Advanced Module Instantiation

The default method of instantiating modules is individually and explicitly. This method poses serious limitations as we try to exploit hierarchy (building large designs out of several smaller, simpler components). Consider the example of building an eight-bit adder out of eight single-bit adders. Our design methodology thus far would yield something like the following:

```verilog
module eight_bit_addr( input [7:0] a, b,
                       input cin,
                       output [7:0] sum,
                       output cout);

logic [6:0] carries;

one_bit_addr addr_8 [7:0] (a[0], b[0], cin, sum[0], carries[0]);
one_bit_addr addr_8 [7:0] (a[1], b[1], cin, sum[1], carries[1]);
one_bit_addr addr_8 [7:0] (a[2], b[2], cin, sum[2], carries[2]);
one_bit_addr addr_8 [7:0] (a[3], b[3], cin, sum[3], carries[3]);
one_bit_addr addr_8 [7:0] (a[4], b[4], cin, sum[4], carries[4]);
one_bit_addr addr_8 [7:0] (a[5], b[5], cin, sum[5], carries[5]);
one_bit_addr addr_8 [7:0] (a[6], b[6], cin, sum[6], carries[6]);
one_bit_addr addr_8 [7:0] (a[7], b[7], cin, sum[7], carries[7]);
endmodule
```

The code above is bulky. It’s manageable right now, but if we wanted to change the design to a 64-bit adder, it starts becoming a significant issue to explicitly instantiate each module, both in terms of code size and the likelihood of typing errors. Verilog-2001 (and to a greater extent, SystemVerilog) offers two powerful constructs to solve these issues: array instantiation and generate blocks.

Array Instantiation

Array instantiation allows a fixed number of modules whose ports are connected by logically contiguous data segments to be instantiated in a single statement. The above adder example can be replaced by the following:

```verilog
module eight_bit_addr( input [7:0] a, b,
                       input cin,
                       output [7:0] sum,
                       output cout);

logic [6:0] carries;

one_bit_addr addr_8 [7:0] (a, b, cin({carries, cin}),
                          .sum(sum), .cout({cout, carries}) );
```
The syntax for array instantiation is as follows:

```
module_name instance_name [top_index : bottom_index] ( 
  .port_name(array or concatenation of signals) …);
```

Key points:
- Array instantiation, like regular instantiation, must exist within a module definition, but outside any procedural block (begin … end blocks)
- The array indexes (top_index and bottom_index above) must be known values at compile time, i.e. using hard-coded values or (better yet) parameters or `define statements
- Following the convention of EECS 470, top_index should be the larger index, and bottom_index should be 0.

When connecting input signal S to a port of an N-array instantiated module, S may be one of two widths, and the compiler will decide how to connect it accordingly.

- Signal width may equal the width of the individual port, multiplied by N. In this case, the compiler will partition S into N equal-sized arrays and feed each one as input to one of the modules. In the adder example, the compiler divides “a” into eight 1-bit signals and addr_8[7] is connected to a[7], addr_8[6] to a[6], etc.

- Signal width may equal the individual port width. In this case, the compiler will make N copies of S and connect one to each port of the array. For example:

```
module sub_module(input [1:0] x, output y);
  …
endmodule

module top();
  logic [1:0] in = 2'b10;
  logic [3:0] out;
  sub_module s1 [3:0] (in, out);
endmodule
```

In this example, in will be duplicated 4 times (8'b10101010) and each module will receive 2'b10 as an input. Each submodule will output a 1-bit signal, which will be concatenated together and driven to “out”.

```
endmodule
```
Generate Blocks

Another more powerful method of instantiation is by using generate blocks. Generate blocks, like regular and array instantiations, exist within a module definition but outside any procedural statement. However, generate blocks allow procedural like statements (like “for” and “if”) within them. Here’s an example instantiation of the 8-bit adder:

```verbatim
parameter N = 8;

module eight_bit_addr( input [N-1:0] a,b, 
        input cin, 
        output [N-1:0] sum, 
        output cout);

logic [N:0] carries;

generate       // start of generate block
    genvar i;
    // special variable that doesn’t represent any real hardware, 
    // just used for evaluation purposes
    for (i=0; i<N; i=i+1) begin
        one_bit_addr sub_addr( .a(a[i]), .b(b[i]),
                .cin(carries[i]),
                .sum(sum[i]),
                .cout(carries[i+1]));
    end

assign carries[0] = cin;
assign cout = carries[N];

generate       //end of generate block
endmodule
```

As a design is being compiled (after syntax checks and before simulation), generate blocks are “evaluated” by the compiler. During this time, for-loops are unwrapped and if-statements are evaluated to see which modules should be instantiated with what connections. Remember, generate blocks will only be evaluated once before simulation, not periodically like procedural blocks. Accordingly, all relevant values (such as N in the above example) must be known at compile time. Genvars may be used to iterate through loops or evaluate if-statements, but dynamic signals cannot be used in anyway other than as connections to modules.
The above example shows how generate blocks can be used interchangeably with array instantiation. However, there are some cases when array instantiation is impractical and generate blocks are clearly a better option. Consider if we wanted to design a generic N-bit priority selector out of 2-1 priority selectors. Using some EECS 281 data structure analysis, we might come up with a tree design like the following:

2-1 priority selector

In words, this N-bit priority selector uses \([N-1]\) 2 bit priority selectors, where output of \(ps2[i]\) “gnt” is connected to “en” of \(ps2[2*i+1]\) and \(ps2[i]\), and “req_up” of \(ps2[2*i]\) and \(ps2[2*i+1]\) is connected to “req” of \(ps[i]\).

Using array instantiation to implement this seems tricky. Particularly when we try to figure out which bit of its parent “gnt” signal each \(ps2\) submodule should use as enable. There’s no obvious way to organize a linear array of signals to neatly accomplish this. Generate blocks, however, allow us to define the problem in a procedural way that can be evaluated at compile time. For each \(ps2\) submodule, we can define a local parameter that will be evaluated to determine if the submodule is a left child or a right child of its parent, which can be used to determine if it should look at the least significant or most significant bit of its parent’s “gnt” signal as an enable signal. We get the following Verilog:

```verilog
// Simple 2-bit priority selector
```
module ps2 (input [1:0] req,  
            input en,  
            output logic [1:0] gnt,  
            output logic req_up);

    assign req_up = req[1] | req[0];
    assign gnt[1] = en & req[1];
    assign gnt[0] = en & req[0] & ~req[1];
endmodule

// Generic N-bit priority selector
module ps_top (input [N-1:0] req,  
               input en,  
               output logic [N-1:0] gnt);

    // req up of lower ps connects to req of higher
    logic [N-1:0] [1:0] sub_reqs;
    // gnt of higher ps connects to enable of lower
    logic [N-1:0] [1:0] sub_gnts;

genenerate
    // Instantiate N-1 ps2 submodules
    for (genvar i=1; i<N; i++) begin
        // the following is a local parameter defined at compile-time to
        // distinguish left subtrees from right subtrees via the least
        // significant bit of i

        // i = odd number -> left sub-tree of parent
        localparam left_right = i[0];
        ps2 sub_ps2 (.req (sub_reqs[i]),  
                      .en (sub_gnts[i/2][left_right]),  
                      .gnt (sub_gnts[i]),  
                      .req_up (sub_reqs[i/2][left_right])
                      );
    end
endgenerate

    // connect top-level requests to bottom layer selectors
    assign sub_reqs[ N-1 : N/2 ] = req;
    // connect enable to top layer selector
    assign sub_gnts[0][1] = en;
    // connect bottom layer selectors to top-level grants
    assign gnt = sub_gnts[ N-1 : N/2 ];
endmodule

Remember that code with generate loops are functionally no different than code with regular instantiation. Generate blocks simply provide a condensed and convenient way to describe repetitive hardware. Be aware of what will actually translate to hardware, and what will just be used by the elaboration tools. In general, any logic with only a genvar as in input will only
be implemented by the software during elaboration, not in the final hardware. We can therefore safely use built-in functions like $clog2$ that would be rather complicated to implement in hardware to layout our hardware without a performance hit. Knowing how a design will translate into hardware can avoid many synthesis bugs.