Recursive and Repetitive Hardware Models in VHDL

Peter J. Ashenden

petera@cs.adelaide.edu.au

Technical Report TR 160/12/93/ECE

Department of Electrical & Computer Engineering,
University of Cincinnati,
Ohio, 45221–0030, USA

also published as

Technical Report 93–19

Department of Computer Science,
The University of Adelaide,
South Australia, 5005, Australia

Recursive and Repetitive Hardware Models in VHDL

Peter J. Ashenden

petera@cs.adelaide.edu.au

Abstract

In this report, techniques for developing VHDL descriptions of recursive hardware structures are examined. In particular, both recursive and repetitive component instantiations are examined for modeling recursive structures. This report examines two example structures: a fanout tree of buffers and a fat tree interconnection network. For both structures, the recursive descriptions are easier to develop, and more clearly express the structure, making them easier to understand. The main difficulty in developing repetitive structures lies in devising a way of interconnecting the basic components comprising the structure. It is shown that the difficulties result from fundamental language design decisions made in VHDL, and that it is not appropriate to modify the language to avoid the difficulties. Hence, in general, the recursive style is preferred over the repetitive style for describing recursive hardware structures.

1. Introduction

Many of the hardware structures that are designed consist of a number of instances of a single basic component, interconnected in a regular pattern. For example, a register consists of a one-dimensional array of flip flop cells with inputs and outputs connected in parallel to buses that carry words of data. Similarly, a memory circuit consists of a two-dimensional array of storage cells, with parallel connections to data and select signals.

In order to describe such regular structures, most hardware description languages, such as VHDL [1, 2], Verilog [3] and Ella [4], include some form of repetitive generator, allowing us to repeatedly instantiate a basic cell and its connections. Most languages also include a conditional generator that can be used to express variations within the regular structure, such as often occur at the boundaries. For example, the end cells of a shift register are not connected to neighbouring cells, but to signals in the enclosing environment.

The mechanism provided in VHDL for generating regular structures is the *generate statement*. The language includes both repetitive and conditional forms of the generate statement, which achieve their effect when a description is elaborated prior to simulation or synthesis. The repetitive form specifies a discrete range of values, and for each of these it generates an instance of the statements in the body of the generate statement. For each repetition, the value from the range is bound to an identifier, and can be used as an index to select elements of signal arrays for connection to components. The conditional form of generate statement includes a boolean expression, which governs whether the statements comprising the body of the generate statement are included. Figure 1 shows an example, in which generate statements are used to describe an adder design.

```
word_adder : for i in 0 to adder_size – 1 generate
    lsb : if i = 0 generate
        lsb_adder : half_adder
            port map ( a => addend(0), b => augend(0), s => sum(0), c => carry(0) );
    end generate lsb;
    other_bits : if i /= 0 generate
        other_adder : full_adder
            port map (  a => addend(i), b => augend(i), c_in => carry(i – 1),
                        s => sum(i), c_out => carry(i)  );
    end generate other_bits;
end generate shift_reg;
```

Figure 1. Examples of generate statements used to describe a ripple-carry adder. The statement labelled word_adder is a repetitive generate statement that enumerates a collection of cells. For each cell, the parameter i is bound to a value in the range 0 to adder_size – 1. The nested statements are conditional generate statements. The statement labelled lsb creates an instance of a half adder cell if the value of the parameter i is zero. The statement labelled other_bits creates instances of full adder cells for all other values of the parameter.

There is, however, an important class of regular structures, namely *recursive structures*, that can be difficult to describe with just these forms of generators. A recursive structure is one which is parameterized with respect to its size, and which is described in terms of smaller instances of the same structure. For example, a tree structure of height h can be described as root cell connected to some number of sub-trees of height h–1. While it is straightforward to describe a recursive structure using just repetitive generators if the size is fixed and known *a priori*, in general it is difficult if the size is a parameter whose actual value is not known. This is a significant problem, since good design practice dictates that we should parameterize aspects of a system to make it resilient to changes in design specifications, and reusable in as many contexts as possible.

In this report, we examine techniques for modeling parameterized recursive structures using VHDL. In particular, we examine, prepare and compare recursive and repetitive constructions using the VHDL generate statement to model such structures. The techniques are illustrated by two example structures, a fanout tree of buffers and a fat tree interconnection network. Comparisons between the recursive and repetitive descriptions highlight the simplifications afforded by the recursive descriptions.

2. Parameterized Recursive Structures

In general, a parameterized recursive structure can be defined as a hardware entity $E_s = (s,\ p_1,\ldots,p_n,\ I,\ O)$, where $s \in \mathbb{N}$ is a parameter that specifies the size of the structure, $p_1,\ldots,p_n$ are additional parameters that determine the implementation of the structure, I is a set of input ports, and O is a set of output ports. The sizes of I and O may depend on any or all of the parameters s and $p_1,\ldots,p_n$. Furthermore, the implementation of E_s includes one or more instances of E_{s-1} ($s > t$, for some t), and the implementation of E_t contains no instances of E_i for any i. This last condition on E_t provides a means of terminating the recursion.

A recursive structure can be expressed in VHDL using an entity declaration, with s and $p_1, \ldots, p_n$ being represented by generic constants, and I and O being represented by ports of the entity. If the ports are of some unconstrained array base type, then the values of the generic constants can be used to constrain the index ranges of the ports, thus making their size depend on the entity parameters. For example, an entity declaration for a recursive structure could be written as shown in Figure 2. In this declaration, T_s is the VHDL type of the parameter s; $T_{p1}, \ldots, T_{pn}$ are the types of the parameters $p_1, \ldots, p_n$; $A_{I1}, A_{I2}, \ldots$ and $A_{O1}, A_{O2}, \ldots$ are unconstrained array types for ports I1, I2, … and O1, O2, …; and $f_{I1}, f_{I2}, \ldots$ and $f_{O1}, f_{O2}, \ldots$ are functions that determine the sizes of the ports from the parameter values.

The implementation of the entity is described in an architecture body, which uses conditional generators including the value of s in their conditions. If $s = t$, a structure that does not include instances of the entity is generated. If $s > t$, a structure is generated that includes instances of the entity, with the generic constant representing the size set to s–1. If the original 1987 version of VHDL [1] is used, the instantiation of the entity within its own architecture body must be done indirectly by instantiating a declared component, which is bound to the entity in a separate configuration declaration. Furthermore, any signals used to interconnect the generated component instances must be declared in the outer part of the architecture body, rather than locally to the

```
entity E is
    generic ( s : Ts;  p1 : Tp1;  …;  pn : Tpn );
    port (    I1 : in AI1(0 to fI1(s, p1, …, pn));
              I2 : in AI2(0 to fI2(s, p1, …, pn));
              …;
              O1 : out AO1(0 to fO1(s, p1, …, pn));
              O2 : out AO2(0 to fO2(s, p1, …, pn));
              …  );
end E;
```

Figure 2. The form of a VHDL entity declaration for a parameterized recursive structure.

```
architecture A of E is
    signal S_I1_1, S_I2_2, … : A_I1(0 to f_SI1(s, p1, …, pn));
    …
    signal S_O1_1, S_O2_2, … : A_O1(0 to f_SO1(s, p1, …, pn));
    …
    component B
        generic ( … );
        port ( … );
    end component;
    component C
        generic ( s : T_s;  p1 : T_p1;  …;  pn : T_pn );
        port (  I1 : in A_I1(0 to f_I1(s, p1, …, pn));
                I2 : in A_I2(0 to f_I2(s, p1, …, pn));
                …;
                O1 : out A_O1(0 to f_O1(s, p1, …, pn));
                O2 : out A_O2(0 to f_O2(s, p1, …, pn));
                …  );
    end component;
begin
    termination : if s = t generate
        …           -- statements not instantiating C
    end generate termination;
    recursion : if s > t generate
        repetition_of_B : for i in … generate
            basic_instance : B
                generic map ( … )
                port map ( … );
        end generate repetition_of_B;
        repetition_of_C : for i in … generate
            recursive_instance : C
                generic map ( s – 1, … )
                port map ( … );
        end generate repetition_of_C;
    end generate recursion;
end A;
```

Figure 3. Outline of an architecture body for a recursive description of a parameterized recursive structure.

generate statements. The value of the generic constant representing the size of the structure may need to be used to determine how many signals to declare. An architecture body of this form is outlined in Figure 3. The component B represents the basic component from which the recursive structure is constructed. The component C is the template corresponding to the entity representing the recursive structure. The statement labelled "termination" is the conditional generator that terminates recursion when $s = t$. It contains instances of the component B only. The statement labelled "recursion" is the conditional generator that creates the recursive instances of the structure. It contains two repetitive generator, one labelled "repetition_of_B" that creates a number of instances of the basic component, and one labelled "repetition_of_C" that creates a number of instances of the component representing the recursive structure. The ports of each of these instances are connected to the signals declared in the declarative region of the architecture body.

The more recent 1993 version of VHDL offers some simplification, particularly if the flexibility of indirect instantiation is not needed. The entity may be directly instantiated in the generated structure, and the interconnecting signals may be declared locally to the generate statements. For most of the examples in this report, the (currently) more widely used VHDL-87 version of the language will be used. In some examples, where the restrictions in VHDL-87 would make the model cumbersome, VHDL-93 is used and the reasons for its use are given.

3. Fanout Tree

A relatively simple example of a recursive structure is a fanout tree, used to generate a number of buffered clock signals to be distributed throughout a circuit. A tree structure is used to generate signals with minimum skew between them, by ensuring that the loading and signals delays are symmetrically distributed in the buffering network. The size of the fanout tree is expressed in terms of its height h. An additional parameter, d, the degree of the tree, specifies how many buffers at each level are connected to a single buffer at the preceding level. Some fanout trees of varying heights and degrees are shown in Figure 4.

3.1. Recursive Description of the Fanout Tree

As Figure 4(c) shows, a fanout tree with $h = 3$ can be described as a composition of d buffers, each connected to a fanout tree with $h = 2$. Each of these in turn can be described as a composition of d buffers, each connected to a fanout tree with $h = 1$. If we take the view that a through connection is a degenerate fanout tree with $h = 0$, then we can describe the fanout tree with $h = 1$ in terms of d fanout trees with $h = 0$, which do not include any further fanout three instances. Thus the fanout tree meets our criteria for recursive structures presented above.

We can describe the fanout tree in VHDL as follows. First, the entity declaration shown in Figure 5 describes the interface. The height of the tree is represented by the generic constant `h`,

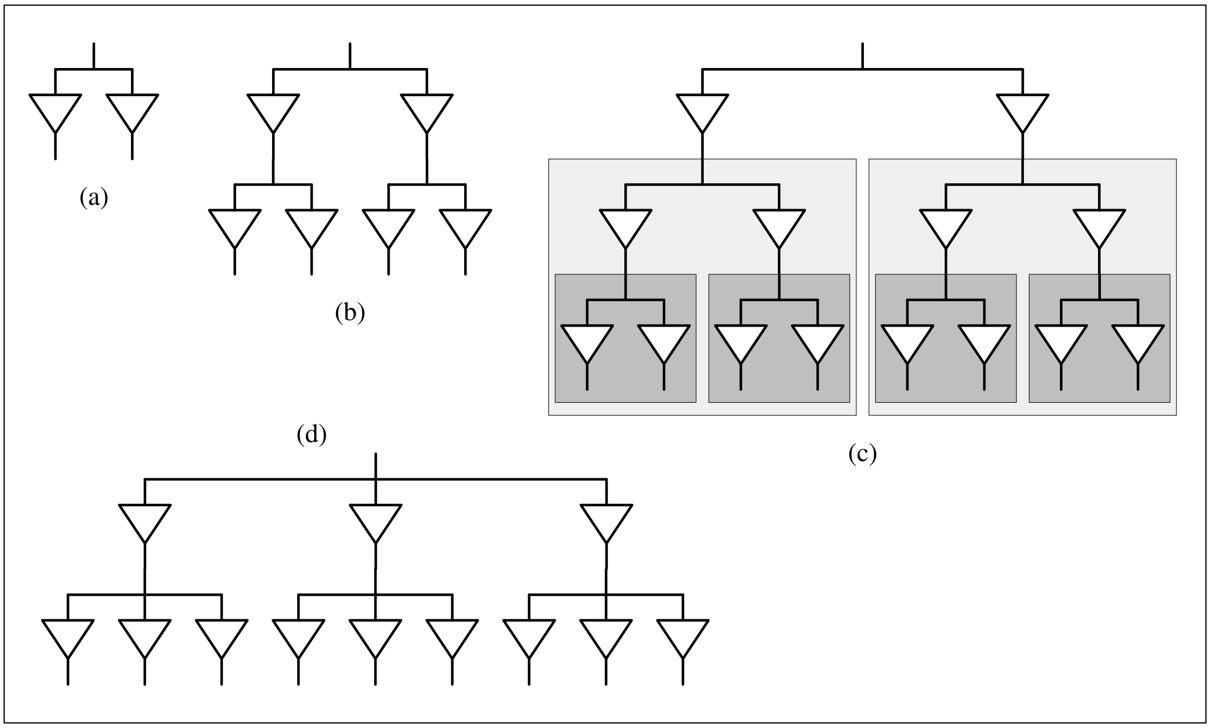

Figure 4. Recursive fanout trees: (a) $h = 1$, $d = 2$; (b) $h = 2$, $d = 2$; (c) $h = 3$, $d = 2$, showing decomposition into subtrees of $h = 2$ and $h = 1$; (d) $h = 2$, $d = 3$.

```
entity fanout_tree is
    generic ( h : natural;  d : positive );
    port ( input : in bit;  output : out bit_vector (0 to d**h – 1) );
end fanout_tree;
```

Figure 5. Entity declaration for the fanout tree.

```
architecture recursive of fanout_tree is
    component buf
        port ( a : in bit;  y : out bit );
    end component;
    component fanout_tree
        generic ( h : natural;  d : positive );
        port ( input : in bit;  output : out bit_vector(0 to d**h – 1) );
    end component;
    signal buffered_input : bit_vector(0 to d – 1);
begin
    degenerate_tree : if h = 0 generate
        output(0) <= input;
    end generate degenerate_tree;
    compound_tree : if h > 0 generate
        subtree_array : for i in 0 to d – 1 generate
            the_buffer : buf
                port map ( a => input, y => buffered_input(i) );
            the_subtree : fanout_tree
                generic map ( h => h – 1, d => d )
                port map (   input => buffered_input(i),
                             output => output(i * d**(h–1) to (i+1) * d**(h–1) –1) );
        end generate subtree_array;
    end generate compound_tree;
end recursive;
```

Figure 6. Recursive architecture body for the fanout tree.

and the second generic constant d represents the fanout degree. The number of elements in the output port is dependent upon both of these parameters.

Next, a recursive architecture declaration for the fanout tree is shown in Figure 6. The component buf represents the buffer from which the fanout tree is constructed. The component fanout_tree is the template that is to be instantiated as a number of subtrees below the top level of the fanout tree. These instances will be bound to the fanout tree entity in a separate configuration declaration. The bit vector signal buffered_input represents the interconnections between the buffers at the top level of the tree and the subtrees that comprise the lower levels. The architecture body include two conditional generate statements. If the value of the generic constant h is zero, a degenerate tree passing the input value directly to the output is generated. This causes

```
configuration recursive_fanout_tree of fanout_tree is
    for recursive
        for compound_tree
            for subtree_array
                for the_buffer : buf
                    use entity work.buf(behaviour);
                end for;
                for the_subtree : fanout_tree
                    use configuration recursive_fanout_tree;
                end for;
            end for;
        end for;
    end for;
end recursive_fanout_tree;
```

Figure 7. Configuration declaration for the recursive fanout tree.

the recursion to bottom out. If the value of the generic constant h is non-zero, the statement labelled compound_tree generates an array of cells, each consisting of a buffer connected to a subtree of height h – 1. This subtree has d^{h-1} outputs, which are connected to a slice of that length of the outputs of the containing tree.

In order to complete the recursion in the fanout tree description, each instance of the fanout tree component must be bound to the fanout tree entity and its recursive architecture body. A configuration declaration that achieves this is shown in Figure 7. No block configuration is required for the generate statement degenerate_tree, since it does not contain any component instances. For the generate statement compound_tree, the block configuration contains a nested block configuration that configures each repetition of the generate statement subtree_array identically. The buffer component instance is bound to a behavioural implementation of a buffer entity. The recursive instance of the fanout tree component is bound using a recursive application of the configuration. Hence, when a fanout tree is elaborated, the configuration will be used for each recursive instance.

3.2. Repetitive Description of the Fanout Tree

Figure 8 shows a purely repetitive description of the fanout tree using just repetitive generate statements. Each iteration of the outer generator, labelled levels, generates one level of the tree. Within that level, the generator labelled buf_array_array generates an array of sub-arrays of buffers. Each sub-array is of width d, and generates d buffered outputs. The input to each sub-array is one of the outputs from the previous level.

The main complication in this description lies in the interconnection of the buffers in the structure. First, it is not possible to generate the signals using generate statements. This is because the signals must be declared in a static declarative region of the description in such a way as to

```
architecture repetitive of fanout_tree is
    component buf
        port ( a : in bit;  y : out bit );
    end component;
    signal buffered_input : bit_vector(0 to (d**(h + 1) – 1) / (d – 1) – 1 );
begin
    buffered_input(0) <= input;
    levels : for k in 1 to h generate
        buf_array_array : for i in 0 to d**(k – 1) – 1 generate
            buf_array : for j in 0 to d – 1 generate
                the_buf : buf
                    port map (   a => buffered_input( (d**(k – 1) – 1) / (d – 1) + i ),
                                 y => buffered_input( (d**k – 1) / (d – 1) + d * i + j ) );
            end generate buf_array;
        end generate buf_array_array;
    end generate levels;
    output <= buffered_input( (d**h – 1) / (d – 1) to (d**(h + 1) – 1) / (d – 1) – 1 );
end repetitive;
```

Figure 8. Repetitive description of the fanout tree.

be visible to the generated component instances to which they are connected. Since each signal used as an output for level k must be used as an input for level $k + 1$, the signals must be declared in the region enclosing the generate statement labelled levels, namely in the architecture body. Second, since the structure is parameterized with respect the height and degree, the number of signals is dependent upon these two parameters.

The solution to the interconnection problem in this repetitive model is to recognise that at each level k there are d^k buffers each with a separate output signal. Thus the total number of signals needed (including the input and output ports) is:

$$S_h = 1 + d + d^2 + \cdots + d^h = \frac{d^{h+1}-1}{d-1}.$$

The signals are represented in a one-dimensional array, buffered_input, with element 0 connected to the fanout tree input, elements 1 to d connecting the level 1 buffers to the next level, and so forth. The last d^h elements are connected to the outputs of the fanout tree. Figure 9 shows this structure for a fanout tree of height $h = 3$.

The position of each buffer component within the fanout tree structure determines the indices of the buffered_input vector to which the buffer's input and output signals are connected. Each subtree has position coordinates (k, i) in the tree, where k is the level, and $i \in \{0, \ldots, d^{k-1}-1\}$ is the subtree index. Each buffer has position coordinates (k, i, j), where k and i are the coordinates of the containing subtree, and $j \in \{0, \ldots, d-1\}$ is the index of the buffer within the subtree.

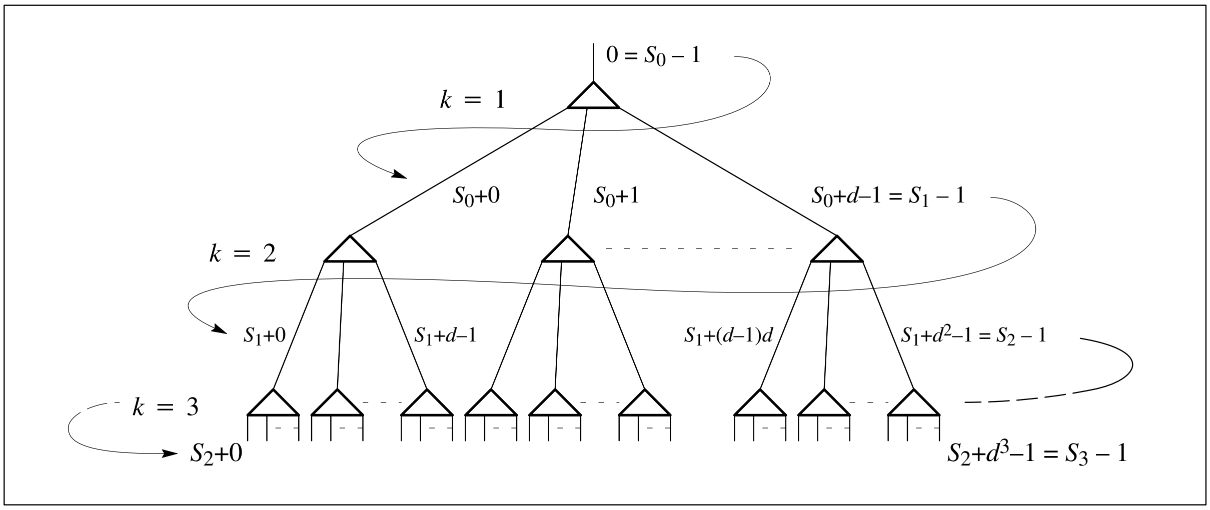

Figure 9. Organisation of the repetitive description of the fanout tree. Each triangle represents an array of d buffers. Interconnecting signals are indexed from 0 at the root of the tree, to $S_h - 1$ at the bottom right leaf of the tree.

These coordinates are represented by the generate statement parameters in the VHDL description. By counting all of the signals in the array buffered_input up to the input of the buffers in the subtree at position (k, i), we see that these buffers take their input from element $M_{in}(k, i)$, where:

$$M_{in}(k, i) = S_{k-2} + i = \frac{d^{k-1}-1}{d-1} + i \, .$$

(Note that $S_{-1} = 0$.) Similarly, the buffer at position (k, i, j) generates its output on element $M_{out}(k, i, j)$, where:

$$M_{out}(k, i, j) = S_{k-1} + di + j = \frac{d^{k}-1}{d-1} + di + j \, .$$

As with the recursive description of the fanout tree, we need to write a configuration declaration to bind component instances to entities. Figure 10 shows a configuration for the repetitive description. It is somewhat simpler than that for the recursive description, since it only needs to bind the instances of the buffer components to the behavioural implementation of the buffer entity.

```
configuration repetitive_fanout_tree of fanout_tree is
    for repetitive
        for levels
            for buf_array_array
                for buf_array
                    for the_buf : buf
                        use entity work.buf(behaviour);
                    end for;
                end for;
            end for;
        end for;
    end for;
end repetitive_fanout_tree;
```

Figure 10. Configuration declaration for the repetitive fanout tree.

4. Fat Tree

A more complex example of a recursive structure is a fat tree interconnection network, described by Leiserson in [5], and used in multicomputers such as the Thinking Machines CM–5 [6]. A fat tree can be composed of switch elements that route messages between elements of a multicomputer. Each switch has c child connections and p parent connections. It can route messages between its children directly, and can forward messages for nephews via any of its parents. Switches are composed into a three dimensional structure, branching downwards across one plane, and upwards across an orthogonal plane. Figure 11 illustrates a number of fat trees with $c = 3$ and $p = 2$, and shows how they can be composed in a bottom up recursive manner. Figure 12 shows how the same networks can be composed in a top down recursive manner. A fat tree with height h has c^h child connections for communication with child switches or the processor elements of the multicomputer, and p^h parent connections to communicate with parent switches in the interconnection network. The topmost parent connections are not used. The important feature of the fat tree topology is that the aggregate communication bandwidth increases

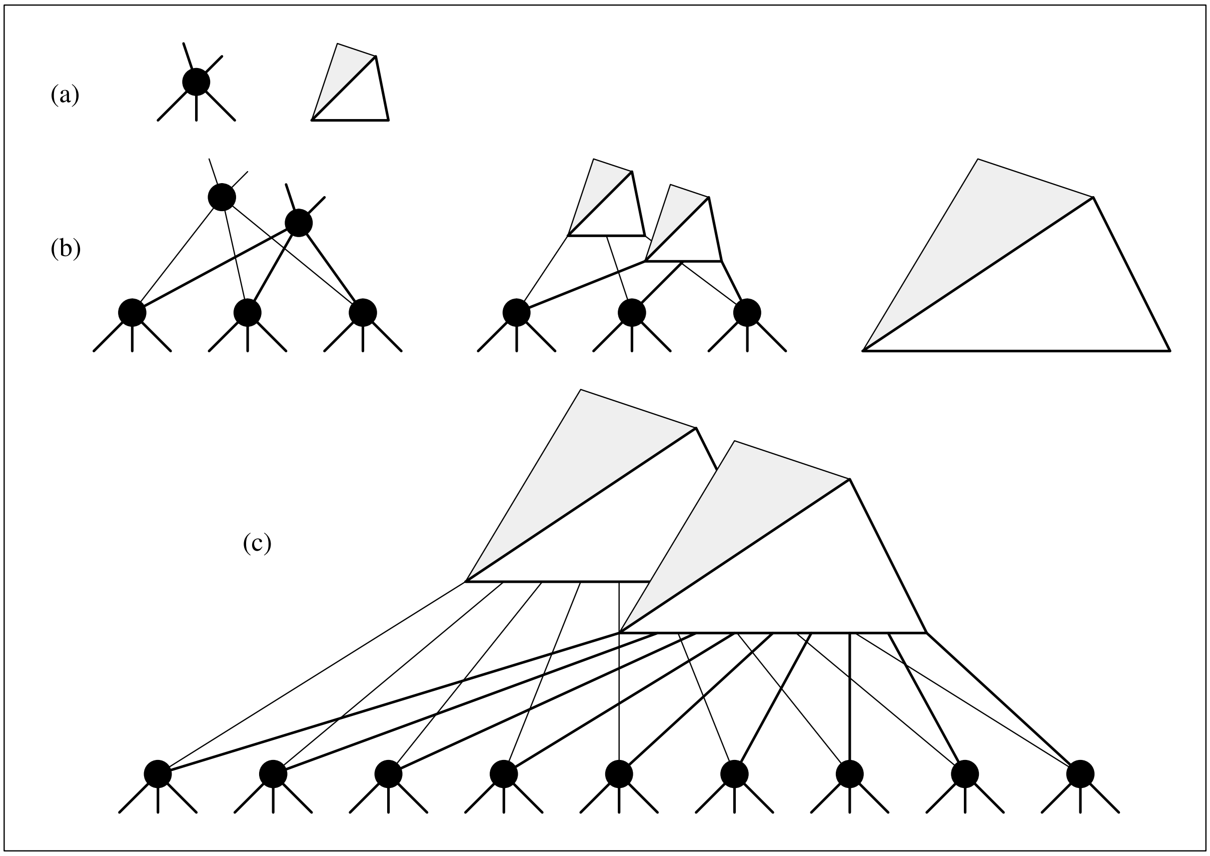

Figure 11. Bottom up compositions of fat trees with $c = 3$ and $p = 2$: (a) a single switch component, comprising a fat tree with $h = 1$, and a symbolic representation of the fat tree; (b) a fat tree with $h = 2$, its decomposition into switch components and fat trees with $h = 1$, and it symbolic representation; (c) a fat tree with $h = 3$, composed of switch components and fat trees with $h = 2$.

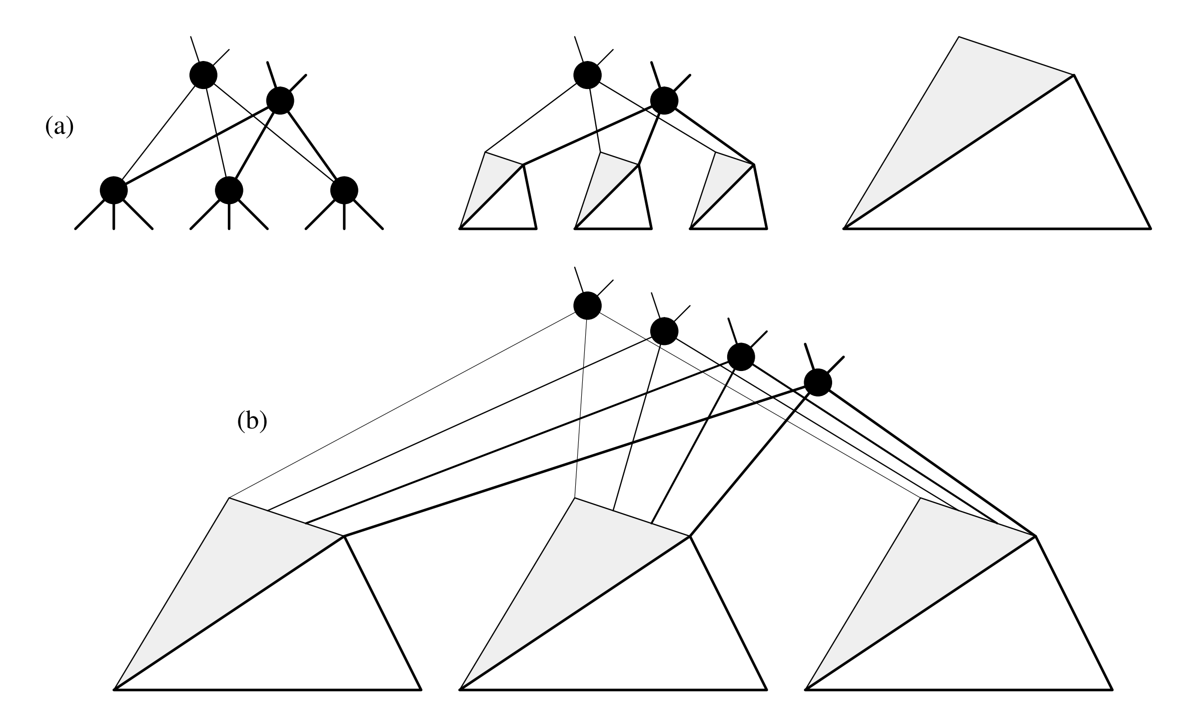

Figure 12. Top down compositions of fat trees with $c = 3$ and $p = 2$: (a) a fat tree with $h = 2$, its decomposition into switch components and fat trees with $h = 1$, and it symbolic representation; (b) a fat tree with $h = 3$, composed of switch components and fat trees with $h = 2$.

```
use work.fat_tree_types.all;
entity fat_tree is
    generic ( h : natural;  c, p : positive );
    port (  receive_msg_from_child :   in connection_vector(0 to c**h – 1);
            send_ack_to_child :        out acknowledge_vector(0 to c**h – 1);
            send_msg_to_child :        out connection_vector(0 to c**h – 1);
            receive_ack_from_child :   in acknowledge_vector(0 to c**h – 1);
            send_msg_to_parent :       out connection_vector(0 to p**h – 1);
            receive_ack_from_parent :  in acknowledge_vector(0 to p**h – 1) := (others => false);
            receive_msg_from_parent :  in connection_vector(0 to p**h – 1)
                                           := (others => default_connection);
            send_ack_to_parent :       out acknowledge_vector(0 to p**h – 1) );
end fat_tree;
```

Figure 13. Entity declaration for a parameterized fat tree interconnection network.

in the higher levels of the network as it scales, thus avoiding the congestion that occurs in a planar tree network [5].

An entity declaration for the fat tree network is shown in Figure 13. The type connection_vector is an array of connection records, and is used to represent a collection of message connections between switches. The type acknowledge_vector is an array of acknowledgment signals used to synchronise the message passing protocol. The precise composition of these types is not directly relevant to a description of the structure, so it is deferred until the fat tree test bench is discussed

```
use work.fat_tree_types.all;
entity aux_fat_tree is
    generic ( h : natural;  c, p : positive;  k : positive;  i, j : natural );
    port (  receive_msg_from_child :   in connection_vector(0 to c**h – 1);
            send_ack_to_child :        out acknowledge_vector(0 to c**h – 1);
            send_msg_to_child :        out connection_vector(0 to c**h – 1);
            receive_ack_from_child :   in acknowledge_vector(0 to c**h – 1);
            send_msg_to_parent :       out connection_vector(0 to p**h – 1);
            receive_ack_from_parent :  in acknowledge_vector(0 to p**h – 1);
            receive_msg_from_parent :in connection_vector(0 to p**h – 1);
            send_ack_to_parent :       out acknowledge_vector(0 to p**h – 1) );
end aux_fat_tree;


architecture aux of fat_tree is
    component aux_fat_tree
        generic ( h : natural;  c, p : positive;  k : positive;  i, j : natural );
        port ( receive_msg_from_child :   in connection_vector(0 to c**h – 1);
               send_ack_to_child :        out acknowledge_vector(0 to c**h – 1);
               send_msg_to_child :        out connection_vector(0 to c**h – 1);
               receive_ack_from_child :   in acknowledge_vector(0 to c**h – 1);
               send_msg_to_parent :       out connection_vector(0 to p**h – 1);
               receive_ack_from_parent :  in acknowledge_vector(0 to p**h – 1);
               receive_msg_from_parent :in connection_vector(0 to p**h – 1);
               send_ack_to_parent :       out acknowledge_vector(0 to p**h – 1) );
    end component;
begin
    the_tree : aux_fat_tree
        generic map ( h, c, p, k => 1, i => 0, j => 0 )
        port map ( receive_msg_from_child, send_ack_to_child,
                   send_msg_to_child, receive_ack_from_child,
                   send_msg_to_parent, receive_ack_from_parent,
                   receive_msg_from_parent, send_ack_to_parent );
end aux;
```

Figure 14. Top: entity declaration for the auxiliary fat tree, with additional generic constants required for the routing algorithm of the switches in the tree. Bottom: architecture declaration for the fat tree, instantiating the auxiliary fat tree component.

later in this section. The fat tree entity is parameterized by height (h), and by child and parent degrees (c and p) using generic constants. Because the number of connections between the tree and the child and parent components must be determined by these parameters, they are implemented as vectors. The number of connections to the child components is determined by c and h, and the number of connections to parent components is determined by p and h.

The entity aux_fat_tree, shown at the top of Figure 14, is an auxiliary entity used to implement the fat tree structure. The recursive descriptions that will be developed in the next two sections require additional generic constants to control the routing algorithm of each switch instantiated in the structure. The generic constant k is the vertical level number of the bottom row of switches in the fat (sub)tree (counting from 1 at the bottom), i is the back-to-front horizontal index of the bottom row of switches (counting from 0 at the back), and j is the left-to-right horizontal index of the top column of switches (counting from 0 at the left). The architecture body for the fat tree entity, shown at the bottom of Figure 14, declares a component corresponding to the auxiliary fat tree entity, and instantiates it with k set to 1, and i and j set to 0. The details of use of these generic constants will be discussed in the next sections.

4.1. Bottom Up Recursive Description of the Fat Tree

A bottom up recursive architecture declaration for the fat tree is shown in Figure 15. In a fat tree of any height h composed in the bottom up manner, there are p subtrees of height h–1. In this description, they are indexed from 0 to p–1, as defined by the subtype subtree_range. The number of switches at the bottom level is c^{h-1}. They are indexed from 0 to $c^{h-1}-1$, as defined by the subtype switch_range. The signals declared in the architecture body are used to connect the switch instances with the subtree instances, and will be discussed in detail below. The component switch represents the basic switching element from which the fat tree is constructed. It has generic constants c and p representing the child and parent degrees, and k, i and j representing the position coordinates within the fat tree structure needed for the routing algorithm. The component aux_fat_tree is the template that is to be instantiated as subtrees above the bottom level of switches. These instances will be bound to the auxiliary fat tree entity in a separate configuration declaration.

The architecture body includes two conditional generate statements. If the value of the generic constant h is one, a simple fat tree is generated, consisting of a single switch connected directly to the child and parent ports of the entity. This causes the recursion to terminate. If the value of the generic constant h is greater than one, the statement labelled compound_tree generates an array of switches, an array of subtrees of height h – 1, and an array of connections between them, as shown in Figure 16. The child ports of each switch are connected to a slice of length c of the child ports of the fat tree entity. The parent ports of each subtree are connected to a slice of length p^{h-1} of the parent ports of the fat tree entity. The generic constant k of each switch is set using the value of k for the fat tree, representing the vertical level of the bottom row of switches. The generic constant i of each switch is set using the value of i for the fat tree, representing the back-to-front index of the bottom of the fat tree. The generic constant j of each switch, representing the switch's left-to-right index at its level of the entire fat tree, is set using the parameter of the generate statement that creates the switches. The generic constant k of each subtree, representing the vertical level of the bottom of the subtree, is set to one more than that of the enclosing tree. The generic constant i of the subtree, representing the back-to-front index of the bottom of the subtree, is based on the index i of the enclosing tree. This value is multiplied by p, since each of the peer trees behind this tree has p subtrees. The parameter of the generate statement that creates the subtree is then added to the result to form the value passed to the subtree. The generic constant j of the subtree, representing the left-to-right index of the top of the subtree, is set to the same

```
architecture bottom_up_recursive of aux_fat_tree is
    subtype subtree_range is natural range 0 to p – 1;
    subtype switch_range is natural range 0 to c**(h – 1) – 1;

    subtype subtree_connection_vector is connection_vector(switch_range);
    subtype subtree_acknowledge_vector is acknowledge_vector(switch_range);
    type subtree_connection_vector_array is array (subtree_range) of subtree_connection_vector;
    type subtree_acknowledge_vector_array is array (subtree_range) of subtree_acknowledge_vector;
    signal  subtree_receive_msg_from_switch,
            subtree_send_msg_to_switch : subtree_connection_vector_array;
    signal  subtree_receive_ack_from_switch,
            subtree_send_ack_to_switch : subtree_acknowledge_vector_array;

    subtype switch_connection_vector is connection_vector(subtree_range);
    subtype switch_acknowledge_vector is acknowledge_vector(subtree_range);
    type switch_connection_vector_array is array (switch_range) of switch_connection_vector;
    type switch_acknowledge_vector_array is array (switch_range) of switch_acknowledge_vector;
    signal  switch_receive_msg_from_subtree,
            switch_send_msg_to_subtree : switch_connection_vector_array;
    signal  switch_receive_ack_from_subtree,
            switch_send_ack_to_subtree : switch_acknowledge_vector_array;

    component switch
        generic ( c, p : positive;  k : positive;  i, j : natural );
        port ( receive_msg_from_child :   in connection_vector(0 to c – 1);
               send_ack_to_child :        out acknowledge_vector(0 to c – 1);
               send_msg_to_child :        out connection_vector(0 to c – 1);
               receive_ack_from_child :   in acknowledge_vector(0 to c – 1);
               send_msg_to_parent :       out connection_vector(0 to p – 1);
               receive_ack_from_parent :  in acknowledge_vector(0 to p – 1);
               receive_msg_from_parent :  in connection_vector(0 to p – 1);
               send_ack_to_parent :       out acknowledge_vector(0 to p – 1) );
    end component;

    component aux_fat_tree
        generic ( h : natural;  c, p : positive;  k : positive;  i, j : natural );
        port ( receive_msg_from_child :   in connection_vector(0 to c**h – 1);
               send_ack_to_child :        out acknowledge_vector(0 to c**h – 1);
               send_msg_to_child :        out connection_vector(0 to c**h – 1);
               receive_ack_from_child :   in acknowledge_vector(0 to c**h – 1);
               send_msg_to_parent :       out connection_vector(0 to p**h – 1);
               receive_ack_from_parent :  in acknowledge_vector(0 to p**h – 1);
               receive_msg_from_parent :  in connection_vector(0 to p**h – 1);
               send_ack_to_parent :       out acknowledge_vector(0 to p**h – 1) );
    end component;

    constant outer_i : natural := i;
begin
    simple_tree : if h = 1 generate
        the_switch : switch
            generic map ( c, p, k, i, j )
            port map ( receive_msg_from_child =>     receive_msg_from_child,
                       send_ack_to_child =>          send_ack_to_child,
                       send_msg_to_child =>          send_msg_to_child,
                       receive_ack_from_child =>     receive_ack_from_child,
                       send_msg_to_parent =>         send_msg_to_parent,
                       receive_ack_from_parent =>    receive_ack_from_parent,
                       receive_msg_from_parent =>    receive_msg_from_parent,
                       send_ack_to_parent =>         send_ack_to_parent );
    end generate simple_tree;
```

Figure 15. Bottom up recursive architecture body for a fat tree.

```
compound_tree : if h > 1 generate
    switch_array : for j in switch_range generate
        the_switch : switch
            generic map ( c, p, k, i, j )
            port map ( receive_msg_from_child =>     receive_msg_from_child( j * c to (j + 1) * c – 1 ),
                       send_ack_to_child =>          send_ack_to_child( j * c to (j + 1) * c – 1 ),
                       send_msg_to_child =>          send_msg_to_child( j * c to (j + 1) * c – 1 ),
                       receive_ack_from_child =>     receive_ack_from_child( j * c to (j + 1) * c – 1 ),
                       send_msg_to_parent =>         switch_send_msg_to_subtree(j),
                       receive_ack_from_parent =>    switch_receive_ack_from_subtree(j),
                       receive_msg_from_parent =>    switch_receive_msg_from_subtree(j),
                       send_ack_to_parent =>         switch_send_ack_to_subtree(j) );
    end generate switch_array;
    subtree_array : for i in subtree_range generate
        the_subtree : aux_fat_tree
            generic map ( h – 1, c, p, k + 1, outer_i * p + i, j )
            port map ( receive_msg_from_child => subtree_receive_msg_from_switch(i),
                       send_ack_to_child =>      subtree_send_ack_to_switch(i),
                       send_msg_to_child =>      subtree_send_msg_to_switch(i),
                       receive_ack_from_child => subtree_receive_ack_from_switch(i),
                       send_msg_to_parent =>
                         send_msg_to_parent( i * p**(h – 1) to (i + 1) * p**(h – 1) – 1 ),
                       receive_ack_from_parent =>
                         receive_ack_from_parent( i * p**(h – 1) to (i + 1) * p**(h – 1) – 1 ),
                       receive_msg_from_parent =>
                         receive_msg_from_parent( i * p**(h – 1) to (i + 1) * p**(h – 1) – 1 ),
                       send_ack_to_parent =>
                         send_ack_to_parent( i * p**(h – 1) to (i + 1) * p**(h – 1) – 1 ) );
    end generate subtree_array;
    connect_subtree : for i in subtree_range generate
        connect_switch : for j in switch_range generate
            switch_receive_msg_from_subtree(j)(i) <=    subtree_send_msg_to_switch(i)(j);
            subtree_receive_ack_from_switch(i)(j) <=    switch_send_ack_to_subtree(j)(i);
            subtree_receive_msg_from_switch(i)(j) <=    switch_send_msg_to_subtree(j)(i);
            switch_receive_ack_from_subtree(j)(i) <=    subtree_send_ack_to_switch(i)(j);
        end generate connect_switch;
    end generate connect_subtree;
end generate compound_tree;
end bottom_up_recursive;
```

Figure 15. (continued)

as the j index of the enclosing tree. The recursion in the description is completed in a separate configuration declaration, which binds each instance of the fat tree component to the fat tree entity using the bottom up recursive architecture.

Ideally, the connections between the switch array and the subtree array would be formed from rectangular matrices of signals of size $p \times c^{h-1}$. The child connections of subtree i would be connected to row i of the matrices, and the parent connections of switch j would be connected to column j of the matrices. However, VHDL does not allow a row or column of a matrix to be treated as a vector slice. For this reason, each of the matrices for sending and receiving messages, and for sending and receiving acknowledgments, are comprised of two separate arrays of vectors.

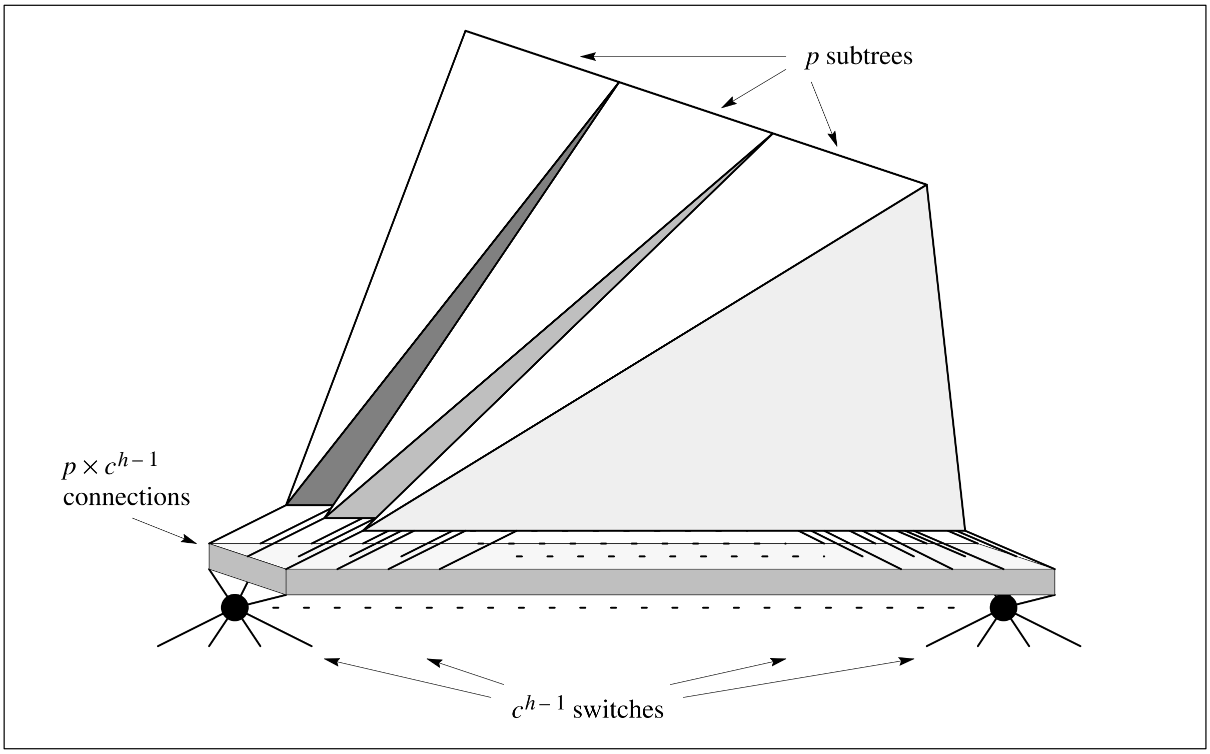

Figure 16. Connections between the bottom row of switches and the subtrees of height $h - 1$.

The four signals subtree_receive_msg_from_switch, subtree_send_ack_to_switch, subtree_send_msg_to_switch and subtree_receive_ack_from_switch are represented by the top surface of the interconnection layer shown in Figure 16. Each of these signals is an array of p elements, and each element is a vector of length c^{h-1}. The vectors in each array are connected to the child ports of the p subtrees. Similarly, the four signals switch_receive_msg_from_subtree, switch_send_ack_to_subtree, switch_send_msg_to_subtree and switch_receive_ack_from_subtree are represented by the bottom surface of the interconnection layer shown in Figure 16. Each of these signals is an array of c^{h-1} elements, and each element is a vector of length p. The vectors in each array are connected to the parent ports of the c^{h-1} switches. The nested generate statements, labelled connect_subtree and connect_switch, create drivers that connect the vertically matching elements of corresponding pairs of signals.

The recursion in the auxiliary fat tree description is completed by the configuration declaration, shown at the top of Figure 17, which binds each instance of the fat tree component to the fat tree entity and the bottom up recursive architecture body. The block configuration for the generate statement simple_tree contains a component configuration which binds the switch instance to a behavioural implementation of a switch entity. For the generate statement compound_tree, the block configuration contains two nested block configurations, one for the array of switches and one for the array of subtrees. The switch component instances are all bound to

```
configuration bottom_up_recursive_aux_fat_tree of aux_fat_tree is
    for bottom_up_recursive
        for simple_tree
            for the_switch : switch use entity work.switch(behavioural);
            end for;
        end for;
        for compound_tree
            for switch_array
                for the_switch : switch use entity work.switch(behavioural);
                end for;
            end for;
            for subtree_array
                for the_subtree : aux_fat_tree
                    use configuration bottom_up_recursive_aux_fat_tree;
                end for;
            end for;
        end for;
    end for;
end bottom_up_recursive_aux_fat_tree;

configuration bottom_up_recursive_fat_tree of fat_tree is
    for aux
        for the_tree : aux_fat_tree use configuration work.bottom_up_recursive_aux_fat_tree;
        end for;
    end for;
end bottom_up_recursive_fat_tree;
```

Figure 17. Top: configuration declaration for the bottom up recursive architecture of the auxiliary fat tree. Bottom: configuration declaration for the main fat tree entity, using the bottom up recursive configuration of the auxiliary fat tree.

a behavioural implementation of a switch entity. The recursive instances of the fanout tree component are each bound using a recursive application of the configuration. Hence, when a fat tree is elaborated, the configuration will be used for each recursive tree instance. The configuration declaration at the bottom of Figure 17 binds the auxiliary fat tree component instance in the aux architecture of the main fat tree using the bottom up recursive configuration of the auxiliary fat tree.

4.2. Top Down Recursive Description of the Fat Tree

A top down recursive architecture declaration for the fat tree can be constructed in a similar way to the bottom up description, and is shown in Figure 18. In a fat tree of any height h composed in the top down manner, there are c subtrees of height h–1. In this description, they are indexed from 0 to c–1, as defined by the subtype subtree_range. The number of switches at the top level is p^{h-1}. They are indexed from 0 to $p^{h-1}-1$, as defined by the subtype switch_range. The signals declared in the architecture body are used to connect the switch instances with the subtree instances, in an analogous manner to the connections in the bottom up architecture.

```
architecture top_down_recursive of aux_fat_tree is
    subtype subtree_range is natural range 0 to c – 1;
    subtype switch_range is natural range 0 to p**(h – 1) – 1;
    subtype subtree_connection_vector is connection_vector(switch_range);
    subtype subtree_acknowledge_vector is acknowledge_vector(switch_range);
    type subtree_connection_vector_array is array (subtree_range) of subtree_connection_vector;
    type subtree_acknowledge_vector_array is array (subtree_range) of subtree_acknowledge_vector;
    signal  subtree_receive_msg_from_switch,
            subtree_send_msg_to_switch : subtree_connection_vector_array;
    signal  subtree_receive_ack_from_switch,
            subtree_send_ack_to_switch : subtree_acknowledge_vector_array;
    subtype switch_connection_vector is connection_vector(subtree_range);
    subtype switch_acknowledge_vector is acknowledge_vector(subtree_range);
    type switch_connection_vector_array is array (switch_range) of switch_connection_vector;
    type switch_acknowledge_vector_array is array (switch_range) of switch_acknowledge_vector;
    signal  switch_receive_msg_from_subtree,
            switch_send_msg_to_subtree : switch_connection_vector_array;
    signal  switch_receive_ack_from_subtree,
            switch_send_ack_to_subtree : switch_acknowledge_vector_array;
    component switch
        generic ( c, p : positive;  k : positive;  i, j : natural );
        port (  receive_msg_from_child :   in connection_vector(0 to c – 1);
                send_ack_to_child :        out acknowledge_vector(0 to c – 1);
                send_msg_to_child :        out connection_vector(0 to c – 1);
                receive_ack_from_child :   in acknowledge_vector(0 to c – 1);
                send_msg_to_parent :       out connection_vector(0 to p – 1);
                receive_ack_from_parent :  in acknowledge_vector(0 to p – 1);
                receive_msg_from_parent :  in connection_vector(0 to p – 1);
                send_ack_to_parent :       out acknowledge_vector(0 to p – 1) );
    end component;
    component aux_fat_tree
        generic ( h : natural;  c, p : positive;  k : positive;  i, j : natural );
        port (  receive_msg_from_child :   in connection_vector(0 to c**h – 1);
                send_ack_to_child :        out acknowledge_vector(0 to c**h – 1);
                send_msg_to_child :        out connection_vector(0 to c**h – 1);
                receive_ack_from_child :   in acknowledge_vector(0 to c**h – 1);
                send_msg_to_parent :       out connection_vector(0 to p**h – 1);
                receive_ack_from_parent :  in acknowledge_vector(0 to p**h – 1);
                receive_msg_from_parent :  in connection_vector(0 to p**h – 1);
                send_ack_to_parent :       out acknowledge_vector(0 to p**h – 1) );
    end component;
    constant outer_j : natural := j;
begin
    simple_tree : if h = 1 generate
        the_switch : switch
            generic map ( c, p, k, i, j )
                port map ( receive_msg_from_child =>   receive_msg_from_child,
                           send_ack_to_child =>        send_ack_to_child,
                           send_msg_to_child =>        send_msg_to_child,
                           receive_ack_from_child =>   receive_ack_from_child,
                           send_msg_to_parent =>       send_msg_to_parent,
                           receive_ack_from_parent =>  receive_ack_from_parent,
                           receive_msg_from_parent =>  receive_msg_from_parent,
                           send_ack_to_parent =>       send_ack_to_parent );
    end generate simple_tree;
```

Figure 18. Top down recursive architecture body for a fat tree.

```
        compound_tree : if h > 1 generate
            switch_array : for i in switch_range generate
                the_switch : switch
                    generic map ( c, p, k + h – 1, i, j )
                    port map ( receive_msg_from_child =>  switch_receive_msg_from_subtree(i),
                               send_ack_to_child =>       switch_send_ack_to_subtree(i),
                               send_msg_to_child =>       switch_send_msg_to_subtree(i),
                               receive_ack_from_child =>  switch_receive_ack_from_subtree(i),
                               send_msg_to_parent =>
                                    send_msg_to_parent( i * p to (i + 1) * p – 1 ),
                               receive_ack_from_parent =>
                                    receive_ack_from_parent( i * p to (i + 1) * p – 1 ),
                               receive_msg_from_parent =>
                                    receive_msg_from_parent( i * p to (i + 1) * p – 1 ),
                               send_ack_to_parent =>
                                    send_ack_to_parent( i * p to (i + 1) * p – 1 ));
            end generate switch_array;
            subtree_array : for j in subtree_range generate
                the_subtree : aux_fat_tree
                    generic map ( h – 1, c, p, k, i, outer_j * c + j )
                    port map (receive_msg_from_child =>
                                    receive_msg_from_child( j * c**(h – 1)  to (j+ 1) * c**(h – 1) – 1 ),
                              send_ack_to_child =>
                                    send_ack_to_child( j * c**(h – 1) to (j + 1) * c**(h – 1) – 1 ),
                              send_msg_to_child =>
                                    send_msg_to_child( j * c**(h – 1) to (j + 1) * c**(h – 1) – 1 ),
                              receive_ack_from_child =>
                                    receive_ack_from_child( j * c**(h – 1)  to (j+ 1) * c**(h – 1) – 1 ),
                              send_msg_to_parent =>       subtree_send_msg_to_switch(j),
                              receive_ack_from_parent =>  subtree_receive_ack_from_switch(j),
                              receive_msg_from_parent =>subtree_receive_msg_from_switch(j),
                              send_ack_to_parent =>       subtree_send_ack_to_switch(j) );
            end generate subtree_array;
            connect_switch : for i in switch_range generate
                connect_subtree : for j in subtree_range generate
                    switch_receive_msg_from_subtree(i)(j) <=   subtree_send_msg_to_switch(j)(i);
                    subtree_receive_ack_from_switch(j)(i) <=   switch_send_ack_to_subtree(i)(j);
                    subtree_receive_msg_from_switch(j)(i) <=   switch_send_msg_to_subtree(i)(j);
                    switch_receive_ack_from_subtree(i)(j) <=   subtree_send_ack_to_switch(j)(i);
                end generate connect_subtree;
            end generate connect_switch;
        end generate compound_tree;
    end top_down_recursive;
```

Figure 18. (continued)

The architecture body contains two conditional generate statements, also analogous to those in the bottom up architecture. The statement labelled compound_tree generates an array of switches, an array of subtrees of height h – 1, and an array of connections between them. The parent ports of each switch are connected to a slice of length p of the parent ports of the fat tree entity, and the child ports of each subtree are connected to a slice of length c^{h-1} of the child ports of the fat tree entity. The generic constant k of each switch is based on the value of k for the fat tree, representing the vertical level of the bottom row of switches, to which is added the height of the fat tree less one. This gives the vertical level of the switches in the entire fat tree. The

```
configuration top_down_recursive_aux_fat_tree of aux_fat_tree is
    for top_down_recursive
        for simple_tree
            for the_switch : switch use entity work.switch(behavioural);
            end for;
        end for;
        for compound_tree
            for switch_array
                for the_switch : switch use entity work.switch(behavioural);
                end for;
            end for;
            for subtree_array
                for the_subtree : aux_fat_tree
                    use configuration top_down_recursive_aux_fat_tree;
                end for;
            end for;
        end for;
    end for;
end top_down_recursive_aux_fat_tree;

configuration top_down_recursive_fat_tree of fat_tree is
    for aux
        for the_tree : aux_fat_tree
            use configuration work.top_down_recursive_aux_fat_tree;
        end for;
    end for;
end top_down_recursive_fat_tree;
```

Figure 19. Top: configuration declaration for the top down recursive architecture of the auxiliary fat tree. Bottom: configuration declaration for the main fat tree entity, using the top down recursive configuration of the auxiliary fat tree.

generic constant i of each switch, representing the switch's beck-to-front index at its level of the entire fat tree, is set using the parameter of the generate statement that creates the switches. The generic constant j of each switch is set using the value of j for the fat tree, representing the left-to-right index of the top of the fat tree. The generic constant k of each subtree, representing the vertical level of the bottom of the subtree, is set to the same as that of the enclosing tree. The generic constant i of the subtree, representing the back-to-front index of the bottom of the subtree, is set to the same as the i index of the enclosing tree. The generic constant j of the subtree, representing the left-to-right index of the top of the subtree, is based on the j index of the enclosing tree. This value is multiplied by c, since each of the peer trees to the left of this tree has c subtrees. The parameter of the generate statement that creates the subtree is then added to form the value passed to the subtree.

Figure 19 shows configuration declarations for the top down recursive description of the auxiliary fat tree and for the main fat tree entity. These are analogous to the configuration declarations for the bottom up description.

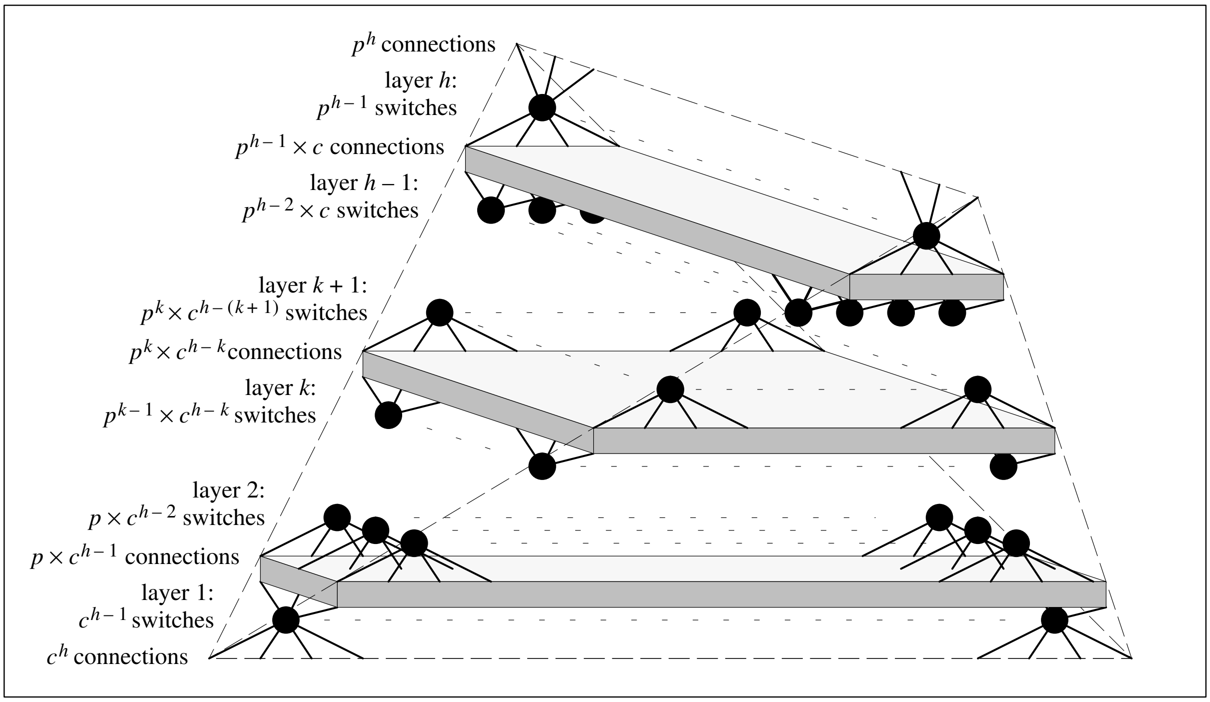

Figure 20. Arrangement of switches and interconnections in a repetitive model of a fat tree.

4.3. Repetitive Description of the Fat Tree

A repetitive description of the fat tree structure is significantly more difficult to design and understand than the recursive description. As with the fanout tree example, the main difficulty lies in the interconnections between the switch elements comprising the structure. As Figure 20 shows, the total number of interconnection pairs needed for sending and receiving messages and acknowledgments (including the parent and child ports of the fat tree) is:

$$S_h = p^h + p^{h-1}c + \cdots + pc^{h-1} + c^h = \begin{cases} \dfrac{p^{h+1} - c^{h+1}}{p - c} & \text{if } p \neq c \\ (h+1)p^h = (h+1)c^h & \text{if } p = c \end{cases}$$

and the total number of switches is S_{h-1}. Ideally, one vector of length S_h would be used for each of the signal sets. The parent switches at each level would connect to slices of length c, whereas the children switches would connect to slices of length p, but with each element in the slice separated by some stride determined by c, p and the position of the switch in the structure. Since this is not possible in VHDL, a pairs of vectors is needed for each set of signals, as shown in Figure 21. One vector is of length $S_h - c^h$, and is divided into slices of length p to connect to the parent ports of each of the switches. It threads the tree from back to front then left to right, starting at the top. The other vector is of length $S_h - p^h$, and is divided into slices of length c to connect to the child ports of each of the switches. It threads the tree from left to right then back to front,

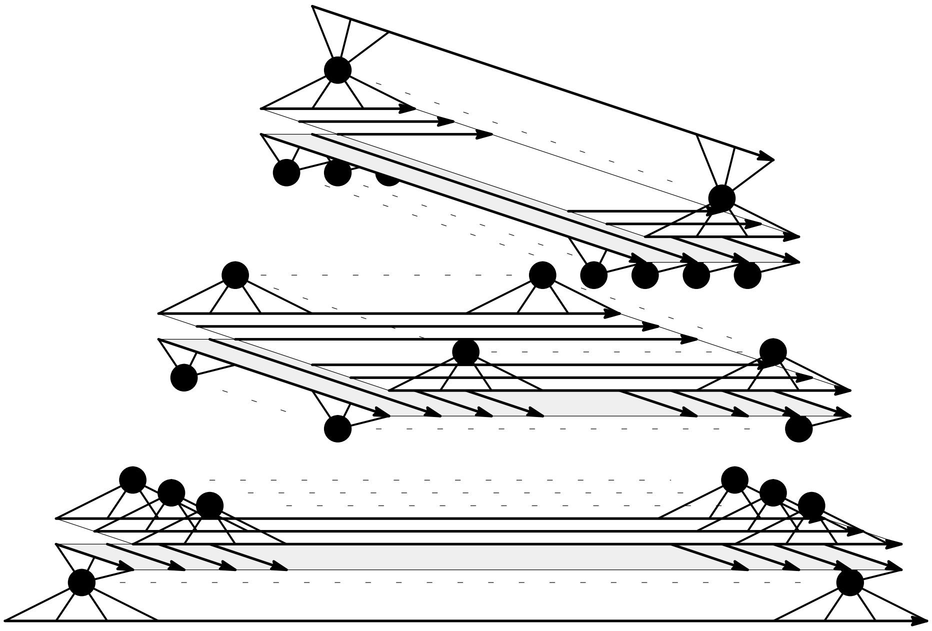

Figure 21. Interconnection vectors threading through the fat tree. One vector, used to connect to the parent ports of each switch, starts at the top of the tree and is aligned from the back to front of the tree parallel to the parent ports. The other vector, used to connect to the child ports of each switch, starts at the bottom of the tree and is aligned from left to right parallel to the child ports.

starting at the bottom. In each layer of the interconnection structure, drivers are created to connect matching elements of the vectors together.

A repetitive architecture body for the fat tree entity is shown in Figure 22. The signals declared in the architecture represent the interconnection vectors described above. The outer generate statement, labelled switch_levels, generates the levels of switches, starting from level 1 at the bottom and repeating upwards to level h. The nested generate statement, labelled rows, generate the horizontal rows of switches at the current level, indexed from 0 at the back to $p^{k-1}-1$ at the front. The innermost generate statement, labelled cols, generates the individual switches within each row, indexed from 0 on the left to $c^{h-k}-1$ at the right. The values of the generate statement parameters k, i and j effectively determine a three dimensional coordinate for each switch created. These coordinates are passed to each switch for use in the routing algorithm.

The complexity in this repetitive description lies in the mapping functions that determine which elements of the vectors each switch must connect to, and which elements of the parent and child connection vectors should be interconnected. The mapping functions are dependent on c, p, h and the coordinates of a switch or connection in the structure.

Consider first a switch at position (k, i, j) within the $p^{k-1} \times c^{h-k}$ array of switches at level k. This switch makes connections to the parent vectors in the layer above it, and to the child vectors

```
architecture repetitive of fat_tree is
    function S ( h : integer ) return natural is
    begin
        if p /= c then
            return (p**(h + 1) – c**(h + 1)) / (p – c);
        elsif h >= 0 then
            return (h + 1) * p**h;
        else  –– p = c and h = –1
            return 0;
        end if;
    end S;
    function Op ( k : natural ) return natural is
    begin
        return p**(k + 1) * S(h – (k + 1));
    end Op;
    function Mp ( k, i, j : natural ) return natural is
    begin
        return Op(k) + j * p**k + i * p;
    end Mp;
    function Oc ( k : natural ) return natural is
    begin
        return c**(h – (k – 2)) * S(k – 2);
    end Oc;
    function Mc ( k, i, j : natural ) return natural is
    begin
        return Oc(k) + i * c**(h – (k – 1)) + j * c;
    end Mc;
    subtype parent_connection_vector is connection_vector(0 to S(h) – c**h – 1);
    subtype parent_acknowledge_vector is acknowledge_vector(0 to S(h) – c**h – 1);
    signal  switch_receive_msg_from_parent,
            switch_send_msg_to_parent : parent_connection_vector;
    signal  switch_receive_ack_from_parent,
            switch_send_ack_to_parent : parent_acknowledge_vector;
    subtype child_connection_vector is connection_vector(0 to S(h) – p**h – 1);
    subtype child_acknowledge_vector is acknowledge_vector(0 to S(h) – p**h – 1);
    signal switch_receive_msg_from_child, switch_send_msg_to_child : child_connection_vector;
    signal switch_receive_ack_from_child, switch_send_ack_to_child : child_acknowledge_vector;
    component switch
        generic ( c, p : positive;  k : positive;  i, j : natural );
        port (  receive_msg_from_child :   in connection_vector(0 to c – 1);
                send_ack_to_child :        out acknowledge_vector(0 to c – 1);
                send_msg_to_child :        out connection_vector(0 to c – 1);
                receive_ack_from_child :   in acknowledge_vector(0 to c – 1);
                send_msg_to_parent :       out connection_vector(0 to p – 1);
                receive_ack_from_parent :  in acknowledge_vector(0 to p – 1);
                receive_msg_from_parent :  in connection_vector(0 to p – 1);
                send_ack_to_parent :       out acknowledge_vector(0 to p – 1) );
    end component;
```

Figure 22. Repetitive architecture body for the fat tree entity.

in the layer below it, as shown in the projections in Figure 23. The parent ports of the switch are connected to slices of length p of the parent vectors starting at element $M_p(k,i,j)$ in each vector, where:

$$M_p(k,i,j) = O_p(k) + jp^k + ip\ .$$

```
begin
    switch_levels : for k in 1 to h generate        –– bottom to top
        rows : for i in 0 to p**(k – 1) – 1 generate       –– back to front
            cols : for j in 0 to c**(h – k) – 1 generate       –– left to right
                the_switch : switch
                    generic map ( c, p, k, i, j )
                    port map (receive_msg_from_child =>
                                    switch_receive_msg_from_child( Mc(k, i, j) to Mc(k, i, j) + c – 1 ),
                              send_ack_to_child =>
                                    switch_send_ack_to_child( Mc(k, i, j) to Mc(k, i, j) + c – 1 ),
                              send_msg_to_child =>
                                    switch_send_msg_to_child( Mc(k, i, j) to Mc(k, i, j) + c – 1 ),
                              receive_ack_from_child =>
                                    switch_receive_ack_from_child( Mc(k, i, j) to Mc(k, i, j) + c – 1 ),
                              send_msg_to_parent =>
                                    switch_send_msg_to_parent( Mp(k, i, j) to Mp(k, i, j) + p – 1 ),
                              receive_ack_from_parent =>
                                    switch_receive_ack_from_parent( Mp(k, i, j) to Mp(k, i, j) + p – 1 ),
                              receive_msg_from_parent =>
                                    switch_receive_msg_from_parent( Mp(k, i, j) to Mp(k, i, j) + p – 1 ),
                              send_ack_to_parent =>
                                    switch_send_ack_to_parent( Mp(k, i, j) to Mp(k, i, j) + p – 1 ) );
            end generate cols;
        end generate rows;
    end generate switch_levels;
    switch_receive_msg_from_child(0 to c**h – 1) <= receive_msg_from_child;
    send_ack_to_child <= switch_send_ack_to_child(0 to c**h – 1);
    send_msg_to_child <= switch_send_msg_to_child(0 to c**h – 1);
    switch_receive_ack_from_child(0 to c**h – 1) <= receive_ack_from_child;
    connection_levels : for k in 1 to h – 1 generate          – – bottom to top
        rows : for i in 0 to p**k – 1 generate                  – – back to front
            cols : for j in 0 to c**(h – k) – 1 generate       – – left to right
                switch_receive_msg_from_child ( Oc(k + 1) + i * c**(h – k) + j ) <=
                            switch_send_msg_to_parent( Op(k) + j * p**k + i );
                switch_receive_ack_from_parent( Op(k) + j * p**k + i ) <=
                            switch_send_ack_to_child( Oc(k + 1) + i * c**(h – k) + j );
                switch_receive_msg_from_parent( Op(k) + j * p**k + i ) <=
                            switch_send_msg_to_child( Oc(k + 1) + i * c**(h – k) + j );
                switch_receive_ack_from_child ( Oc(k + 1) + i * c**(h – k) + j ) <=
                            switch_send_ack_to_parent( Op(k) + j * p**k + i );
            end generate cols;
        end generate rows;
    end generate connection_levels;
    send_msg_to_parent <= switch_send_msg_to_parent(0 to p**h – 1);
    switch_receive_ack_from_parent(0 to p**h – 1) <= receive_ack_from_parent;
    switch_receive_msg_from_parent(0 to p**h – 1) <= receive_msg_from_parent;
    send_ack_to_parent <= switch_send_ack_to_parent(0 to p**h – 1);
end repetitive;
```

Figure 22. (continued)

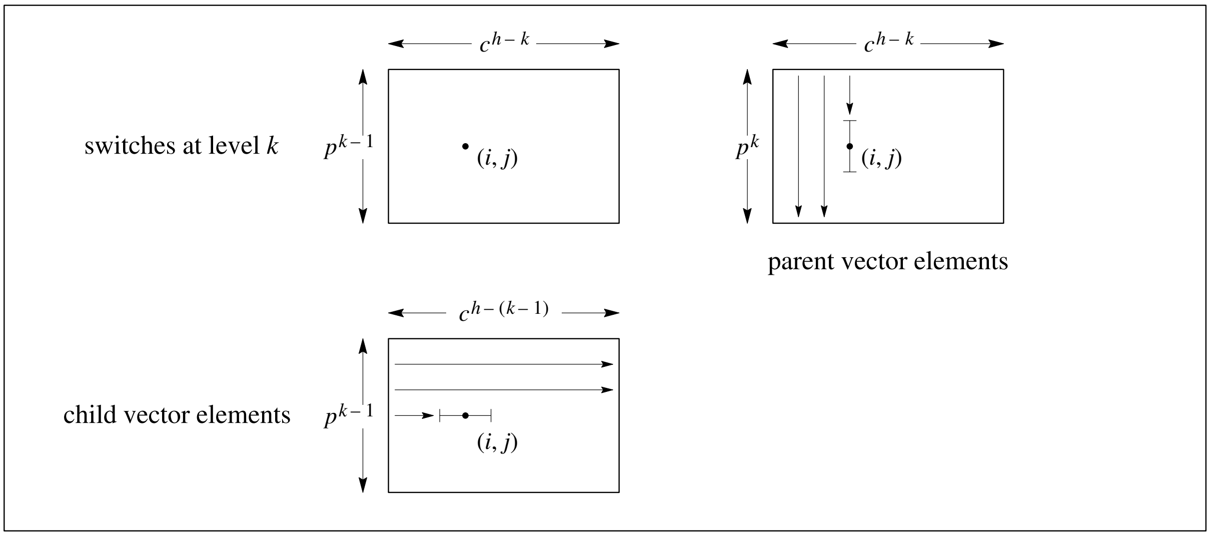

Figure 23. Connections from a switch to the parent vectors in the interconnection layer above and to the child vectors in the interconnection layer below.

The function $O_p(k)$ is an offset function that gives the starting index in the parent vectors for the parent connections of switches at level k. Its value is the number of parent vector elements for switches at levels h down to $k + 1$, as follows:

$$\begin{aligned} O_p(k) &= p^h + p^{h-1}c + p^{h-2}c^2 + \cdots + p^{k+1}c^{h-(k+1)} \\ &= p^{k+1}\left(p^{h-(k+1)} + p^{h-(k+1)-1}c + \cdots + c^{h-(k+1)}\right) \\ &= p^{k+1}S_{h-(k+1)} \end{aligned}$$

(Note that $S_{-1} = 0$.) Similarly, the child ports of the switch are connected to slices of length c of the child connection vectors starting at element $M_c(k, i, j)$, where:

$$M_c(k, i, j) = O_c(k) + ic^{h-(k-1)} + jc\ .$$

The function $O_c(k)$ is an offset function that gives the starting index in the child vectors for the child connections of switches at level k. Its value is the number of child vector elements for switches at levels 1 up to k–1, as follows:

$$\begin{aligned} O_c(k) &= c^h + pc^{h-1} + p^2c^{h-2} + \cdots + p^{k-2}c^{h-(k-2)} \\ &= c^{h-(k-2)}\left(c^{k-2} + pc^{(k-2)-1} + \cdots + p^{k-2}\right) \\ &= c^{h-(k-2)}S_{k-2} \end{aligned}$$

These formulas are represented in the architecture body by the functions Mp, Op, Mc, Oc and S. They are used in the component instantiation statement for the switch to determine the bounds of the slices of the vectors associated with the ports of the switch. The use of these functions

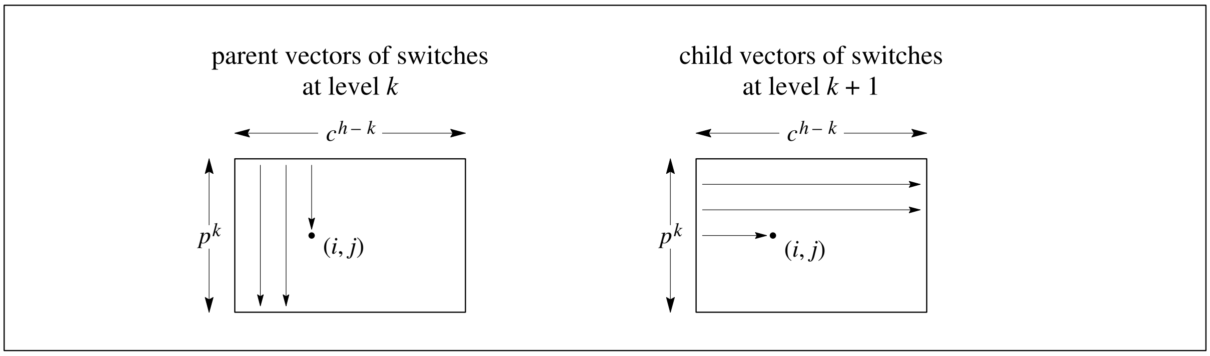

Figure 24. Interconnections between parent and child connection vectors.

in the expressions determining slice bounds for signals is not legal in VHDL-87, although it is legal in VHDL-93. The VHDL-93 form is used here to avoid the added complexity, necessitated in VHDL-87, of writing the expressions "in line". The difficulty is that the expression used to calculate the function S_h depends on whether or not $p = c$. Furthermore, if $h = -1$, the second form (for $p = c$) cannot be used, since it involves exponentiating p to the power -1, which is illegal in VHDL. The solution to these problems in a VHDL-87 version of the description is to replace the single component instantiation statement for the switch with a number of conditional generate statements, corresponding to the various conditions under which S_h must be calculated. Each of these generate statements contains an instantiation of the switch component with ports mapped using the appropriate "in-lined" versions of the expressions for M_p and M_c. This solution is shown in Appendix B.

Now consider the interconnections between the parent signal vectors connected to switches at level k and the child signal vectors connected to switches at level $k + 1$, as shown in Figure 24. The elements of the parent vectors at position (i,j) in the layer have index given by $O_p(k) + jp^k + i$, and the elements of the child vectors at position (i,j) in the layer have index given by $O_c(k + 1) + ic^{h-k} + j$, where O_p and O_c are as defined above. The interconnections between these elements are generated by the generate statement labelled connection_levels and its nested generate statements in the repetitive architecture body. The immediately preceding set of concurrent signal assignments interconnect the child ports of the fat tree with the bottom row of the child signal vectors, and the concurrent signal assignments after the generate statements interconnect the parent ports of the fat tree with the top column of the parent signal vectors.

The restrictions in VHDL-87 on expressions used in signal slice bounds mentioned above also apply to expressions used as signal indices. Such expressions may not include function applications. For this reason, the VHDL-93 form, using the function Op and Oc, is shown here for the concurrent signal assignments that interconnect the vector elements. In a VHDL-87 version they are replaced by a number of conditional generate statements, corresponding to the different ways

```
configuration repetitive_fat_tree of fat_tree is
    for repetitive
        for switch_levels
            for rows
                for cols
                    for the_switch : switch
                        use entity work.switch(behavioural);
                    end for;
                end for;
            end for;
        end for;
    end for;
end repetitive_fat_tree;
```

Figure 25. Configuration declaration for the repetitive fat tree.

in which the functions must be calculated. Each of these contain versions of the concurrent signal assignment statements with signal indices calculated using "in-lined" versions of the expressions for O_p and O_c, as shown in Appendix B.

As with the recursive description of the fanout tree, we need to write a configuration declaration to bind component instances to entities. Figure 25 shows a configuration for the repetitive description. It is somewhat simpler than that for the recursive description, since it only needs to bind the instances of the switch components to the behavioural implementation of the switch entity. Furthermore, the complication of the auxiliary entity is obviated.

5. Comparison of Recursive and Repetitive Descriptions

The fanout tree and the fat tree examples illustrate the advantages of recursive VHDL descriptions of recursive structures over repetitive descriptions. The main difference between the two styles of description lies in the interconnection of the basic components used to construct the structure. In a recursive description, signals are needed to interconnect a small set of components with a small set of recursive instances of the structure. These interconnections are organised into a regular pattern, such as a simple array or matrix. The complexity of the full set of interconnections in the entire structure comes about through the recursive instantiation of the structure, including sub-interconnections, with different parameter values for different instances.

In a repetitive description, on the other hand, all of the signals for the full set of interconnections for the entire structure must be statically declared in the architecture body. The signals must be declared in such a way that their number is dependent upon the parameters that characterise the structure. Since the values of these parameters cannot be known *a priori*, the signals must be declared as arrays, whose size depends on the parameters. A set of mapping functions must then be devised to map ports of instances of the basic components onto slices of these signal arrays. This step may require considerable ingenuity on the part of the designer, and its implementation significantly obscures the design.

Given this complication in a repetitive description, one must ask whether the language is lacking sufficient expressive power. The complication stems from the requirement that the signals be statically declared, and be global to the generate statements that create instances of the basic components. An alternative language design is to allow signals to be generated at elaboration time, in an analogous way to that in which component instances are generated. This would also require that the names of such dynamically generated signals be visible outside of the generate statements that create them, so that they can be referred to in the port association lists of the component instances within other generate statements. Such a name visibility scheme essentially amounts to dynamic scoping, similar to visibility schemes seen in early versions of Lisp and other programming languages. Experience with these languages indicates that there are major software engineering and reliability issues arising from dynamic scoping, which is one of the reasons that languages such as Ada and VHDL use a static scoping scheme. Thus to modify VHDL to incorporate these facilities to ease the implementation of repetitive descriptions of recursive structures would be contrary to the fundamental design requirements of the language.

6. Conclusions

In this report, techniques for developing VHDL descriptions of recursive hardware structures have been examined. Both recursive and repetitive component instantiations were examined for modeling recursive structures. Two example structures, a fanout tree of buffers and a fat tree interconnection network were developed and compared. For both structures, the recursive descriptions were significantly easier to develop, and more clearly express the structure, making them easier to understand. The main difficulty in developing repetitive structures lay in devising a way of interconnecting the basic components comprising the structure. It was shown that the difficulties result from fundamental language design decisions made in VHDL, and that it is not appropriate to modify the language to avoid the difficulties. Hence, in general, the recursive style is preferred over the repetitive style for describing recursive hardware structures.

Acknowledgements

This work was undertaken during the author's study leave visit to the Computer Design Lab at the Department of Electrical and Computer Engineering, University of Cincinnati. The author would like to thank Dr. Philip Wilsey for support and the conducive environment that made the work possible and enjoyable. The author also acknowledges the support of the University of Adelaide through its study leave grants scheme.

References

[1] Institute of Electrical and Electronic Engineers, *IEEE Standard VHDL Language Reference Manual*, IEEE/ANSI Std. 1076–1987, IEEE, New York (1988).

[2] Institute of Electrical and Electronic Engineers, *Draft IEEE Standard VHDL Language Reference Manual*, P1076–1993/C, IEEE, New York (1993).

[3] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, Boston, Massechusetts (1991)

[4] J. D. Morrison and A. S. Clarke, *Ella 2000: a Language for Electronic System Design*, McGraw–Hill (1993).

[5] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, Vol. C-34, No. 10 (October 1985), pp. 892–901.

[6] C. E. Leiserson *et al*, "The Network Architecture of the Connection Machine CM–5," Technical paper, Thinking Machines Corporations, Cambridge, Massechusetts (April 1992).

A. Test Bench for the Fanout Tree

To demonstrate the equivalence and the correctness of behaviour of the two fanout tree descriptions, a test bench was constructed and simulated. The entity and behavioural architecture body for the buffer used in the test bench are shown in Figure 26. The test bench architecture body is shown in Figure 27. It creates two instances of the fanout tree component, one bound

```
entity buf is
    generic ( Tpd : time := 1 ns );
    port ( a : in bit;  y : out bit );
end buf;

architecture behaviour of buf is
begin
    y <= a after Tpd;
end behaviour;
```

Figure 26. Entity declaration and behavioural architecture body for the buffer used in the fanout tree test bench.

```
architecture bench of tree_test is
    constant test_height : positive := 3;
    constant test_degree : positive := 2;
    signal clock : bit;
    signal recursive_result, repetitive_result : bit_vector(0 to test_degree**test_height – 1);
    component fanout_tree
        generic ( h : natural;  d : positive );
        port ( input : in bit;  output : out bit_vector(0 to d**h – 1) );
    end component;
    for recursive_dut : fanout_tree
        use configuration work.recursive_fanout_tree;
    for repetitive_dut : fanout_tree
        use configuration work.repetitive_fanout_tree;
begin
    recursive_dut : fanout_tree
        generic map ( h => test_height, d => test_degree )
        port map ( input => clock, output => recursive_result );
    repetitive_dut : fanout_tree
        generic map ( h => test_height, d => test_degree )
        port map ( input => clock, output => repetitive_result );
    clock_gen : process
    begin
        clock <= '1' after 10 ns, '0' after 20 ns;
        wait for 20 ns;
    end process clock_gen;
    compare_results :
        assert recursive_result = repetitive_result
            report "Results differ";
end bench;
```

Figure 27. Architecture body for the fanout tree test bench.

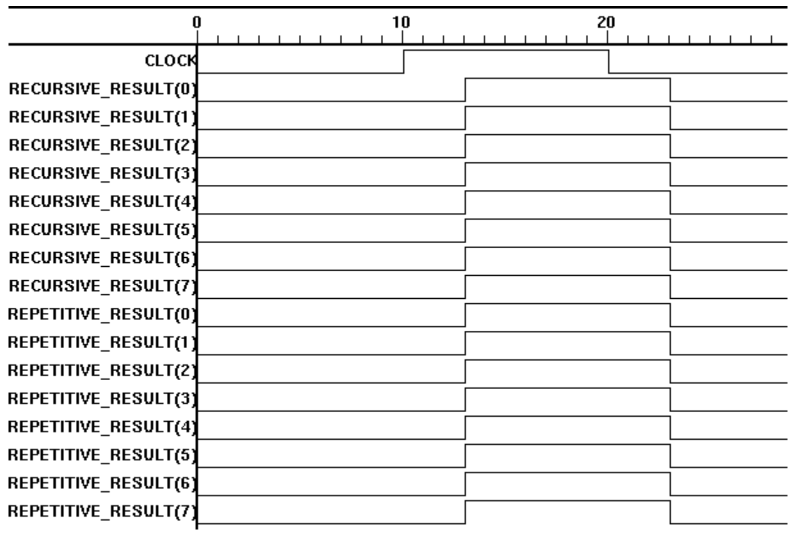

Figure 28. Simulation results for the fanout tree test bench, showing the clock input waveform, and the buffered output waveforms for the recursive and repetitive fanout tree components.

using the recursive configuration of the fanout tree entity, and the other bound using the repetitive configuration. Both instances have a fanout degree of two and a height of three. A stimulus process generates a clock waveform which is connected to the inputs of both fanout trees. An assertion statement is resumed whenever either of the result vectors changes value, and compares the results.

Figure 28 shows the simulation results for the test bench. The output waveforms generated by the recursive and the repetitive descriptions are identical. Figure 29 illustrates the way in which the buffered_input signal is threaded through the tree in the repetitive description. Element 0 is identical to the clock input. Elements 1 and 2 are connected to the outputs of the first row of buffers, hence they lag the input by one propagation delay. Elements 3 to 6 are connected to the outputs of the next row of buffers, and hence lag the input by an additional propagation delay. Finally, elements 7 to 14 are connected to the last row of buffers, and also form the outputs of the entire fanout tree.

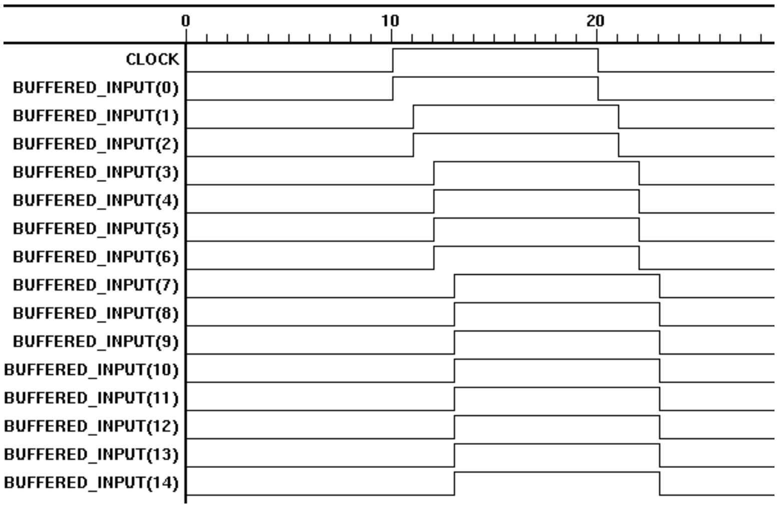

Figure 29. Simulation results showing waveforms on the buffered_input signal in the repetitive description.

B. Test Bench for the Fat Tree

As for the fanout tree, a test bench was constructed and simulated to demonstrate the equivalence and the correctness of behaviour of the descriptions of the fat tree. Figure 30 shows the package declaration containing the definition of the message connection and acknowledgment vector types used in the port specifications of the switch and fat tree entities. Each element of a connection vector is a record containing a message signal for sending messages and a flag to indicate that a new message is waiting. Each element of an acknowledgment vector is a flag indicating that a message has been received. Two entities communicate using a four-phase protocol synchronised with the message waiting and acknowledgment flags. A message record consists of source and destination addresses (indices from left to right across the bottom of the fat tree, starting from 0), and a record of the path taken through the fat tree by the message. The number of hops records the number of switch elements through which the record passed, and the path vector records their coordinates. The message has no substantive content, since it is not relevant

```
use std.textio.line;
package fat_tree_types is
    type coordinate is
        record
            k : positive;
            i, j : natural;
        end record;
    constant max_path_levels : positive := 5;
    constant max_path_hops : positive := 2 * max_path_levels – 1;
    type path_vector is array (1 to max_path_hops) of coordinate;
    type message_record is
        record
            source, destination : natural;
            id : natural;
            hops : natural;
            path : path_vector;
        end record;
    type connection_record is
        record
            message_waiting : boolean;
            message : message_record;
        end record;
    subtype acknowledge is boolean;
    constant default_connection : connection_record
                := (  message_waiting => false,
                      message => ( source => 0, destination => 0, id => 0,
                                   hops => 0, path => (others => (1,0,0)) ) );
    type connection_vector is array (natural range <>) of connection_record;
    type acknowledge_vector is array (natural range <>) of acknowledge;
    procedure write_message ( L : line;  msg : message_record );
end fat_tree_types;
```

Figure 30. Package declaration defining the connection vector type.

```
use work.fat_tree_types.all;
entity switch is
    generic ( c, p : positive;  k : positive;  i, j : natural );
    port (  receive_msg_from_child :   in connection_vector(0 to c – 1);
            send_ack_to_child :        out acknowledge_vector(0 to c – 1);
            send_msg_to_child :        out connection_vector(0 to c – 1);
            receive_ack_from_child :   in acknowledge_vector(0 to c – 1);
            send_msg_to_parent :       out connection_vector(0 to p – 1);
            receive_ack_from_parent :  in acknowledge_vector(0 to p – 1);
            receive_msg_from_parent :  in connection_vector(0 to p – 1);
            send_ack_to_parent :       out acknowledge_vector(0 to p – 1) );
end switch;
```

Figure 31. Entity declaration for the switch used in the fat tree test bench.

to the test. The package also declares a procedure for writing the content of a message record to a line. This is used in the implementation of the test bench.

The entity and behavioural architecture body for the switch used in the test bench are shown in Figure 31 and Figure 32 respectively. The implementation used in the test bench is greatly simplified over a realistic switch. It follows a very simple routing algorithm, and does not handle contention for access to message channels. It assumes that only one message is in transit through the interconnection at any time. These assumptions are adequate to demonstrate the correctness and equivalence of the different descriptions of the fat tree structure, which is the purpose of the test bench.

The switch architecture body consists of a single process that implements the routing algorithm. The process waits until a message arrives on any of the child or parent receive ports. When it is resumed, it searches through the child and parent receive ports to determine which one received the message, and stores a local copy of the message record. The process increments the hop count in the message, and, if the path vector has not yet been filled, adds its coordinates as the next element in the path. Next, the process examines the destination address of the message to determine whether to forward it to a child or a parent. It compares the destination address $d = \langle d_{h-1}, \ldots, d_0 \rangle$ with its own left-to-right horizontal coordinate $j = \langle j_{h-k-1}, \ldots, j_0 \rangle$, each treated as radix-$c$ numbers. If $\langle d_{h-1}, \ldots, d_k \rangle = \langle j_{h-k-1}, \ldots, j_0 \rangle$, the message is destined for a descendent of the switch, so the message is forwarded to child d_{k-1} of the switch. This is the closest ancestor of the destination, and has d_{k-1} as the least significant digit of its left-to-right horizontal index. It and its descendants apply the same routing algorithm, and eventually forward the message down to the destination. If the process finds $\langle d_{h-1}, \ldots, d_k \rangle \neq \langle j_{h-k-1}, \ldots, j_0 \rangle$, the destination is not in the subtree with the switch as the root, so the message is forwarded to a parent. The parent applies the same routing algorithm, so the message will be passed up the tree to a switch

```
architecture behavioural of switch is
begin
    router : process
        variable msg : message_record;
        alias d : natural is msg.destination;
        variable child_msg_received, parent_msg_received : boolean;
        variable msg_received_from : natural;
        variable next_parent : natural := 0;
    begin
        child_msg_received := false;  parent_msg_received := false;
        loop
            wait on receive_msg_from_child, receive_msg_from_parent;
            for index in 0 to c – 1 loop
                if receive_msg_from_child(index).message_waiting then
                    msg := receive_msg_from_child(index).message;
                    child_msg_received := true;
                    msg_received_from := index;
                    exit;
                end if;
            end loop;
            for index in 0 to p – 1 loop
                if receive_msg_from_parent(index).message_waiting then
                    msg := receive_msg_from_parent(index).message;
                    parent_msg_received := true;
                    msg_received_from := index;
                    exit;
                end if;
            end loop;
            exit when child_msg_received or parent_msg_received;
        end loop;
        msg.hops := msg.hops + 1;
        if msg.hops <= max_path_hops then
            msg.path(msg.hops) := coordinate'(k, i, j);
        end if;
        if d / c**k =  j then
            send_msg_to_child( (d mod c**k) / c**(k – 1) ).message <= msg;
            send_msg_to_child( (d mod c**k) / c**(k – 1) ).message_waiting <= true;
            wait until receive_ack_from_child( (d mod c**k) / c**(k – 1) );
            send_msg_to_child( (d mod c**k) / c**(k – 1) ).message_waiting <= false;
            wait until not receive_ack_from_child( (d mod c**k) / c**(k – 1) );
        else
            send_msg_to_parent(next_parent).message <= msg;
            send_msg_to_parent(next_parent).message_waiting <= true;
            wait until receive_ack_from_parent(next_parent);
            send_msg_to_parent(next_parent).message_waiting <= false;
            wait until not receive_ack_from_parent(next_parent);
            next_parent := (next_parent + 1) mod p;
        end if;
        if child_msg_received then
            send_ack_to_child(msg_received_from) <= true;
            wait until not receive_msg_from_child(msg_received_from).message_waiting;
            send_ack_to_child(msg_received_from) <= false;
        end if;
        if parent_msg_received then
            send_ack_to_parent(msg_received_from) <= true;
            wait until not receive_msg_from_parent(msg_received_from).message_waiting;
            send_ack_to_parent(msg_received_from) <= false;
        end if;
    end process router;
end behavioural;
```

Figure 32. Behavioural architecture body for the switch used in the fat tree test bench.

which is a closest common ancestor of the original switch and the destination. From this point, the message is forwarded down the tree to the destination.

This implementation of the switch chooses the parent to which to forward a message in a round robin fashion. The variable next_parent is used each time a parent is chosen, and is then incremented modulo p, ready for the next choice. This simple approach is satisfactory for the test bench. A more realistic implementation of a switch would base its choice on contention for routes through the parents.

Each time a switch forwards a message, it waits for an acknowledgment and completes the four-phase handshaking protocol. The switch then sends an acknowledgment to the child or parent from which it received the message, and completes the handshaking protocol with that switch. Thus as a message is routed through the tree, a chain of switches waiting for acknowledgment is set up along the path taken by the message. When a message reaches the destination, the destination acknowledges receipt, and acknowledgments cascade back along the path to the source. Each switch is then free to accept the next message.

The test bench architecture body is shown in Figure 33. It creates three instances of the fat tree component, one bound using the bottom up recursive configuration, the second bound using the top down configuration, and the third bound using the repetitive configuration. The process message_generator generates a sequence of messages, one for each possible pair of source and addresses. Each message is sent through all three fat trees. The generate statement receivers creates an array of receiver processes, one attached to each address position of each of the three trees. These receive arriving messages, synchronising using the four-phase handshake protocol. For each message they receive, they write a line of trace information to the log file.

The repetitive architecture of the fat tree used in the test bench was a VHDL-87 version derived from the VHDL-93 version, and is shown in Figure 34. The component instantiation statement for the switch component is replicated in each of five conditional generate statements. The first, case_1, represents the case where $p \neq c$, in which the same expression can be used to calculate slice bounds for all switches in the tree. The second generate statement, case_2, represents the case where $p = c$ and the switch is in an intermediate layer (above the bottom row and below the top column). This corresponds to the case in which the function $S(h)$ would be invoked with $h \geq 0$. The remaining three generate statements represent the cases where $p = c$ and the switch is in the bottom row (case_3a), top column (case_3b) or both (case_3c, where the switch is thus the only one in the tree). These correspond to cases in which the function $S(h)$ would be invoked with $h = -1$. In a similar way, the concurrent signal assignment statements interconnecting the signal vectors are replicated in each of the two generate statements case_1 and case_2. These correspond to the same conditions as the generate statements with the same names that replicate

```
use work.fat_tree_types.all;
architecture bench of fat_tree_test is
    constant h : positive := 2;
    constant c : positive := 3;
    constant p : positive := 2;
    subtype node_range is natural range 0 to c**h – 1;
    subtype test_connection_vector is connection_vector(node_range);
    signal  send_msg_to_bottom_up_tree, receive_msg_from_bottom_up_tree,
            send_msg_to_top_down_tree, receive_msg_from_top_down_tree,
            send_msg_to_repetitive_tree, receive_msg_from_repetitive_tree : test_connection_vector;
    subtype test_acknowledge_vector is acknowledge_vector(node_range);
    signal  send_ack_to_bottom_up_tree, receive_ack_from_bottom_up_tree,
            send_ack_to_top_down_tree, receive_ack_from_top_down_tree,
            send_ack_to_repetitive_tree, receive_ack_from_repetitive_tree : test_acknowledge_vector;
    component fat_tree
        generic ( h : natural;  c, p : positive );
        port (  receive_msg_from_child :   in connection_vector(0 to c**h – 1);
                send_ack_to_child :        out acknowledge_vector(0 to c**h – 1);
                send_msg_to_child :        out connection_vector(0 to c**h – 1);
                receive_ack_from_child :   in acknowledge_vector(0 to c**h – 1);
                send_msg_to_parent :       out connection_vector(0 to p**h – 1);
                receive_ack_from_parent :  in acknowledge_vector(0 to p**h – 1) := (others => false);
                receive_msg_from_parent :in connection_vector(0 to p**h – 1)
                                                := (others => default_connection);
                send_ack_to_parent :       out acknowledge_vector(0 to p**h – 1) );
    end component;
    for bottom_up_dut : fat_tree use configuration work.bottom_up_recursive_fat_tree;
    for top_down_dut : fat_tree use configuration work.top_down_recursive_fat_tree;
    for repetitive_dut : fat_tree use configuration work.repetitive_fat_tree;
    use std.textio.all;
    file log_file : text is out ”fat_tree_test.log”;
begin
    bottom_up_dut : fat_tree
        generic map ( h, c, p )
        port map (  receive_msg_from_child => send_msg_to_bottom_up_tree,
                    send_ack_to_child => receive_ack_from_bottom_up_tree,
                    send_msg_to_child => receive_msg_from_bottom_up_tree,
                    receive_ack_from_child => send_ack_to_bottom_up_tree,
                    send_msg_to_parent => open,
                    receive_ack_from_parent => open,
                    receive_msg_from_parent => open,
                    send_ack_to_parent => open );
    top_down_dut : fat_tree
        generic map ( h, c, p )
        port map (  receive_msg_from_child => send_msg_to_top_down_tree,
                    send_ack_to_child => receive_ack_from_top_down_tree,
                    send_msg_to_child => receive_msg_from_top_down_tree,
                    receive_ack_from_child => send_ack_to_top_down_tree,
                    send_msg_to_parent => open,
                    receive_ack_from_parent => open,
                    receive_msg_from_parent => open,
                    send_ack_to_parent => open );
```

Figure 33. Architecture body for the fat tree test bench.

```
repetitive_dut : fat_tree
    generic map ( h, c, p )
    port map (  receive_msg_from_child => send_msg_to_repetitive_tree,
                send_ack_to_child => receive_ack_from_repetitive_tree,
                send_msg_to_child => receive_msg_from_repetitive_tree,
                receive_ack_from_child => send_ack_to_repetitive_tree,
                send_msg_to_parent => open,
                receive_ack_from_parent => open,
                receive_msg_from_parent => open,
                send_ack_to_parent => open );
message_generator : process
    variable msg : message_record;
    variable next_id : natural := 0;
    constant id_modulo : natural := 2**30;      -- something large-ish
    variable L : line;
begin
    write(L, string'("Parameters: h = "));  write(L, h);
    write(L, string'(", c = "));  write(L, c);
    write(L, string'(", p = "));  write(L, p);
    writeline(log_file, L);  writeline(log_file, L);
    write(L, string'("------------------------------------------------------------------------"));
    writeline(log_file, L);  writeline(log_file, L);
    --
    for source in node_range loop
        for destination in node_range loop
            msg := message_record'(  source, destination, next_id,
                                     hops => 0, path => (others => (1,0,0)) );
            next_id := (next_id + 1) mod id_modulo;
            send_msg_to_bottom_up_tree(source).message <= msg;
            send_msg_to_bottom_up_tree(source).message_waiting <= true;
            send_msg_to_top_down_tree(source).message <= msg;
            send_msg_to_top_down_tree(source).message_waiting <= true;
            send_msg_to_repetitive_tree(source).message <= msg;
            send_msg_to_repetitive_tree(source).message_waiting <= true;
            wait until receive_ack_from_bottom_up_tree(source)
                        and receive_ack_from_top_down_tree(source)
                        and receive_ack_from_repetitive_tree(source);
            send_msg_to_bottom_up_tree(source).message_waiting <= false;
            send_msg_to_top_down_tree(source).message_waiting <= false;
            send_msg_to_repetitive_tree(source).message_waiting <= false;
            wait until not receive_ack_from_bottom_up_tree(source)
                            and not receive_ack_from_top_down_tree(source)
                            and not receive_ack_from_repetitive_tree(source);
            writeline(log_file, L);
        end loop;
        write(L, string'("------------------------------------------------------------------------"));
        writeline(log_file, L);  writeline(log_file, L);
    end loop;
    wait;
end process message_generator;
```

Figure 33. (continued)

the switch instantiation statement. The conditions corresponding to case_3a, case_3b and case_3c do not arise in the expressions used to calculate the interconnection vector indices.

Figure 35 shows the log file generated by the test bench with the constants h set to 2, c set to 3 and p set to 2. Figure 36 shows the log file generated with h set to 3, c set to 2 and p set to 2. In both cases the messages are routed through all three trees identically and correctly.

```
    receivers : for index in node_range generate
        bottom_up_receiver : process
            variable L : line;
        begin
            wait until receive_msg_from_bottom_up_tree(index).message_waiting;
            write(L, string'("Receiver ")); write(L, index);
            write(L, string'(" from bottom up tree: "));
            write_message(L, receive_msg_from_bottom_up_tree(index).message);
            writeline(log_file, L);
            send_ack_to_bottom_up_tree(index) <= true;
            wait until not receive_msg_from_bottom_up_tree(index).message_waiting;
            send_ack_to_bottom_up_tree(index) <= false;
        end process bottom_up_receiver;
        top_down_receiver : process
            variable L : line;
        begin
            wait until receive_msg_from_top_down_tree(index).message_waiting;
            write(L, string'("Receiver ")); write(L, index);
            write(L, string'(" from top down tree: "));
            write_message(L, receive_msg_from_top_down_tree(index).message);
            writeline(log_file, L);
            send_ack_to_top_down_tree(index) <= true;
            wait until not receive_msg_from_top_down_tree(index).message_waiting;
            send_ack_to_top_down_tree(index) <= false;
        end process top_down_receiver;
        repetitive_receiver : process
            variable L : line;
        begin
            wait until receive_msg_from_repetitive_tree(index).message_waiting;
            write(L, string'("Receiver ")); write(L, index);
            write(L, string'(" from repetitive tree: "));
            write_message(L, receive_msg_from_repetitive_tree(index).message);
            writeline(log_file, L);
            send_ack_to_repetitive_tree(index) <= true;
            wait until not receive_msg_from_repetitive_tree(index).message_waiting;
            send_ack_to_repetitive_tree(index) <= false;
        end process repetitive_receiver;
    end generate receivers;
end bench;
```

Figure 33. (continued)

```
architecture repetitive of fat_tree is
  function S ( h : integer ) return natural is
  begin
    if p /= c then return (p**(h + 1) – c**(h + 1)) / (p – c);
    elsif h >= 0 then return (h + 1) * p**h;
    else  –– p = c and h = –1
      return 0;
    end if;
  end S;
  subtype parent_connection_vector is connection_vector(0 to S(h) – c**h – 1);
  subtype parent_acknowledge_vector is acknowledge_vector(0 to S(h) – c**h – 1);
  signal switch_receive_msg_from_parent, switch_send_msg_to_parent : parent_connection_vector;
  signal switch_receive_ack_from_parent, switch_send_ack_to_parent : parent_acknowledge_vector;
  subtype child_connection_vector is connection_vector(0 to S(h) – p**h – 1);
  subtype child_acknowledge_vector is acknowledge_vector(0 to S(h) – p**h – 1);
  signal switch_receive_msg_from_child, switch_send_msg_to_child : child_connection_vector;
  signal switch_receive_ack_from_child, switch_send_ack_to_child : child_acknowledge_vector;
  component switch
    generic ( c, p : positive;  k : positive;  i, j : natural );
    port (  receive_msg_from_child : in connection_vector(0 to c – 1);
            send_ack_to_child : out acknowledge_vector(0 to c – 1);
            send_msg_to_child : out connection_vector(0 to c – 1);
            receive_ack_from_child : in acknowledge_vector(0 to c – 1);
            send_msg_to_parent : out connection_vector(0 to p – 1);
            receive_ack_from_parent : in acknowledge_vector(0 to p – 1);
            receive_msg_from_parent : in connection_vector(0 to p – 1);
            send_ack_to_parent : out acknowledge_vector(0 to p – 1) );
  end component;
begin
  switch_levels : for k in 1 to h generate        –– bottom to top
    rows : for i in 0 to p**(k – 1) – 1 generate      –– back to front
      cols : for j in 0 to c**(h – k) – 1 generate      –– left to right
        case_1 : if p /= c generate
          the_switch : switch
            generic map ( c, p, k, i, j )
            port map (receive_msg_from_child =>
                    switch_receive_msg_from_child(c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                    + i * c**(h – (k – 1)) + j * c
                                                  to c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                    + i * c**(h – (k – 1)) + j * c + c – 1 ),
                  send_ack_to_child =>
                    switch_send_ack_to_child(  c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                + i * c**(h – (k – 1)) + j * c
                                              to c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                + i * c**(h – (k – 1)) + j * c + c – 1 ),
                  send_msg_to_child =>
                    switch_send_msg_to_child( c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                + i * c**(h – (k – 1)) + j * c
                                              to c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                + i * c**(h – (k – 1)) + j * c + c – 1 ),
                  receive_ack_from_child =>
                    switch_receive_ack_from_child( c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                    + i * c**(h – (k – 1)) + j * c
                                                  to c**(h – (k – 2)) * ((p**(k – 1) – c**(k – 1)) / (p – c))
                                                    + i * c**(h – (k – 1)) + j * c + c – 1 ),
                  send_msg_to_parent =>
                    switch_send_msg_to_parent(  p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p
                                                to p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p + p – 1 ),
                  receive_ack_from_parent =>
                    switch_receive_ack_from_parent(  p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p
                                                      to p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c))+ j * p**k + i * p + p – 1 ),
                  receive_msg_from_parent =>
                    switch_receive_msg_from_parent(p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p
                                            to p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p + p – 1 ),
                  send_ack_to_parent =>
                    switch_send_ack_to_parent(p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p
                                        to p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i * p + p – 1 ) );
        end generate case_1;
```

Figure 34. VHDL-87 version of the repetitive architecture body for the fat tree.

```
case_2 : if p = c and k /= 1 and k /= h generate  -- switch in an intermediate plane
  the_switch : switch
    generic map ( c, p, k, i, j )
    port map (receive_msg_from_child =>
                switch_receive_msg_from_child(c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c
                                to c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c + c – 1 ),
              send_ack_to_child =>
                switch_send_ack_to_child(  c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c
                                to c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c + c – 1 ),
              send_msg_to_child =>
              switch_send_msg_to_child( c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c
                                to c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c + c – 1 ),
              receive_ack_from_child =>
                switch_receive_ack_from_child( c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c
                                to c**(h – (k – 2)) * ((k – 1) * c**(k – 2)) + i * c**(h – (k – 1)) + j * c + c – 1 ),
              send_msg_to_parent =>
                switch_send_msg_to_parent(  p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p
                                to p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p + p – 1 ),
              receive_ack_from_parent =>
                switch_receive_ack_from_parent(  p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p
                                to p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p + p – 1 ),
              receive_msg_from_parent =>
                switch_receive_msg_from_parent(p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p
                                to p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p + p – 1 ),
              send_ack_to_parent =>
                switch_send_ack_to_parent(p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p
                                to p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i * p + p – 1 ) );
end generate case_2;
case_3a : if p = c and k = 1 and h /= 1 generate  -- switch in bottom row of tree h > 1
  the_switch : switch
    generic map ( c, p, k, i, j )
    port map (receive_msg_from_child => switch_receive_msg_from_child( j * c to j * c + c – 1 ),
                send_ack_to_child => switch_send_ack_to_child( j * c to j * c + c – 1 ),
                send_msg_to_child => switch_send_msg_to_child( j * c to j * c + c – 1 ),
                receive_ack_from_child => switch_receive_ack_from_child( j * c to j * c + c – 1 ),
                send_msg_to_parent =>
                  switch_send_msg_to_parent(  p**2 * ((h – 1) * p**(h – 2)) + j * p
                                to p**2 * ((h – 1) * p**(h – 2)) + j * p + p – 1 ),
                receive_ack_from_parent =>
                  switch_receive_ack_from_parent(  p**2 * ((h – 1) * p**(h – 2)) + j * p
                                to p**2 * ((h – 1) * p**(h – 2)) + j * p + p – 1 ),
                receive_msg_from_parent =>
                  switch_receive_msg_from_parent(p**2 * ((h – 1) * p**(h – 2)) + j * p
                                to p**2 * ((h – 1) * p**(h – 2)) + j * p + p – 1 ),
                send_ack_to_parent =>
                  switch_send_ack_to_parent(p**2 * ((h – 1) * p**(h – 2)) + j * p
                                to p**2 * ((h – 1) * p**(h – 2)) + j * p + p – 1 ) );
end generate case_3a;
case_3b : if p = c and k = h and h /= 1 generate  -- switch in top col of tree h > 1
  the_switch : switch
    generic map ( c, p, k, i, j )
    port map (receive_msg_from_child =>
                  switch_receive_msg_from_child(c**2 * ((h – 1) * c**(h – 2)) + i * c
                                to c**2 * ((h – 1) * c**(h – 2)) + i * c + c – 1 ),
                send_ack_to_child =>
                  switch_send_ack_to_child(  c**2 * ((h – 1) * c**(h – 2)) + i * c
                                to c**2 * ((h – 1) * c**(h – 2)) + i * c + c – 1 ),
                send_msg_to_child =>
                  switch_send_msg_to_child( c**2 * ((h – 1) * c**(h – 2)) + i * c
                                to c**2 * ((h – 1) * c**(h – 2)) + i * c + c – 1 ),
                receive_ack_from_child =>
                  switch_receive_ack_from_child( c**2 * ((h – 1) * c**(h – 2)) + i * c
                                to c**2 * ((h – 1) * c**(h – 2)) + i * c + c – 1 ),
                send_msg_to_parent => switch_send_msg_to_parent( i * p to  i * p+ p – 1 ),
                receive_ack_from_parent => switch_receive_ack_from_parent( i * p to  i * p+ p – 1 ),
                receive_msg_from_parent => switch_receive_msg_from_parent( i * p to  i * p+ p – 1 ),
                send_ack_to_parent => switch_send_ack_to_parent( i * p to  i * p+ p – 1 ) );
end generate case_3b;
```

Figure 34. (continued)

```
            case_3c : if p = c and h = 1 generate      –– switch is the only one in the tree
              the_switch : switch
                generic map ( c, p, k, i, j )
                port map (receive_msg_from_child => switch_receive_msg_from_child( 0 to c – 1 ),
                          send_ack_to_child => switch_send_ack_to_child( 0 to c – 1 ),
                          send_msg_to_child => switch_send_msg_to_child( 0 to c – 1 ),
                          receive_ack_from_child => switch_receive_ack_from_child( 0 to c – 1 ),
                          send_msg_to_parent => switch_send_msg_to_parent( 0 to p – 1 ),
                          receive_ack_from_parent => switch_receive_ack_from_parent( 0 to p – 1 ),
                          receive_msg_from_parent => switch_receive_msg_from_parent( 0 to p – 1 ),
                          send_ack_to_parent => switch_send_ack_to_parent( 0 to p – 1 ) );
            end generate case_3c;
          end generate cols;
        end generate rows;
      end generate switch_levels;
      switch_receive_msg_from_child(0 to c**h – 1) <= receive_msg_from_child;
      send_ack_to_child <= switch_send_ack_to_child(0 to c**h – 1);
      send_msg_to_child <= switch_send_msg_to_child(0 to c**h – 1);
      switch_receive_ack_from_child(0 to c**h – 1) <= receive_ack_from_child;
      connection_levels : for k in 1 to h – 1 generate    –– bottom to top
        rows : for i in 0 to p**k – 1 generate        –– back to front
          cols : for j in 0 to c**(h – k) – 1 generate      –– left to right
            case_1 : if p /= c generate
              switch_receive_msg_from_child ( c**(h – (k – 1)) * ((p**k – c**k) / (p – c)) + i * c**(h – k) + j ) <=
                switch_send_msg_to_parent( p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i );
              switch_receive_ack_from_parent( p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i ) <=
                switch_send_ack_to_child( c**(h – (k – 1)) * ((p**k – c**k) / (p – c)) + i * c**(h – k) + j );
              switch_receive_msg_from_parent( p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i ) <=
                switch_send_msg_to_child( c**(h – (k – 1)) * ((p**k – c**k) / (p – c)) + i * c**(h – k) + j );
              switch_receive_ack_from_child ( c**(h – (k – 1)) * ((p**k – c**k) / (p – c)) + i * c**(h – k) + j ) <=
                switch_send_ack_to_parent( p**(k + 1) * ((p**(h – k) – c**(h – k)) / (p – c)) + j * p**k + i );
            end generate case_1;
            case_2 : if p = c generate
              switch_receive_msg_from_child ( c**(h – (k – 1)) * (k * c**(k – 1)) + i * c**(h – k) + j ) <=
                switch_send_msg_to_parent( p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i );
              switch_receive_ack_from_parent( p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i ) <=
                switch_send_ack_to_child( c**(h – (k – 1)) * (k * c**(k – 1)) + i * c**(h – k) + j );
              switch_receive_msg_from_parent( p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i ) <=
                switch_send_msg_to_child( c**(h – (k – 1)) * (k * c**(k – 1)) + i * c**(h – k) + j );
              switch_receive_ack_from_child ( c**(h – (k – 1)) * (k * c**(k – 1)) + i * c**(h – k) + j ) <=
                switch_send_ack_to_parent( p**(k + 1) * ((h – k) * p**(h – (k + 1))) + j * p**k + i );
            end generate case_2;
          end generate cols;
        end generate rows;
      end generate connection_levels;
      send_msg_to_parent <= switch_send_msg_to_parent(0 to p**h – 1);
      switch_receive_ack_from_parent(0 to p**h – 1) <= receive_ack_from_parent;
      switch_receive_msg_from_parent(0 to p**h – 1) <= receive_msg_from_parent;
      send_ack_to_parent <= switch_send_ack_to_parent(0 to p**h – 1);
    end repetitive;
```

Figure 34. (continued)

```
Parameters: h = 2, c = 3, p = 2
________________________________________________________________
Receiver 0 from bottom up tree:  message 0 from 0 to 0 via 1 hop: (1,0,0)
Receiver 0 from top down tree:   message 0 from 0 to 0 via 1 hop: (1,0,0)
Receiver 0 from repetitive tree: message 0 from 0 to 0 via 1 hop: (1,0,0)
Receiver 1 from bottom up tree:  message 1 from 0 to 1 via 1 hop: (1,0,0)
Receiver 1 from top down tree:   message 1 from 0 to 1 via 1 hop: (1,0,0)
Receiver 1 from repetitive tree: message 1 from 0 to 1 via 1 hop: (1,0,0)
Receiver 2 from bottom up tree:  message 2 from 0 to 2 via 1 hop: (1,0,0)
Receiver 2 from top down tree:   message 2 from 0 to 2 via 1 hop: (1,0,0)
Receiver 2 from repetitive tree: message 2 from 0 to 2 via 1 hop: (1,0,0)
Receiver 3 from bottom up tree:  message 3 from 0 to 3 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 3 from top down tree:   message 3 from 0 to 3 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 3 from repetitive tree: message 3 from 0 to 3 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 4 from bottom up tree:  message 4 from 0 to 4 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 4 from top down tree:   message 4 from 0 to 4 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 4 from repetitive tree: message 4 from 0 to 4 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 5 from bottom up tree:  message 5 from 0 to 5 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 5 from top down tree:   message 5 from 0 to 5 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 5 from repetitive tree: message 5 from 0 to 5 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 6 from bottom up tree:  message 6 from 0 to 6 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 6 from top down tree:   message 6 from 0 to 6 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 6 from repetitive tree: message 6 from 0 to 6 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 7 from bottom up tree:  message 7 from 0 to 7 via 3 hops: (1,0,0) (2,0,0) (1,0,2)
Receiver 7 from top down tree:   message 7 from 0 to 7 via 3 hops: (1,0,0) (2,0,0) (1,0,2)
Receiver 7 from repetitive tree: message 7 from 0 to 7 via 3 hops: (1,0,0) (2,0,0) (1,0,2)
Receiver 8 from bottom up tree:  message 8 from 0 to 8 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 8 from top down tree:   message 8 from 0 to 8 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 8 from repetitive tree: message 8 from 0 to 8 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
________________________________________________________________
Receiver 0 from bottom up tree:  message 9 from 1 to 0 via 1 hop: (1,0,0)
Receiver 0 from top down tree:   message 9 from 1 to 0 via 1 hop: (1,0,0)
Receiver 0 from repetitive tree: message 9 from 1 to 0 via 1 hop: (1,0,0)
Receiver 1 from bottom up tree:  message 10 from 1 to 1 via 1 hop: (1,0,0)
Receiver 1 from top down tree:   message 10 from 1 to 1 via 1 hop: (1,0,0)
Receiver 1 from repetitive tree: message 10 from 1 to 1 via 1 hop: (1,0,0)
Receiver 2 from bottom up tree:  message 11 from 1 to 2 via 1 hop: (1,0,0)
Receiver 2 from top down tree:   message 11 from 1 to 2 via 1 hop: (1,0,0)
Receiver 2 from repetitive tree: message 11 from 1 to 2 via 1 hop: (1,0,0)
Receiver 3 from bottom up tree:  message 12 from 1 to 3 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 3 from top down tree:   message 12 from 1 to 3 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 3 from repetitive tree: message 12 from 1 to 3 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 4 from bottom up tree:  message 13 from 1 to 4 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 4 from top down tree:   message 13 from 1 to 4 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 4 from repetitive tree: message 13 from 1 to 4 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 5 from bottom up tree:  message 14 from 1 to 5 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 5 from top down tree:   message 14 from 1 to 5 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 5 from repetitive tree: message 14 from 1 to 5 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 6 from bottom up tree:  message 15 from 1 to 6 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 6 from top down tree:   message 15 from 1 to 6 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 6 from repetitive tree: message 15 from 1 to 6 via 3 hops: (1,0,0) (2,1,0) (1,0,2)
Receiver 7 from bottom up tree:  message 16 from 1 to 7 via 3 hops: (1,0,0) (2,0,0) (1,0,2)
Receiver 7 from top down tree:   message 16 from 1 to 7 via 3 hops: (1,0,0) (2,0,0) (1,0,2)
Receiver 7 from repetitive tree: message 16 from 1 to 7 via 3 hops: (1,0,0) (2,0,0) (1,0,2)
. . .
```

Figure 35. Extract from the log file for the test bench with h set to 2, c set to 3 and p set to 2.

```
Parameters: h = 3, c = 2, p = 2
_______________________________________________________________________
Receiver 0 from bottom up tree:  message 0 from 0 to 0 via 1 hop: (1,0,0)
Receiver 0 from top down tree:   message 0 from 0 to 0 via 1 hop: (1,0,0)
Receiver 0 from repetitive tree: message 0 from 0 to 0 via 1 hop: (1,0,0)
Receiver 1 from bottom up tree:  message 1 from 0 to 1 via 1 hop: (1,0,0)
Receiver 1 from top down tree:   message 1 from 0 to 1 via 1 hop: (1,0,0)
Receiver 1 from repetitive tree: message 1 from 0 to 1 via 1 hop: (1,0,0)
Receiver 2 from bottom up tree:  message 2 from 0 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 2 from top down tree:   message 2 from 0 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 2 from repetitive tree: message 2 from 0 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 3 from bottom up tree:  message 3 from 0 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 3 from top down tree:   message 3 from 0 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 3 from repetitive tree: message 3 from 0 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 4 from bottom up tree:  message 4 from 0 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Receiver 4 from top down tree:   message 4 from 0 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Receiver 4 from repetitive tree: message 4 from 0 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Receiver 5 from bottom up tree:  message 5 from 0 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Receiver 5 from top down tree:   message 5 from 0 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Receiver 5 from repetitive tree: message 5 from 0 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Receiver 6 from bottom up tree:  message 6 from 0 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Receiver 6 from top down tree:   message 6 from 0 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Receiver 6 from repetitive tree: message 6 from 0 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Receiver 7 from bottom up tree:  message 7 from 0 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
Receiver 7 from top down tree:   message 7 from 0 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
Receiver 7 from repetitive tree: message 7 from 0 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
_______________________________________________________________________
Receiver 0 from bottom up tree:  message 8 from 1 to 0 via 1 hop: (1,0,0)
Receiver 0 from top down tree:   message 8 from 1 to 0 via 1 hop: (1,0,0)
Receiver 0 from repetitive tree: message 8 from 1 to 0 via 1 hop: (1,0,0)
Receiver 1 from bottom up tree:  message 9 from 1 to 1 via 1 hop: (1,0,0)
Receiver 1 from top down tree:   message 9 from 1 to 1 via 1 hop: (1,0,0)
Receiver 1 from repetitive tree: message 9 from 1 to 1 via 1 hop: (1,0,0)
Receiver 2 from bottom up tree:  message 10 from 1 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 2 from top down tree:   message 10 from 1 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 2 from repetitive tree: message 10 from 1 to 2 via 3 hops: (1,0,0) (2,0,0) (1,0,1)
Receiver 3 from bottom up tree:  message 11 from 1 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 3 from top down tree:   message 11 from 1 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 3 from repetitive tree: message 11 from 1 to 3 via 3 hops: (1,0,0) (2,1,0) (1,0,1)
Receiver 4 from bottom up tree:  message 12 from 1 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Receiver 4 from top down tree:   message 12 from 1 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Receiver 4 from repetitive tree: message 12 from 1 to 4 via 5 hops: (1,0,0) (2,0,0) (3,0,0) (2,0,1) (1,0,2)
Receiver 5 from bottom up tree:  message 13 from 1 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Receiver 5 from top down tree:   message 13 from 1 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Receiver 5 from repetitive tree: message 13 from 1 to 5 via 5 hops: (1,0,0) (2,1,0) (3,2,0) (2,1,1) (1,0,2)
Receiver 6 from bottom up tree:  message 14 from 1 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Receiver 6 from top down tree:   message 14 from 1 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Receiver 6 from repetitive tree: message 14 from 1 to 6 via 5 hops: (1,0,0) (2,0,0) (3,1,0) (2,0,1) (1,0,3)
Receiver 7 from bottom up tree:  message 15 from 1 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
Receiver 7 from top down tree:   message 15 from 1 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
Receiver 7 from repetitive tree: message 15 from 1 to 7 via 5 hops: (1,0,0) (2,1,0) (3,3,0) (2,1,1) (1,0,3)
. . .
```

Figure 36. Extract from the log file for the test bench with h set to 3, c set to 2 and p set to 2.