Creating Generic Hardware

Introduction

This document introduces two important features of Verilog: parameterized modules, and `generate` blocks. These capabilities will allow you to create modules that are generic and re-usable. For example, you may already know how to build a 4-bit adder or an 8-bit adder, but now you will be able to create an N-bit adder. When instantiated, the value of N can be selected, and can differ between instantiations, all while only having had to write one version of the adder.

1 Parameterized Modules

The `parameter` keyword in Verilog allows a module to be configured during instantiation, with the configuration value visible as a named constant for use within the module. As an example, we will create an 8-bit counter that counts up from 0 to N-1, where N is a parameter:

```
// upcount.sv
module upcount #
(
  parameter N
)
(
  input clk,
  input sreset,
  output [7:0] o_val,
  input i_enable,
  output o_last
);
  logic [7:0] cnt;

  always_ff @ (posedge clk) begin
    if (sreset) cnt <= 8'd0;
    else if (i_enable) begin
      if (o_last) cnt <= 8'd0;
      else cnt <= cnt + 8'd1;
    end
  end

  assign o_val = cnt;
  assign o_last = cnt == N-1;
endmodule
```

```
// top.sv
wire [7:0] cnt17_val;  // goes to 16
wire [7:0] cnt112_val; // goes to 111

// explicit parameter and signal mappings
upcount #
(
  .N(17)
)
count17
(
  .clk(CLOCK_50),
  .o_val(cnt17_val),
  // ... rest of connections
);

// ordered parameters, explicit signals
upcount #(112) count112
(
  .o_val(cnt112_val),
  // ...
);
```

At the start of the counter module is a parameters section that begins with a #, coming before the usual input/output signals section. In here you can put one or more `parameter` declarations. The parameter N is used later within the module to set a maximum value to count to.

Inside the instantiating module (`top.v`), there are two instances of the counter: one that counts from 0 to 16 and one that counts to 111. When instantiating a parameterized module, there is a section that

begins again with a # **but comes before the instance name.** This section assigns concrete values to the parameters, and looks a lot like the usual signal-connection section (with parameter names being set instead of signals being connected). It also comes in the same two flavours: one which uses explicit by-name assignments to the parameters (using .), and a more compact one that does it by order. Both are shown.

There is an older style of syntax that you may still see. It's still valid in the latest versions of Verilog and SystemVerilog, so for completeness, here's the same example using it:

```
// upcount_old.v
module upcount
(
  input clk,
  input sreset,
  output [7:0] o_val,
  input i_enable,
  output o_last
);
  reg [7:0] cnt;
  parameter N;

  // rest of code is the same
endmodule
```

```
// top_old.v
upcount count64
(
  // just port connections
);

defparam count64.N = 64;
```

All of the above examples are still 8-bit counters, even if not all the bits are necessary. What if we wanted the `o_val` output of the counter to have a width that automatically adjusts based on N? The below example introduces a few new concepts: parameters can depend on other parameters, parameters can be used to change the width of signals, and the `$clog2` built-in system function, which is only available in SystemVerilog: given a constant value, it evaluates (at compile time!) the base 2 logarithm of the value and rounds it up to the next integer. In effect, it counts how many bits are required to represent a given value.

```
module upcount #
(
  parameter N,
  parameter Nbits = $clog2(N)
)
(
  ...
  output [Nbits-1:0] o_val,
  ...
);
  logic [Nbits-1:0] cnt;
  ...
```

不需要assign Nbits value

Note that parameters can also have default values assigned to them, so that the instantiating module doesn't have to explicitly set them. This was done above for `Nbits`.

2 `generate` blocks

In both Verilog and SystemVerilog, there exists syntax that allows you automate the instantiation of hardware: the `generate` block. These contain `if` statements and/or `for` loops that can conditionally instantiate hardware, or automatically create and name many instances of the same module.

As an example, we will create an 8-bit ripple-carry adder. It instantiates 8 full-adder modules and connects them together. However, instead of manually instantiating them, we'll use a `generate for` loop. The full adder module looks like this:

```
module fa
(
  input x, y, cin,
  output s, cout
);
  assign s = x ^ y ^ cin;
  assign cout = x&y | x&cin | y&cin;
endmodule
```

Here are two versions of the same 8-bit adder, one using `generate` statements and one without:

```
module add8
(
  input [7:0] x,
  input [7:0] y,
  output [8:0] sum
);
  logic [8:0] cin;

  assign cin[0] = 1'b0;
  assign sum[8] = cin[8];

  genvar i;
  generate
    for (i = 0; i < 8; i++) begin : adders
      fa fa_inst
      (
        .x(x[i]),
        .y(y[i]),
        .cin(cin[i]),
        .s(sum[i]),
        .cout(cin[i+1])
      );
    end
  endgenerate
endmodule
```

parameter (+1)

```
module add8
(
  input [7:0] x,
  input [7:0] y,
  output [8:0] sum
);
  logic [8:0] cin;

  fa fa0(x[0],y[0],1'b0,sum[0],cin[1]);
  fa fa1(x[1],y[1],cin[1],sum[1],cin[2]);
  fa fa2(x[2],y[2],cin[2],sum[2],cin[3]);
  fa fa3(x[3],y[3],cin[3],sum[3],cin[4]);
  fa fa4(x[4],y[4],cin[4],sum[4],cin[5]);
  fa fa5(x[5],y[5],cin[5],sum[5],cin[6]);
  fa fa6(x[6],y[6],cin[6],sum[6],cin[7]);
  fa fa7(x[7],y[7],cin[7],sum[7],sum[8]);
endmodule
```

There are a few things to note about the syntax. The `generate` block must have a matching `endgenerate`. Within it, any `if` or `for` blocks must have their own `begin` and `end` statements, and the blocks must be named (`'adders'` in this case). Variables like `i` that are used in `generate for` blocks must be declared as a special `genvar` type, outside the `generate` block. Although not shown, you are allowed to declare any other structural or behavioral code (like `assign` statements, creating `wire/logic` signals, entire `always` blocks) within `generate for` and `generate if` constructs. Finally, a powerful combination is possible: we could also have parameterized this module to be an N-bit ripple-carry adder. This is left as an exercise to the reader.