, the assignment will be resumed.

7.6.2. Transparent Latch

An entity declaration and architecture body for a latch is listed in Figure7-18. The entity has an enable input bit, a bit-vector input d, and a bit-vector output q. The size of the bit-vector ports is determined by the generic constant width, which must be specified when the entity is used in a structural description. The architecture body contains a process which is

```
use work.dp32_types.all;

entity mux2 is
    generic (width : positive;
                Tpd : Time := unit_delay);
    port (i0, i1 : in bit_vector(width-1 downto 0);
          y : out bit_vector(width-1 downto 0);
          sel : in bit);
end mux2;

architecture behaviour of mux2 is
begin
    with sel select
        y <=    i0 after Tpd when '0',
                i1 after Tpd when '1';
end behaviour;
```

Figure7-17. Description of 2-input multiplexor.

```
use work.dp32_types.all;

entity latch is
    generic (width : positive;
                Tpd : Time := unit_delay);
    port (d : in bit_vector(width-1 downto 0);
            q : out bit_vector(width-1 downto 0);
            en : in bit);
end latch;

architecture behaviour of latch is

begin

    process (d, en)
    begin
        if en = '1' then
            q <= d after Tpd;
        end if;
    end process;

end behaviour;
```

Figure7-18. Description of a transparent latch.

sensitive to the d and en inputs. The behaviour of the latch is such that when en is '1', changes on d are transmitted through to q. However, when en changes to '0', any new value on d is ignored, and the current value on q is maintained. In the model shown in Figure7-18, the latch storage is provided by the output port, in that if no new value is assigned to it, the current value does not change.

7.6.3. Buffer

An entity declaration and architecture body for a buffer is listed in Figure7-19. The entity has an enable input bit en, a bit-vector input a, and a resolved bit-vector bus output b. It is not possible to make this entity generic with respect to input and output port width, because of a limitation imposed by the VHDL language semantics. The output port needs to be a resolved signal, so a bus resolution function is specified in the definition of the port type. This function takes a parameter which is an unconstrained array. In order to make the buffer port width generic, we would need to specify a bus resolution function which took as a parameter an unconstrained array of bit-vector elements whose length is not known. VHDL does not allow the element type of an unconstrained array to be an unconstrained array, so this approach is not possible. For this reason, we define a buffer entity with fixed port widths of 32bits.

The behaviour of the buffer is implemented by a process sensitive to the en and a inputs. If en is '1', the a input is transmitted through to the b output. If en is '0', the driver for b is disconnected, and the value on a is ignored.

```
use work.dp32_types.all;

entity buffer_32 is
    generic (Tpd : Time := unit_delay);
    port (a : in bit_32;
          b : out bus_bit_32 bus;
          en : in bit);
end buffer_32;

architecture behaviour of buffer_32 is

begin

    b_driver: process (en, a)
    begin
        if en = '1' then
            b <= a after Tpd;
        else
            b <= null after Tpd;
        end if;
    end process b_driver;

end behaviour;
```

Figure7-19. Description of a buffer.

```
use work.dp32_types.all;

entity signext_8_32 is
    generic (Tpd : Time := unit_delay);
    port (a : in bit_8;
          b : out bus_bit_32 bus;
          en : in bit);
end signext_8_32;

architecture behaviour of signext_8_32 is

begin

    b_driver: process (en, a)
    begin
        if en = '1' then
            b(7 downto 0) <= a after Tpd;
            if a(7) = '1' then
                b(31 downto 8) <= X"FFFF_FF" after Tpd;
            else
                b(31 downto 8) <= X"0000_00" after Tpd;
            end if;
        else
            b <= null after Tpd;
        end if;
    end process b_driver;

end behaviour;
```

Figure7-20. Description of the sign extending buffer.

7.6.4. Sign Extending Buffer

The sign-extending buffer shown in Figure7-20 is almost identical to the plain buffer, except that it has an 8-bit input. This input is treated as a twos-complement signed integer, and the output is the same integer, but extended to 32bits. The extension is achieved by replicating the sign bit into bits8 to31 of the output.

7.6.5. Latching Buffer

Figure7-21 lists an entity declaration an architecture body for a latching buffer. This model is a combination of those for the plain latch and buffer. When latch_en is '1', changes on d are stored in the latch, and may be transmitted through to q. However, when latch_en changes to '0', any new value on d is ignored, and the currently stored value is maintained. The out_en input controls whether the stored value is tranmitted to the output. Unlike the plain latch, explicit storage must be provided (in the form of the variable latched_value), since the output driver may be disconnected when a new value is to be stored.

7.6.6. Program Counter Register

The entity declaration and architecture body of the PC register are listed in Figure7-22. The PC register is a master/slave type register, which can be reset to all zeros by asserting the reset input. When reset is negated, the latch operates normally. With latch_en at '1', the value of the d input is stored in the variable master_PC, but the output (if enabled) is driven from the previously stored value in slave_PC. Then when latch_en changes from

```
use work.dp32_types.all;

entity latch_buffer_32 is
    generic (Tpd : Time := unit_delay);
    port (d : in bit_32;
          q : out bus_bit_32 bus;
          latch_en : in bit;
          out_en : in bit);
end latch_buffer_32;

architecture behaviour of latch_buffer_32 is

begin

    process (d, latch_en, out_en)
        variable latched_value : bit_32;
    begin
        if latch_en = '1' then
            latched_value := d;
        end if;
        if out_en = '1' then
            q <= latched_value after Tpd;
        else
            q <= null after Tpd;
        end if;
    end process;

end behaviour;
```

Figure7-21. Description of a latching buffer.

```
use work.dp32_types.all;

entity PC_reg is
    generic (Tpd : Time := unit_delay);
    port (d : in bit_32;
          q : out bus_bit_32 bus;
          latch_en : in bit;
          out_en : in bit;
          reset : in bit);
end PC_reg;

architecture behaviour of PC_reg is

begin

    process (d, latch_en, out_en, reset)
        variable master_PC, slave_PC : bit_32;
    begin
        if reset = '1' then
            slave_PC := X"0000_0000";
        elsif latch_en = '1' then
            master_PC := d;
        else
            slave_PC := master_PC;
        end if;
        if out_en = '1' then
            q <= slave_PC after Tpd;
        else
            q <= null after Tpd;
        end if;
    end process;

end behaviour;
```

Figure7-22. Description of the PC register.

'1' to '0', the slave value is update from the master value, and any subsequent changes in the d input are ignored. This behaviour means that the PC register output can be used to derive a new value, and the new value written back at the same time. If an ordinary transparent latch were used, a race condition would be created, since the new value would be transmitted through to the output in place of the old value, affecting the calculation of the new value.

7.6.7. Register File

Figure7-23 lists the description of the 3-port register file, with two read ports and one write port. Each port has an address input (a1, a2 and a3) and an enable input (en1, en2 and en3). The read ports have data bus outputs (q1 and q2), and the write port has a data input (d3). The number bits in the port addresses is determined by the generic constant depth. The behaviour of the entity is implemented by the process reg_file. It declares a numeric type used to index the register file, and an array for the register file storage. When any of the inputs change, firstly the write port enable is checked, and if asserted, the addressed register is updated. Then each of the read port enables is checked. If asserted, the addressed data is fetched and driven onto the corresponding data output bus. If the port is disabled, the data output bus driver is disconnected.

```
use work.dp32_types.all;

entity reg_file_32_rrw is
    generic (depth : positive;                  -- number of address bits
                Tpd : Time := unit_delay;
                Tac : Time := unit_delay);
    port (a1 : in bit_vector(depth-1 downto 0);
            q1 : out bus_bit_32 bus;
            en1 : in bit;
            a2 : in bit_vector(depth-1 downto 0);
            q2 : out bus_bit_32 bus;
            en2 : in bit;
            a3 : in bit_vector(depth-1 downto 0);
            d3 : in bit_32;
            en3 : in bit);
end reg_file_32_rrw;

architecture behaviour of reg_file_32_rrw is

begin

    reg_file: process (a1, en1, a2, en2, a3, d3, en3)

        subtype reg_addr is natural range 0 to depth-1;
        type register_array is array (reg_addr) of bit_32;

        variable registers : register_array;

    begin
        if en3 = '1' then
            registers(bits_to_natural(a3)) := d3;
        end if;
        if en1 = '1' then
            q1 <= registers(bits_to_natural(a1)) after Tac;
        else
            q1 <= null after Tpd;
        end if;
        if en2 = '1' then
            q2 <= registers(bits_to_natural(a2)) after Tac;
        else
            q2 <= null after Tpd;
        end if;
    end process reg_file;

end behaviour;
```

Figure7-23. Description of the 3-port register file.

7.6.8. Arithmetic & Logic Unit

The description of the ALU is listed in Figure7-24. The package ALU_32_types defines an enumerated type for specifying the ALU function. This must be placed in a package, since it is required for both the ALU description and for entities that make use of the ALU. There is no corresponding package body, since the type is fully defined in the package specification.

The ALU entity declaration uses the ALU_32_types package as well as the general dp32_types package. It has two operand input ports, a result output and condition code output ports, and a command input port. This last port is an example of a port which is of an enumerated type, since at this stage

of design, no encoding is known or specified for the ALU function command.

The ALU behaviour is implemented by the process ALU_function, sensitive to changes on the operand and command input ports. If the command to be performed is an arithmetic operation, the model firstly converts the operands to integers. This is followed by a case statement dispatching on the command. For the disable command, no operation is performed, and for the pass1 command, the result is operand1 unchanged. The result for logic commands is derived by applying the corresponding VHDL logical operations to the bit-vector operands. For arithmetic commands the result is computed the same was as it was in the behavioural model of the DP32 presented in Section7.4. Also, the overflow condition code bit (cc_V), which is only defined for arithmetic operations, is assigned here. Finally, the result and remaining condition code bits are assigned. The result output is only driven if the command is not disable, otherwise it is disconnected.

```
package ALU_32_types is

    type ALU_command is (disable, pass1, incr1,
                                add, subtract, multiply, divide,
                                log_and, log_or, log_xor, log_mask);

end ALU_32_types;


use work.dp32_types.all, work.ALU_32_types.all;

entity ALU_32 is
    generic (Tpd : Time := unit_delay);
    port (operand1 : in bit_32;
          operand2 : in bit_32;
          result : out bus_bit_32 bus;
          cond_code : out CC_bits;
          command : in ALU_command);
end ALU_32;
```

Figure7-24. Description of the Arithmetic and Logic Unit.

```
architecture behaviour of ALU_32 is

    alias cc_V : bit is cond_code(2);
    alias cc_N : bit is cond_code(1);
    alias cc_Z : bit is cond_code(0);

begin

    ALU_function: process (operand1, operand2, command)

        variable a, b : integer;
        variable temp_result : bit_32;

    begin
        case command is
            when add | subtract | multiply | divide =>
                a := bits_to_int(operand1);
                b := bits_to_int(operand2);
            when incr1 =>
                a := bits_to_int(operand1);
                b := 1;
            when others =>
                null;
        end case;
        case command is
            when disable =>
                null;
            when pass1 =>
                temp_result := operand1;
            when log_and =>
                 temp_result := operand1 and operand2;
            when log_or =>
                temp_result := operand1 or operand2;
            when log_xor =>
                temp_result := operand1 xor operand2;
            when log_mask =>
                temp_result := operand1 and not operand2;
            when add | incr1 =>
                if b > 0 and a > integer'high-b then       -- positive overflow
                    int_to_bits(((integer'low+a)+b)-integer'high-1, temp_result);
                    cc_V <= '1' after Tpd;
                elsif b < 0 and a < integer'low-b then    -- negative overflow
                     int_to_bits(((integer'high+a)+b)-integer'low+1, temp_result);
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a + b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
            when subtract =>
                if b < 0 and a > integer'high+b then      -- positive overflow
                    int_to_bits(((integer'low+a)-b)-integer'high-1, temp_result);
                    cc_V <= '1' after Tpd;
                elsif b > 0 and a < integer'low+b then  -- negative overflow
                    int_to_bits(((integer'high+a)-b)-integer'low+1, temp_result);
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a - b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
```

Figure7-24 (continued).

```
            when multiply =>
                if ((a>0 and b>0) or (a<0 and b<0))       -- result positive
                        and (abs a > integer'high / abs b) then
                    -- positive overflow
                    int_to_bits(integer'high, temp_result);
                    cc_V <= '1' after Tpd;
                elsif ((a>0 and b<0) or (a<0 and b>0))       -- result negative
                        and ((- abs a) < integer'low / abs b) then
                    -- negative overflow
                    int_to_bits(integer'low, temp_result);
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a * b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
            when divide =>
                if b=0 then
                    if a>=0 then                              -- positive overflow
                        int_to_bits(integer'high, temp_result);
                    else
                        int_to_bits(integer'low, temp_result);
                    end if;
                    cc_V <= '1' after Tpd;
                else
                    int_to_bits(a / b, temp_result);
                    cc_V <= '0' after Tpd;
                end if;
        end case;
        if command /= disable then
            result <= temp_result after Tpd;
        else
            result <= null after Tpd;
        end if;
        cc_Z <= bool_to_bit(temp_result = X"00000000") after Tpd;
        cc_N <= bool_to_bit(temp_result(31) = '1') after Tpd;
    end process ALU_function;

end behaviour;
```

Figure7-24 (continued).

```
use work.dp32_types.all;

entity cond_code_comparator is
    generic (Tpd : Time := unit_delay);
    port (cc : in CC_bits;
          cm : in cm_bits;
          result : out bit);
end cond_code_comparator;

architecture behaviour of cond_code_comparator is
    alias cc_V : bit is cc(2);
    alias cc_N : bit is cc(1);
    alias cc_Z : bit is cc(0);
    alias cm_i : bit is cm(3);
    alias cm_V : bit is cm(2);
    alias cm_N : bit is cm(1);
    alias cm_Z : bit is cm(0);
begin
    result <= bool_to_bit(((cm_V and cc_V)
                           or (cm_N and cc_N)
                           or (cm_Z and cc_Z)) = cm_i) after Tpd;
end behaviour;
```

Figure7-25. Description of the condition code comparator.

7.6.9. Condition Code Comparator

The description of the condition code comparator is listed in Figure7-25. The cc input port contains the three condition code bits V, N and Z, and the cm input contains the four condition mask bits derived from a DP32 instruction. Aliases for each of these bits are declared in the architecture body. The behaviour is implemented by a single concurrent signal assignment statement, which is sensitive to all of the input bits. Whenever any of the bits changes value, the assignment will be resumed and a new result bit computed.

7.6.10. Structural Architecture of the DP32

In this section, a structural architecture body for the DP32 processor, corresponding to Figure7-16, will be described. See Figure7-26 for a listing of the architecture body.

```
use work.dp32_types.all, work.ALU_32_types.all;

architecture RTL of dp32 is

    component reg_file_32_rrw
        generic (depth : positive);
        port (a1 : in bit_vector(depth-1 downto 0);
                q1 : out bus_bit_32 bus;
                en1 : in bit;
                a2 : in bit_vector(depth-1 downto 0);
                q2 : out bus_bit_32 bus;
                en2 : in bit;
                a3 : in bit_vector(depth-1 downto 0);
                d3 : in bit_32;
                en3 : in bit);
    end component;

    component mux2
        generic (width : positive);
        port (i0, i1 : in bit_vector(width-1 downto 0);
                y : out bit_vector(width-1 downto 0);
                sel : in bit);
    end component;

    component PC_reg
        port (d : in bit_32;
                q : out bus_bit_32 bus;
                latch_en : in bit;
                out_en : in bit;
                reset : in bit);
    end component;

    component ALU_32
        port (operand1 : in bit_32;
                operand2 : in bit_32;
                result : out bus_bit_32 bus;
                cond_code : out CC_bits;
                command : in ALU_command);
    end component;

    component cond_code_comparator
        port (cc : in CC_bits;
                cm : in cm_bits;
                result : out bit);
    end component;

    component buffer_32
        port (a : in bit_32;
                b : out bus_bit_32 bus;
                en : in bit);
    end component;

    component latch
        generic (width : positive);
        port (d : in bit_vector(width-1 downto 0);
                q : out bit_vector(width-1 downto 0);
                en : in bit);
    end component;
```

Figure7-26. Structural description of the DP32 processor.

```
    component latch_buffer_32
        port (d : in bit_32;
              q : out bus_bit_32 bus;
              latch_en : in bit;
              out_en : in bit);
    end component;

    component signext_8_32
        port (a : in bit_8;
              b : out bus_bit_32 bus;
              en : in bit);
    end component;

    signal op1_bus : bus_bit_32;
    signal op2_bus : bus_bit_32;
    signal r_bus : bus_bit_32;

    signal ALU_CC : CC_bits;
    signal CC : CC_bits;

    signal current_instr : bit_32;
    alias instr_a1 : bit_8 is current_instr(15 downto 8);
    alias instr_a2 : bit_8 is current_instr(7 downto 0);
    alias instr_a3 : bit_8 is current_instr(23 downto 16);
    alias instr_op : bit_8 is current_instr(31 downto 24);
    alias instr_cm : cm_bits is current_instr(19 downto 16);

    signal reg_a2 : bit_8;
    signal reg_result : bit_32;

    signal addr_latch_en : bit;
    signal disp_latch_en : bit;
    signal disp_out_en : bit;
    signal d2_en : bit;
    signal dr_en : bit;
    signal instr_latch_en : bit;
    signal immed_signext_en : bit;
    signal ALU_op : ALU_command;
    signal CC_latch_en : bit;
    signal CC_comp_result : bit;
    signal PC_latch_en : bit;
    signal PC_out_en : bit;
    signal reg_port1_en : bit;
    signal reg_port2_en : bit;
    signal reg_port3_en : bit;
    signal reg_port2_mux_sel : bit;
    signal reg_res_latch_en : bit;

begin -- architecture RTL of dp32

    reg_file : reg_file_32_RRW
        generic map (depth => 8)
        port map (a1 => instr_a1, q1 => op1_bus, en1 => reg_port1_en,
                  a2 => reg_a2, q2 => op2_bus, en2 => reg_port2_en,
                  a3 => instr_a3, d3 => reg_result, en3 => reg_port3_en);

    reg_port2_mux : mux2
        generic map (width => 8)
        port map (i0 => instr_a2, i1 => instr_a3, y => reg_a2,
                  sel => reg_port2_mux_sel);
```

Figure7-26 (continued).

The architecture refers to the items declared in the packages dp32_types and ALU_32_types, so a use clause for these packages is included. The declaration section of the architecture contains a number of component declarations, corresponding to the entity declarations listed in Sections7.6.1 to7.6.9. Instances of these components are subsequently used to construct the processor architecture.

Next, a number of signals are declared, corresponding to the buses illustrated in Figure7-16. These are followed by further signal declarations for control signals not shown in the figure. The control signals are used to connect the data path component instances with the control unit implemented in the block called controller.

```
reg_res_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => reg_result, en => reg_res_latch_en);

PC : PC_reg
    port map (d => r_bus, q => op1_bus,
              latch_en => PC_latch_en, out_en => PC_out_en,
              reset => reset);

ALU : ALU_32
    port map (operand1 => op1_bus, operand2 => op2_bus,
              result => r_bus, cond_code => ALU_CC,
              command => ALU_op);

CC_reg : latch
    generic map (width => 3)
    port map (d => ALU_CC, q => CC, en  => CC_latch_en);

CC_comp : cond_code_comparator
    port map (cc => CC, cm => instr_cm, result => CC_comp_result);

dr_buffer : buffer_32
    port map (a => d_bus, b => r_bus, en => dr_en);

d2_buffer : buffer_32
    port map (a => op2_bus, b => d_bus, en => d2_en);

disp_latch : latch_buffer_32
    port map (d => d_bus, q => op2_bus,
              latch_en => disp_latch_en, out_en => disp_out_en);

addr_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => a_bus, en => addr_latch_en);

instr_latch : latch
    generic map (width => 32)
    port map (d => r_bus, q => current_instr, en => instr_latch_en);

immed_signext : signext_8_32
    port map (a => instr_a2, b => op2_bus, en => immed_signext_en);
```

Figure7-26 (continued).

```
controller : block

    port (phi1, phi2 : in bit;
          reset : in bit;
          opcode : in bit_8;
          read, write, fetch : out bit;
          ready : in bit;
          addr_latch_en : out bit;
          disp_latch_en : out bit;
          disp_out_en : out bit;
          d2_en : out bit;
          dr_en : out bit;
          instr_latch_en : out bit;
          immed_signext_en : out bit;
          ALU_op : out ALU_command;
          CC_latch_en : out bit;
          CC_comp_result : in bit;
          PC_latch_en : out bit;
          PC_out_en : out bit;
          reg_port1_en : out bit;
          reg_port2_en : out bit;
          reg_port3_en : out bit;
          reg_port2_mux_sel : out bit;
          reg_res_latch_en : out bit);

    port map (phi1 => phi1, phi2 => phi2,
              reset => reset,
              opcode => instr_op,
              read => read, write => write, fetch => fetch,
              ready => ready,
              addr_latch_en => addr_latch_en,
              disp_latch_en => disp_latch_en,
              disp_out_en => disp_out_en,
              d2_en => d2_en,
              dr_en => dr_en,
              instr_latch_en => instr_latch_en,
              immed_signext_en => immed_signext_en,
              ALU_op => ALU_op,
              CC_latch_en => CC_latch_en,
              CC_comp_result => CC_comp_result,
              PC_latch_en => PC_latch_en, PC_out_en => PC_out_en,
              reg_port1_en => reg_port1_en,
              reg_port2_en => reg_port2_en,
              reg_port3_en => reg_port3_en,
              reg_port2_mux_sel => reg_port2_mux_sel,
              reg_res_latch_en => reg_res_latch_en);
```

Figure7-26 (continued).

The control unit is a state machine, whose behaviour is described by a single process called state_machine. The controller sequences through the states listed in the declaration of the type controller_state to fetch, decode and execute instructions. The variable state holds the controller state for the current clock cycle, and next_state is set to determine the state for the next clock cycle. Write_back_pending is a flag used to schedule a register write operation for the next clock cycle. The constant ALU_op_select is a lookup table used to determine the ALU function from the instruction op-code.

Figure7-26 (continued).

```
begin -- block controller
    state_machine: process
        type controller_state is
            (resetting, fetch_0, fetch_1, fetch_2, decode,
                disp_fetch_0, disp_fetch_1, disp_fetch_2,
                execute_0, execute_1, execute_2);
        variable state, next_state : controller_state;
        variable write_back_pending : boolean;
        type ALU_op_select_table is
            array (natural range 0 to 255) of ALU_command;
        constant ALU_op_select : ALU_op_select_table :=
            (16#00# => add,
                16#01# => subtract,
                16#02# => multiply,
                16#03# => divide,
                16#10# => add,
                16#11# => subtract,
                16#12# => multiply,
                16#13# => divide,
                16#04# => log_and,
                16#05# => log_or,
                16#06# => log_xor,
                16#07# => log_mask,
                others => disable);
```

The body of the state machine process starts by waiting for the leading edge of the phi1 clock, indicating the start of a clock cycle. When this occurs, the reset signal is checked, and if it is asserted the controller state is set to resetting and all control outputs are negated. On the other hand, if reset is negated, the controller state is updated to the previously computed next state.

```
    begin -- process state_machine
        --
        -- start of clock cycle
        --
        wait until phi1 = '1';
        --
        -- check for reset
        --
        if reset = '1' then
            state := resetting;
            --
            -- reset external bus signals
            --
            read <= '0' after Tpd;
            fetch <= '0' after Tpd;
            write <= '0' after Tpd;
            --
            -- reset dp32 internal control signals
            --
            addr_latch_en <= '0' after Tpd;
            disp_latch_en <= '0' after Tpd;
            disp_out_en <= '0' after Tpd;
            d2_en <= '0' after Tpd;
            dr_en <= '0' after Tpd;
            instr_latch_en <= '0' after Tpd;
            immed_signext_en <= '0' after Tpd;
            ALU_op <= disable after Tpd;
            CC_latch_en <= '0' after Tpd;
            PC_latch_en <= '0' after Tpd;
            PC_out_en <= '0' after Tpd;
            reg_port1_en <= '0' after Tpd;
            reg_port2_en <= '0' after Tpd;
            reg_port3_en <= '0' after Tpd;
            reg_port2_mux_sel <= '0' after Tpd;
            reg_res_latch_en <= '0' after Tpd;
            --
            -- clear write-back flag
            --
            write_back_pending := false;
            --
        else -- reset = '0'
            state := next_state;
        end if;
```

Figure7-26 (continued).

The remainder of the state machine body is a case statement using the current state to determine the action to be performed for this clock cycle. If the processor is being reset (in the resetting state), it waits until the trailing edge of phi2 at the end of the clock cycle, and checks the reset signal again. If reset has been negated, the processor can start fetching instructions, so the next state is set to fetch_0, otherwise it is is set to resetting again.

```
--
-- dispatch action for current state
--
case state is
    when resetting =>
        --
        -- check for reset going inactive at end of clock cycle
        --
        wait until phi2 = '0';
        if reset = '0' then
            next_state := fetch_0;
        else
            next_state := resetting;
        end if;
        --
    when fetch_0 =>
        --
        -- clean up after previous execute cycles
        --
        reg_port1_en <= '0' after Tpd;
        reg_port2_mux_sel <= '0' after Tpd;
        reg_port2_en <= '0' after Tpd;
        immed_signext_en <= '0' after Tpd;
        disp_out_en <= '0' after Tpd;
        dr_en <= '0' after Tpd;
        read <= '0' after Tpd;
        d2_en <= '0' after Tpd;
        write <= '0' after Tpd;
        --
        -- handle pending register write-back
        --
        if write_back_pending then
            reg_port3_en <= '1' after Tpd;
        end if;
        --
        -- enable PC via ALU to address latch
        --
        PC_out_en <= '1' after Tpd;          -- enable PC onto op1_bus
        ALU_op <= pass1 after Tpd;           -- pass PC to r_bus
        --
        wait until phi2 = '1';
        addr_latch_en <= '1' after Tpd;      -- latch instr address
        wait until phi2 = '0';
        addr_latch_en <= '0' after Tpd;
        --
        next_state := fetch_1;
        --
```

Figure7-26 (continued).

The processor fetches an instruction from memory by sequencing through the states fetch_0, fetch_1 and fetch_2 on successive clock cycles. Figure7-27 shows the timing of control signals for an instruction fetch. The fetch_0 processor cycle corresponds to a Ti cycle on the memory bus. During this cycle, the PC register output is enabled onto the op1 bus, and the ALU function set to pass1. The ALU passes the PC value through to the result bus, and it is latched into the memory address register during the second half of the cycle. The PC value is thus set up on the memory address bus. The fetch_1 cycle corresponds to a memory bus T1 cycle. The controller starts the memory transaction by asserting fetch and read. At the same time, it changes the ALU function code to incr1, causing the ALU to place

```
when fetch_1 =>
    --
    -- clear pending register write-back
    --
    if write_back_pending then
        reg_port3_en <= '0' after Tpd;
        write_back_pending := false;
    end if;
    --
    -- increment PC & start bus read
    --
    ALU_op <= incr1 after Tpd;          -- increment PC onto r_bus
    fetch <= '1' after Tpd;
    read <= '1' after Tpd;
    --
    wait until phi2 = '1';
    PC_latch_en <= '1' after Tpd;       -- latch incremented PC
    wait until phi2 = '0';
    PC_latch_en <= '0' after Tpd;
    --
    next_state := fetch_2;
    --
when fetch_2 =>
    --
    -- cleanup after previous fetch_1
    --
    PC_out_en <= '0' after Tpd;         -- disable PC from op1_bus
    ALU_op <= disable after Tpd;        -- disable ALU from r_bus
                --
    -- latch current instruction
    --
    dr_en <= '1' after Tpd;             -- enable fetched instr onto r_bus
    --
    wait until phi2 = '1';
    instr_latch_en <= '1' after Tpd;  -- latch fetched instr from r_bus
    wait until phi2 = '0';
    instr_latch_en <= '0' after Tpd;
    --
    if ready = '1' then
        next_state := decode;
    else
        next_state := fetch_2;          -- extend bus read
    end if;
```

Figure7-26 (continued).

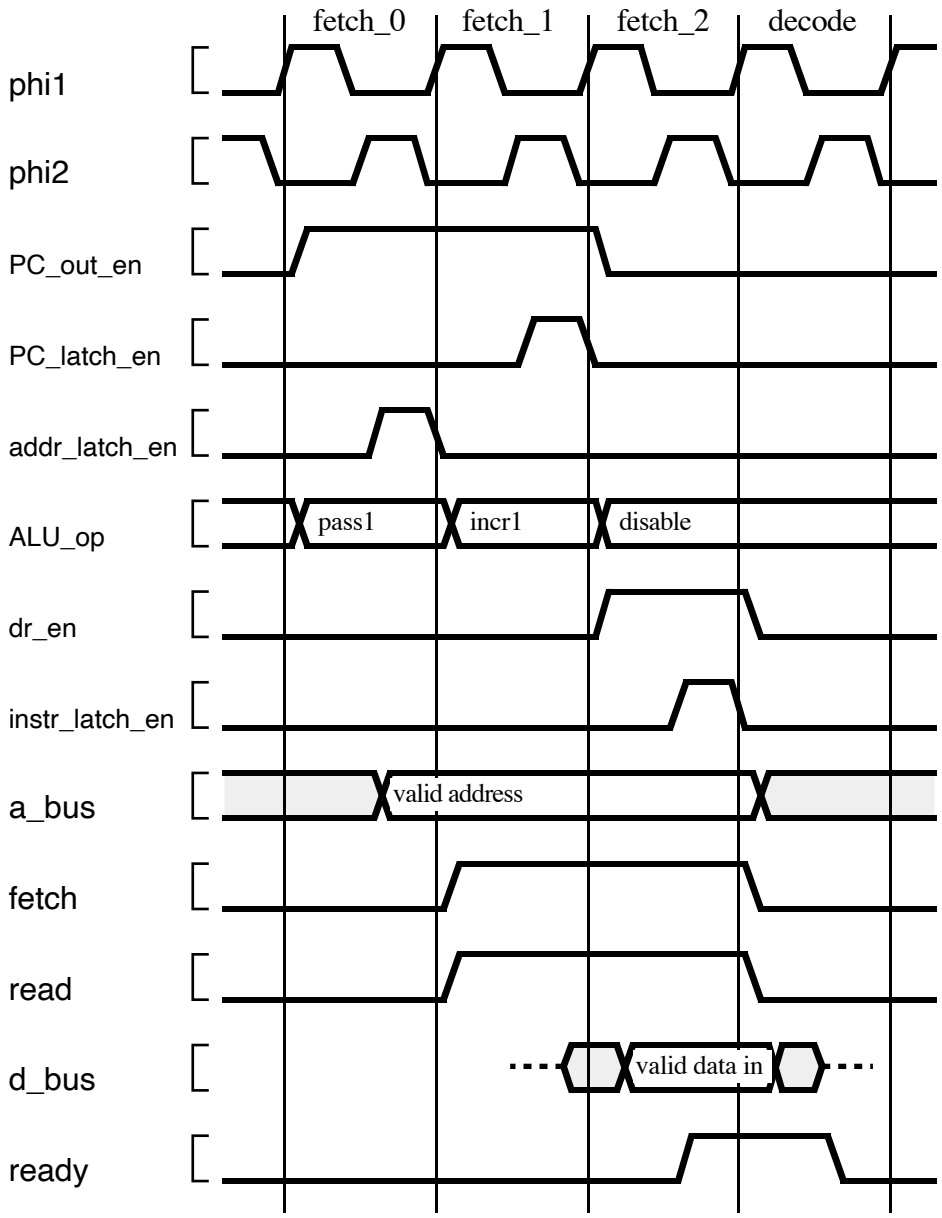


Figure7-27. Timing for DP32 instruction fetch.

the incremented PC value on the result bus. This is then latched back into the PC register during the second half of the cycle. The fetch_2 processor cycle corresponds to the memory bus T2 cycle, during which data is returned to the processor from the memory. The controller disables the PC from the op1 bus and the ALU from the result bus, and enables the data input buffer to accept memory data onto the result bus. This data is latched into the current instruction register during the second half of the cycle. If ready is false, the processor repeats the F2 cycle, otherwise it completes the bus transaction and moves to the decode state, corresponding to a bus Ti cycle.

Returning to the VHDL description, we see that the fetch_0 branch of the case statement implements the first cycle of an instruction fetch. Firstly, any signals left asserted from previous cycle are negated again. Next, any register write scheduled from the previously executed instruction is

handled. (This will be described fully below.) Then the PC register output is enabled and the ALU function set, as described above. The process then waits until the leading edge of phi2, by which time the PC should be valid on the result bus. It pulses the address latch enable signal by asserting it, waiting until the trailing edge of phi2, then negating the signal. Finally, the next state variable is set to fetch_1, so that when the process resumes in the next cycle, it will move to this state.

When the process is in state fetch_1, it starts the cycle by terminating any register write back that may have been pending. It then changes the ALU function code to increment the PC value, and starts the bus transaction. In the second half of the cycle, when phi2 is asserted, the PC latch enable is asserted to store the incremented PC value. The next state is then set to

```
when decode =>
    --
    -- terminate bus read from previous fetch_2
    --
    fetch <= '0' after Tpd;
    read <= '0' after Tpd;
    dr_en <= '0' after Tpd;         -- disable fetched instr from r_bus
    --
    -- delay to allow decode logic to settle
    --
    wait until phi2 = '0';
    --
    -- next state based on opcode of currect instruction
    --
    case opcode is
        when op_add | op_sub | op_mul | op_div
                | op_addq | op_subq | op_mulq | op_divq
                | op_land | op_lor | op_lxor | op_lmask
                | op_ldq | op_stq =>
            next_state := execute_0;
        when op_ld | op_st =>
            next_state := disp_fetch_0;      -- fetch offset
        when op_br | op_bi =>
            if CC_comp_result = '1' then     -- if branch taken
                next_state := disp_fetch_0; --    fetch displacement
            else                             -- else
                next_state := execute_0;     --    increment PC
                                             --    past displacement
            end if;
        when op_brq | op_biq =>
            if CC_comp_result = '1' then     -- if branch taken
                next_state := execute_0;     --    add immed
                                             --        displacement to PC
            else                             -- else
                next_state := fetch_0;       --    no action needed
            end if;
        when others =>
            assert false report "illegal instruction" severity warning;
            next_state := fetch_0;           -- ignore and carry on
    end case; -- op
    --
```

Figure7-26 (continued).

fetch_2.

The last cycle of the instruction fetch is state fetch_2. The controller disables the PC register and ALU outputs, and enables the buffer between the memory data bus and the result bus. During the second half of the cycle, it asserts the instruction register latch enable. At the end of the cycle, when phi2 has returned to '0', the ready input is checked. If it is asserted, the state machine can continue to the decode state in the next cycle, otherwise the fetch_2 state must be repeated.

In the decode state, the controller terminates the previous bus transaction and disables the bus input buffer. It then delays for the rest of the cycle, modeling the time required for decode logic to analyse the current instruction and for the condition code comparator to stabilize. The op-code part of the instruction is then examined to determine the next state. For arithmetic, logical and quick load/store instructions, the next state is execute_0, in which the instruction is interpreted. For load/store instructions with a long displacement, a bus transaction must be performed to read the displacement, so the next state is disp_fetch_0. For branch instructions with a long displacement, the fetch is only required if the branch is to be taken, in which case the next state is disp_fetch_0. Otherwise the next state is execute_0, in which the PC will be incremented past the displacement stored in memory. For branch quick instructions, the displacement is encoded in the instruction. If the branch is taken, the next state is execute_0 to update the PC. Otherwise no further action is needed to interpret the instruction, so the next state is fetch_0. If any other op-code is detected, an assertion is used to report the illegal instruction. The instruction is ignored and execution continues with the next instruction, so the next state is fetch_0.

```
when disp_fetch_0 =>
    --
    -- enable PC via ALU to address latch
    --
    PC_out_en <= '1' after Tpd;          -- enable PC onto op1_bus
    ALU_op <= pass1 after Tpd;           -- pass PC to r_bus
    --
    wait until phi2 = '1';
    addr_latch_en <= '1' after Tpd;      -- latch displacement address
    wait until phi2 = '0';
    addr_latch_en <= '0' after Tpd;
    --
    next_state := disp_fetch_1;
    --
when disp_fetch_1 =>
    --
    -- increment PC & start bus read
    --
    ALU_op <= incr1 after Tpd;           -- increment PC onto r_bus
    fetch <= '1' after Tpd;
    read <= '1' after Tpd;
    --
    wait until phi2 = '1';
    PC_latch_en <= '1' after Tpd;        -- latch incremented PC
    wait until phi2 = '0';
    PC_latch_en <= '0' after Tpd;
    --
    next_state := disp_fetch_2;
    --
when disp_fetch_2 =>
    --
    -- cleanup after previous disp_fetch_1
    --
    PC_out_en <= '0' after Tpd;          -- disable PC from op1_bus
    ALU_op <= disable after Tpd;         -- disable ALU from r_bus
    --
    -- latch displacement
    --
    wait until phi2 = '1';
    disp_latch_en <= '1' after Tpd;  -- latch fetched disp from r_bus
    wait until phi2 = '0';
    disp_latch_en <= '0' after Tpd;
    --
    if ready = '1' then
        next_state := execute_0;
    else
        next_state := disp_fetch_2;      -- extend bus read
    end if;
```

Figure7-26 (continued).

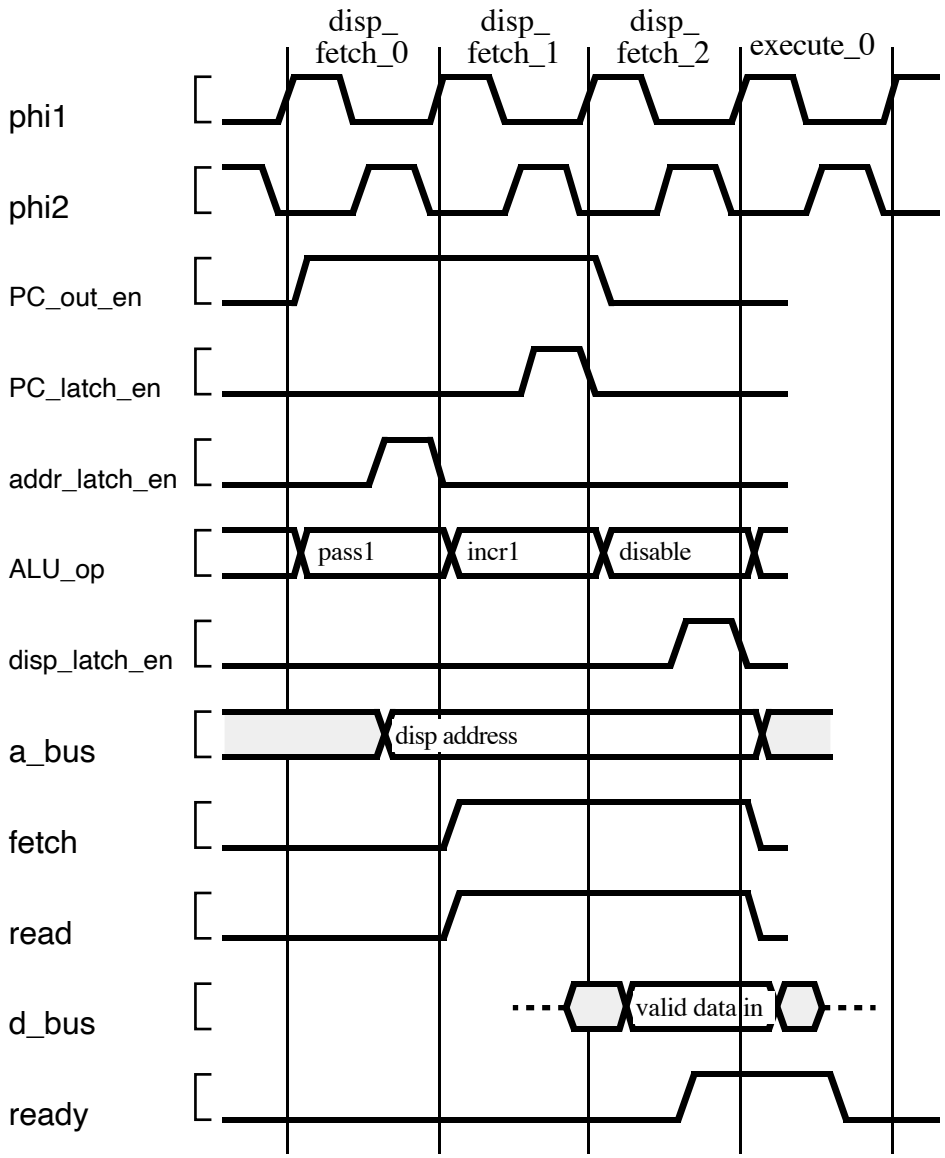


Figure7-28. Timing for DP32 displacement fetch.

The sequence for fetching a displacement from memory is similar to that for fetching the instruction word. The only difference is that instead of the read word being enabled onto the result bus and latched into the instruction register, the word is simply latched from the memory data bus into the displacement latch. The timing for a displacement fetch is shown in Figure7-28. The sequence consists of the processor states disp_fetch_0, disp_fetch_1 and one or more repetitions of disp_fetch_2, corresponding to bus states Ti, T1 and T2 respectively. This sequence is always followed by the first execute state, corresponding to the bus Ti state at the end of the bus transaction. In the VHDL description, the case branches for disp_fetch_0, disp_fetch_1 and disp_fetch_2 implement this behaviour.

```
when execute_0 =>
    --
    -- terminate bus read from previous disp_fetch_2
    --
    fetch <= '0' after Tpd;
    read <= '0' after Tpd;
    --
    case opcode is
        when op_add | op_sub | op_mul | op_div
                | op_addq | op_subq | op_mulq | op_divq
                | op_land | op_lor | op_lxor | op_lmask =>
            -- enable r1 onto op1_bus
            reg_port1_en <= '1' after Tpd;
            if opcode = op_addq or opcode = op_subq
                    or opcode = op_mulq or opcode = op_divq then
                -- enable i8 onto op2_bus
                immed_signext_en <= '1' after Tpd;
            else
                -- select a2 as port2 address
                reg_port2_mux_sel <= '0' after Tpd;
                -- enable r2 onto op2_bus
                reg_port2_en <= '1' after Tpd;
            end if;
            -- select ALU operation
            ALU_op <= ALU_op_select(bits_to_int(opcode)) after Tpd;
            --
            wait until phi2 = '1';
            -- latch cond codes from ALU
            CC_latch_en <= '1' after Tpd;
            -- latch result for reg write
            reg_res_latch_en <= '1' after Tpd;
            wait until phi2 = '0';
            CC_latch_en <= '0' after Tpd;
            reg_res_latch_en <= '0' after Tpd;
            --
            next_state := fetch_0;          -- execution complete
            write_back_pending := true; -- register write_back required
            --
        when op_ld | op_st | op_ldq | op_stq =>
            -- enable r1 to op1_bus
            reg_port1_en <= '1' after Tpd;
            if opcode = op_ld or opcode = op_st then
                -- enable displacement to op2_bus
                disp_out_en <= '1' after Tpd;
            else
                -- enable i8 to op2_bus
                immed_signext_en <= '1' after Tpd;
            end if;
            ALU_op <= add after Tpd;  -- effective address to r_bus
            --
            wait until phi2 = '1';
            addr_latch_en <= '1' after Tpd; -- latch effective address
            wait until phi2 = '0';
            addr_latch_en <= '0' after Tpd;
            --
            next_state := execute_1;
            --
```

Figure7-26 (continued).

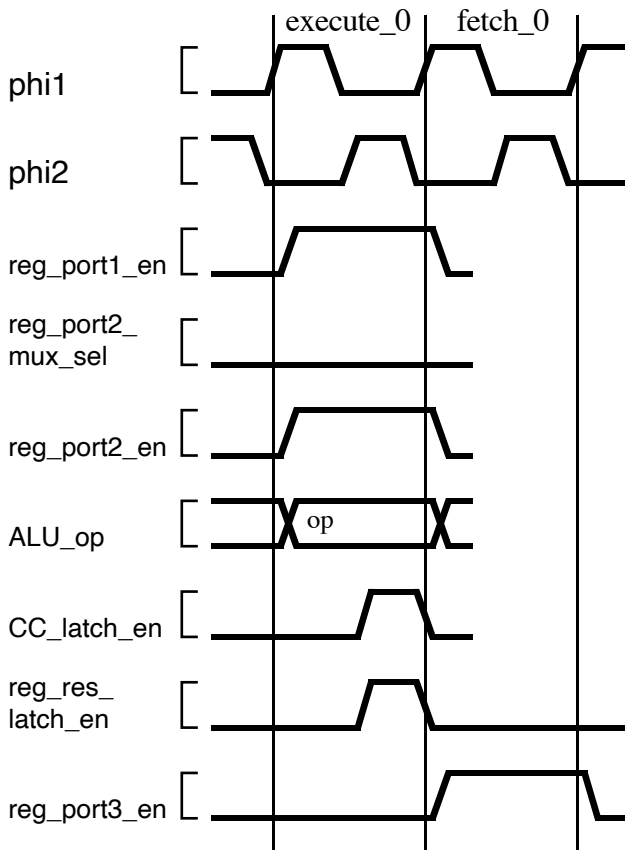


Figure7-29. Execution of register/register operations.

Execution of instructions starts in state execute_0. The first action is to negate the bus control signals that may have been active from a previous displacement fetch sequence. Subsequent action depends on the instruction being executed, so a nested case statement is used, with the op-code as the selection expression.

Arithmetic and logic instructions only require one cycle to exectute. The processor timing for the case where both operands are in registers is shown in Figure7-29. The address for register port1 is derived from the r1 field of the current instruction, and this port output is enabled onto the op1 bus. The multiplexor for the address for register port2 is set to select field r2 of the current instruction, and this port output is enabled onto the op2 bus. The ALU function code is set according to the op-code of the current instruction, and the ALU output is placed on the result bus. During the second half of the cycle, when the ALU result and condition codes are stable, the register result latch and condition code latch are enabled, capturing the results of the operation. In the next cycle, the register read ports and the latches are are disabled, and the register write port is enabled to write the result back into the destination register. This write back operation overlaps the first cycle of the next instruction fetch. The result register address, derived from the r3 field of the current instruction, is not overwritten until the end of the next instruction fetch, so the write back is performed to the correct register.

The timing for arithmetic and logical instructions where the second operand is an immediate constant is shown in Figure7-30. The difference is that register port2 is not enabled; instead, the sign extension buffer is enabled. This converts the 8-bit signed i8 field of the current instruction to a 32-bit signed integer on the op2 bus.

Looking again at the exectute_0 branch of the state machine, the nested case statement contains a branch for arithmetic and logical instructions. It firstly enables port1 of the register file, and then enables either port2 or the sign extension buffer, depending on the op-code. The lookup table ALU_op_select is indexed by the op-code to determine the ALU function code. The process then waits until the leading edge of phi2, and asserts the register result and condition code latch enables while phi2 is '1'. At the end of the cycle, the next state is set to fetch_0, and the write back pending flag is set. During the subsequent instruction fetch, this flag is checked (in the fetch_0 branch of the outer case statement). The register port3 write enable control signal is asserted during the fetch_0 state, and then at the beginning of the fetch_1 state it is negated and the flag cleared.

```
            when op_br | op_bi | op_brq | op_biq =>
                if CC_comp_result = '1' then
                    if opcode = op_br then
                        PC_out_en <= '1' after Tpd;
                        disp_out_en <= '1' after Tpd;
                    elsif opcode = op_bi then
                        reg_port1_en <= '1' after Tpd;
                        disp_out_en <= '1' after Tpd;
                    elsif opcode = op_brq then
                        PC_out_en <= '1' after Tpd;
                        immed_signext_en <= '1' after Tpd;
                    else -- opcode = op_biq
                        reg_port1_en <= '1' after Tpd;
                        immed_signext_en <= '1' after Tpd;
                    end if;
                    ALU_op <= add after Tpd;
                else
                    assert opcode = op_br or opcode = op_bi
                        report "reached state execute_0 "
                            & "when brq or biq not taken"
                        severity error;
                    PC_out_en <= '1' after Tpd;
                    ALU_op <= incr1 after Tpd;
                end if;
                --
                wait until phi2 = '1';
                PC_latch_en <= '1' after Tpd;   -- latch incremented PC
                wait until phi2 = '0';
                PC_latch_en <= '0' after Tpd;
                --
                next_state := fetch_0;
                --
            when others =>
                null;
        end case; -- op
        --
```

Figure7-26 (continued).

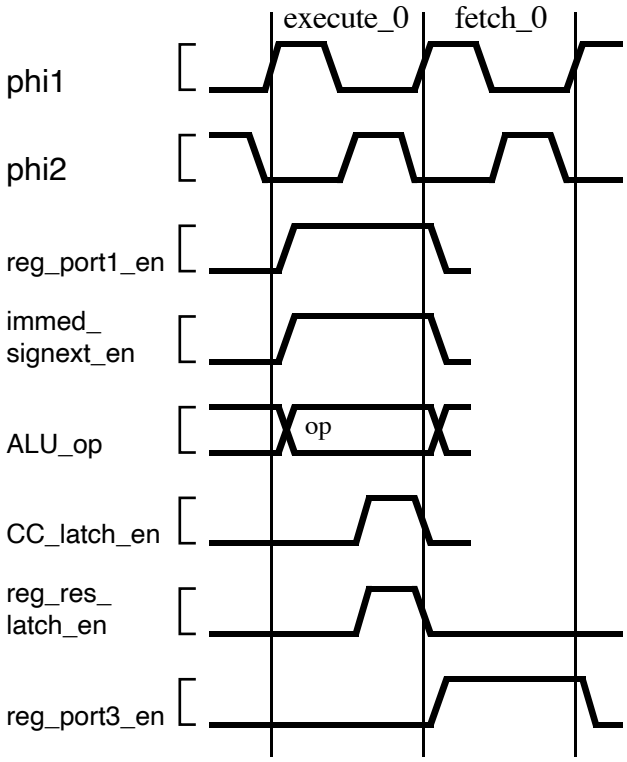


Figure7-30. Execution of register/immed operations.

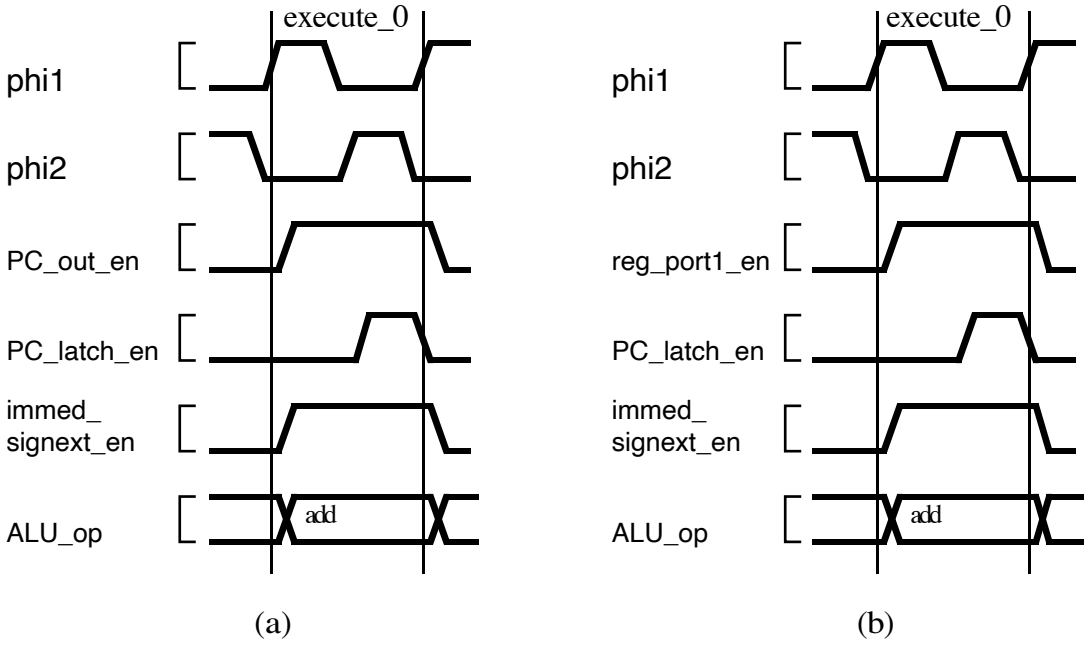


Figure7-31. Execution of quick branch with branch taken.

```
                when execute_1 =>
                    --
                    -- opcode is load or store instruction.
                    -- cleanup after previous execute_0
                    --
                    reg_port1_en <= '0' after Tpd;
                    if opcode = op_ld or opcode = op_st then
                        -- disable displacement from op2_bus
                        disp_out_en <= '0' after Tpd;
                    else
                        -- disable i8 from op2_bus
                        immed_signext_en <= '0' after Tpd;
                    end if;
                    ALU_op <= add after Tpd;        -- disable ALU from r_bus
                    --
                    -- start bus cycle
                    --
                    if opcode = op_ld or opcode = op_ldq then
                        fetch <= '0' after Tpd;              -- start bus read
                        read <= '1' after Tpd;
                    else -- opcode = op_st or opcode = op_stq
                        reg_port2_mux_sel <= '1' after Tpd;     -- address a3 to port2
                        reg_port2_en <= '1' after Tpd;          -- reg port2 to op2_bus
                        d2_en <= '1' after Tpd;       -- enable op2_bus to d_bus buffer
                        write <= '1' after Tpd;       -- start bus write
                    end if;
                    --
                    next_state := execute_2;
                    --
                when execute_2 =>
                    --
                    -- opcode is load or store instruction.
                    -- for load, enable read data onto r_bus
                    --
                    if opcode = op_ld or opcode = op_ldq then
                        dr_en <= '1' after Tpd;       -- enable data to r_bus
                        wait until phi2 = '1';
                        -- latch data in reg result latch
                        reg_res_latch_en <= '1' after Tpd;
                        wait until phi2 = '0';
                        reg_res_latch_en <= '0' after Tpd;
                        write_back_pending := true;          -- write-back pending
                    end if;
                    --
                    next_state := fetch_0;
                    --
            end case; -- state
        end process state_machine;

    end block controller;

end RTL;
```

Figure7-26 (continued).

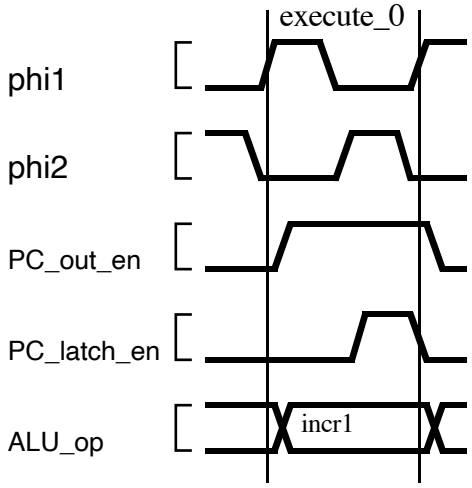


Figure7-32. Execution of branch with branch not taken.

We now move on to the execution of branch instructions. We saw previously that for quick branches, when the branch is not taken execution completes after the decode state. When the branch is taken a single execute cycle is required to update the PC with the effective address. The timing for this case is shown in Figure7-31. Figure7-31(a) shows an ordinary quick branch, in which the PC is enabled onto the op1 bus. Figure7-31(b) shows an indexed quick branch, in which the index register, read from register file port1 is enabled onto the op1 bus. The sign extension buffer is enabled to place the immediate displacement on the op2 bus, and the ALU function code is set to add the two values, forming the effective address of the branch on the result bus. This is latched back into the PC register during the second half of the execution cycle.

For branches with a long displacement, a single execution cycle is

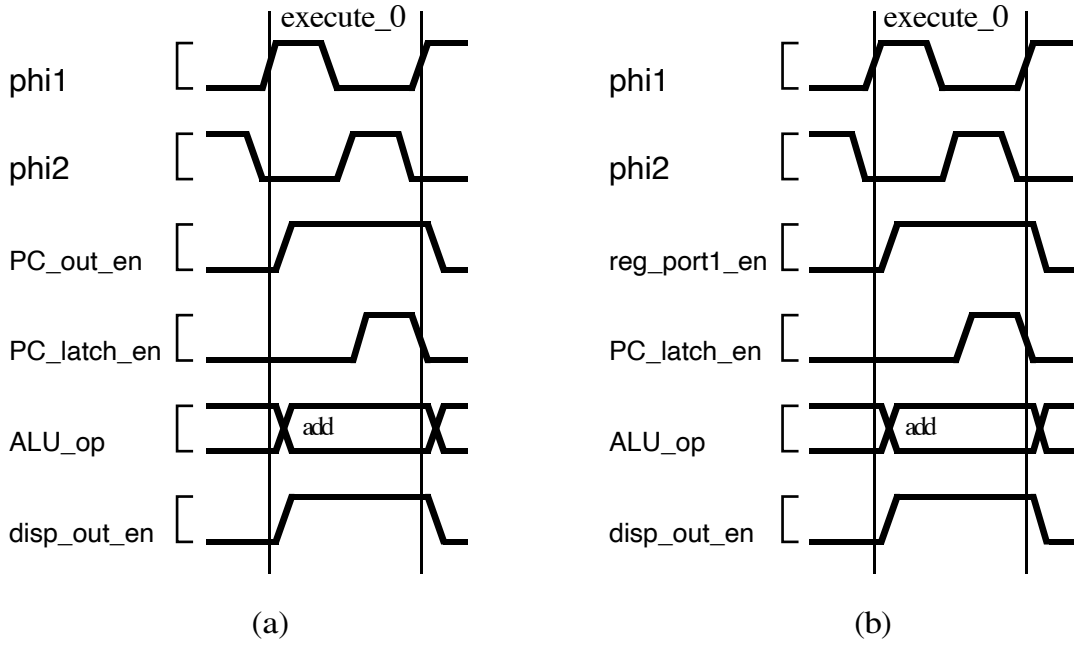


Figure7-33. Execution of branch with branch taken.

always required. If the branch is not taken, the PC must be incremented to point to the instruction after the displacment. The timing for this is shown in Figure7-32. The PC is enabled onto the op1 bus, and the ALU function is set to incr1. This increments the value and places it on the result bus. Then during the second half of the cycle, the new value is latched back into the PC register.

For long displacement branches where the branch is taken, the PC must be updated with the effective address. Figure7-33(a) shows the timing for an ordinary branch, in which the PC is enabled onto the op1 bus. Figure7-33(b) shows the timing for an indexed branch, in which the index register is enabled from register port1 onto the op1 bus. The displacement register output is enabled onto the op2 bus, and the ALU function is set to add, to add the displacement to the base address, forming the effective address on the result bus. This is latched back into the PC register during the second half of the cycle.

The VHDL description implements the execution of a branch instruction as part of the nested case statement for the execute_0 state. The process checks the result bit from the condition code comparator. If it is set, the branch is taken, so the base address and displacement are enabled (depending on the type of branch), and the ALU function code set to add. Otherwise, if the condition code comparator result is clear, the branch is not taken. This should only be the case for long branches, since quick branches should never get to the execute_0 state. An assertion statement is used to verify this condition. For long branches which are not taken, the PC is enabled onto the op1 bus and the ALU function code set to incr1 to increment the value past the displacement in memory. The PC latch enable signal is then pulsed when phi2 changes to '1'. Finally, the next state is set to fetch_0, so the processor will continue with the next instruction.

The remaining instructions to be considered are the load and store instructions. These all take three cycles to execute, since a bus transaction is required to transfer the data to or from the memory. For long displacement loads and stores, the displacement has been previously fetched into the displacement register. For the quick forms, the immediate displacement in the instruction word is used.

Figure7-34 shows the timing for execution of load and quick load instructions. The base address register is read from register file port1 and enabled onto the op1 bus. For long displacement loads, the previously fetched displacement is enabled onto the op2 bus, and for quick loads, the sign extended immediate displacement is enabled onto the op2 bus. The ALU function code is set to add, to form the effective address on the result bus, and this is latched into the memory bus address register during the second half of the first execute cycle. During the next two cycles the controller performs a memory read transaction, with the fetch signal held negated. The data from the data bus is enabled onto the result bus through the connecting buffer, and latched into the register result latch. This value is then written back to the register file during the first cycle of the subsequent instruction fetch.

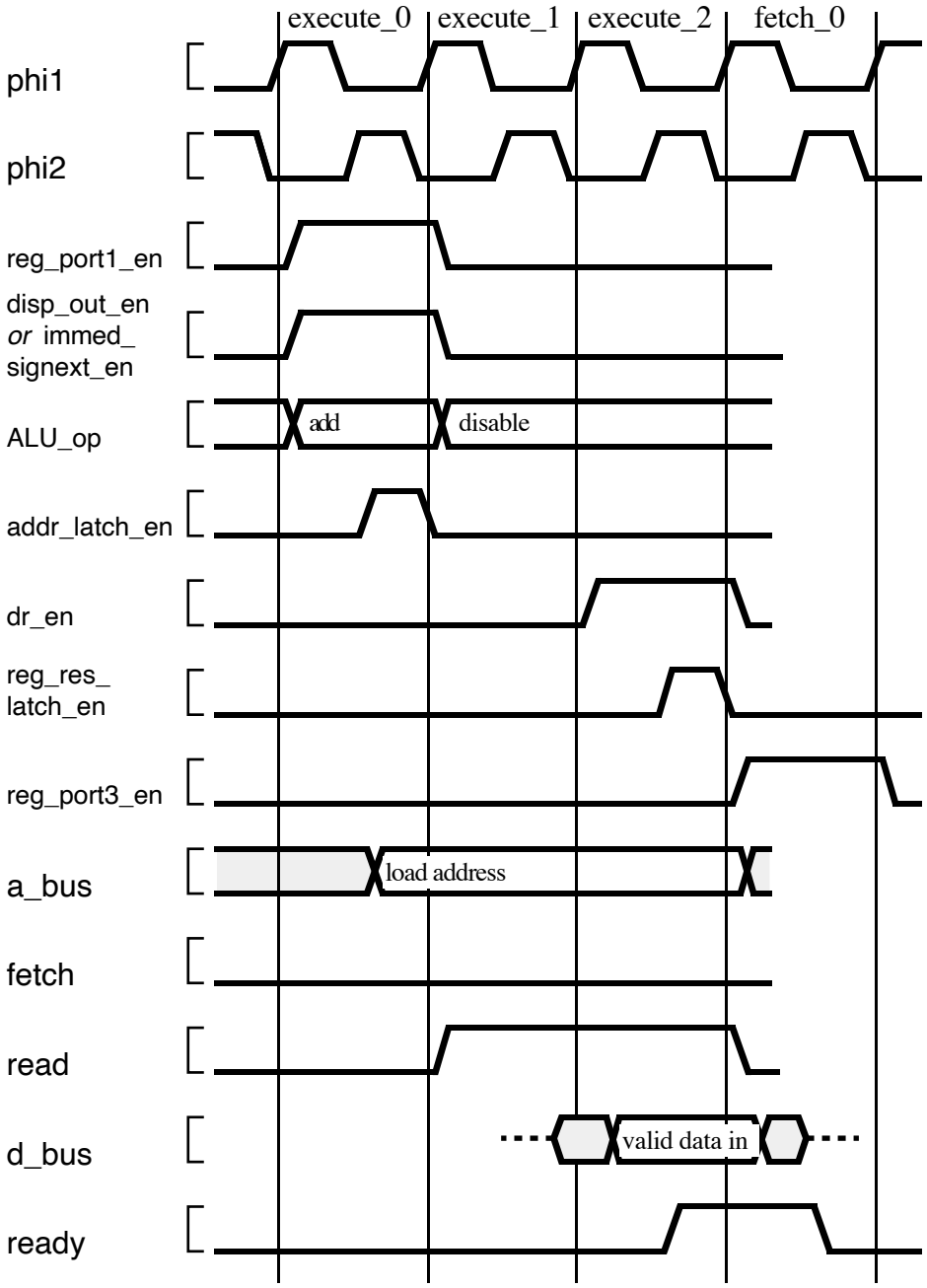


Figure7-34. Execution of load instructions.

The timing for execution of store and quick store instructions is shown in Figure7-35. As with load instructions, the base address and displacement are added, and the effective address is latched in the memory bus address register. During the next two cycles the controller performs a bus write transaction. The multiplexor for the register file port2 address is set to select the r3 field of the instruction, which specifies the register to be stored, and the port2 output is enabled onto the op2 bus. The buffer between the op2 bus and the memory data bus is enabled to transmit the data to the memory. Execution of the instruction completes at the end of the bus transaction.

Returning to the VHDL description, the first cycle of execution of load and store instructions is included as a branch of the nested case in the execute_0 state. The base address register output port is enabled, and either the displacement latch output or the sign extension buffer is enabled, depending on the instruction type. The ALU function code is set to add the two to form the effective address. The process then waits until phi2 changes to '1', indicating the second half of the cycle, and pulses the address latch enable. The next state is then set to execute_1 to continue execution of the instruction.

In state execute_1, the process firstly removes the base address, displacement and effective address from the DP32 internal buses, then starts a memory bus transaction. For load instructions, the fetch signal is negated and the read signal is asserted. For store instructions, the source register value is enabled onto the op2 bus, the memory data bus output buffer is enabled, and the write signal is aserted. The next state variable is then set to execute_2 for the next cycle.

In state execute_2, for load instructions, the memory data bus input buffer is enabled to transmit the data onto the result bus. The process then waits until phi2 is '1', in the second half of the cycle, and pulses the enable for the register result latch. The write back pending flag is then set to schedule the destination register write during the next instruction fetch cycle. For both load and store instructions, the next state is fetch_0. All control signals set during the execute_1 state will be returned to their negated values in the fetch_0 state.

The test bench described in Section7.5 can be used to test the register transfer architecture of the DP32. This is done using an alternate configuration, replacing the behavioural architecture in the test bench with the register transfer architecture. Figure7-36 shows such a configuration. The entity bindings for the clock generator and memory are the same, using the behavioural architectures, but the processor component instance uses the rtl architecture of the dp32 entity. This binding indication is followed by a configuration for that architecture, binding the entities described in Sections7.6.1–7.6.9 to the component instances contained in the architecture. The newly configured description can be simulated using the same test programs as before, and the results compared to verify that they implement the same behaviour.

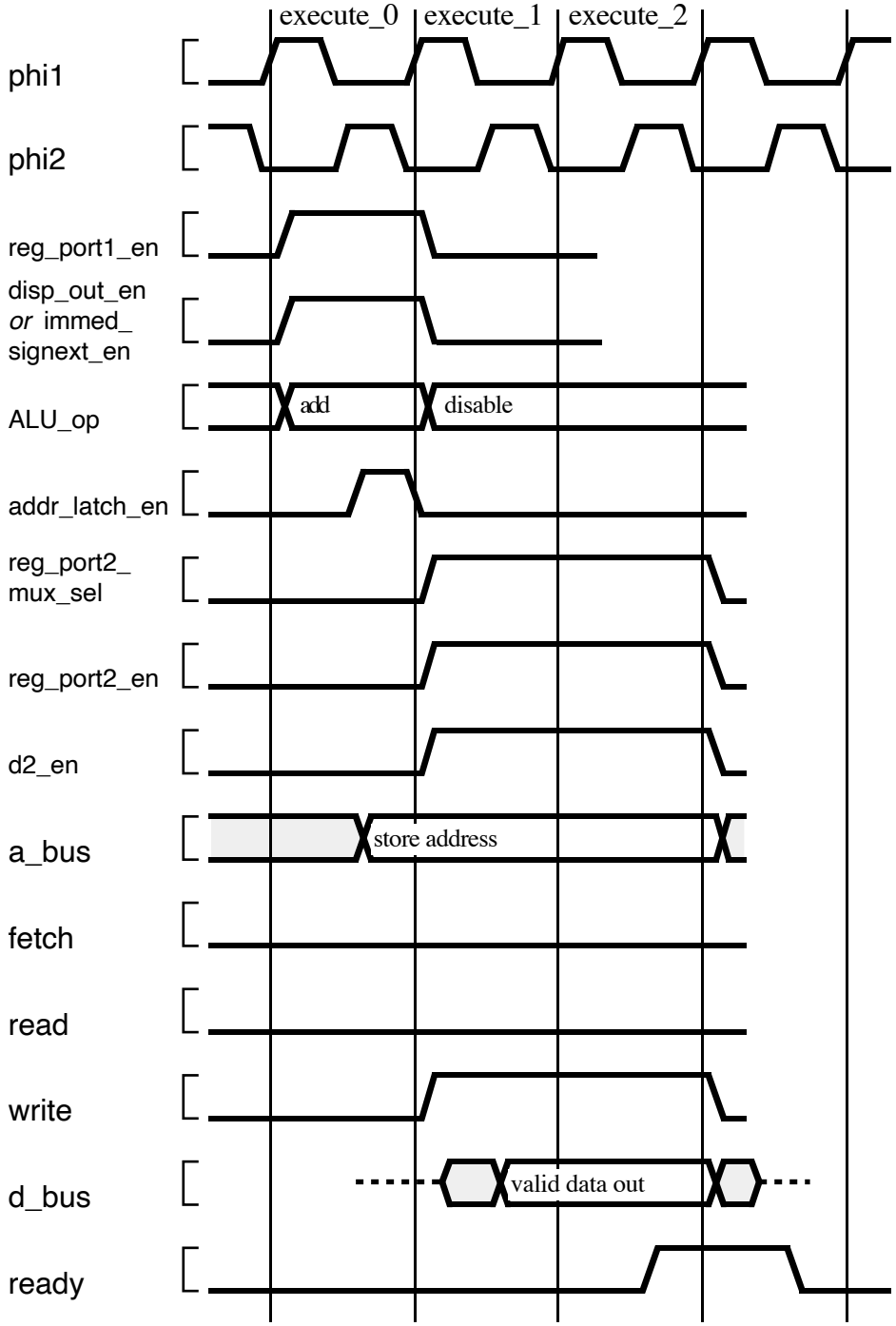


Figure7-35. Execution of store instructions.

```
use work.dp32_types.all;

configuration dp32_rtl_test of dp32_test is

    for structure
        for cg : clock_gen
            use entity work.clock_gen(behaviour)
                generic map (Tpw => 8 ns, Tps => 2 ns);
        end for;
        for mem : memory
            use entity work.memory(behaviour);
        end for;
        for proc : dp32
            use entity work.dp32(rtl);
            for rtl
                for all : reg_file_32_rrw
                    use entity work.reg_file_32_rrw(behaviour);
                end for;
                for all : mux2
                    use entity work.mux2(behaviour);
                end for;
                for all : latch
                    use entity work.latch(behaviour);
                end for;
                for all : PC_reg
                    use entity work.PC_reg(behaviour);
                end for;
                for all : ALU_32
                    use entity work.ALU_32(behaviour);
                end for;
                for all : cond_code_comparator
                    use entity work.cond_code_comparator(behaviour);
                end for;
                for all : buffer_32
                    use entity work.buffer_32(behaviour);
                end for;
                for all : latch_buffer_32
                    use entity work.latch_buffer_32(behaviour);
                end for;
                for all : signext_8_32
                    use entity work.signext_8_32(behaviour);
                end for;
            end for;
        end for;
    end for;

end dp32_rtl_test;
```

Figure7-36. Configuration using register transfer architecture of DP32.