Verilog Lesson 2

Verilog Lesson 2 by T. Miller

Now that we know about module ports and wires, we can put together some simple combinatorial logic blocks. We'll stick to what we've covered so far and just add math and logic operators. C programmers will feel right at home.

```
module simple_math (
    input1, input2, sel_in,
    out_sum, out_dif,
    out_shl, out_shr,
    out_bit_and, out_bit_or, out_bit_xor, out_bit_not,
    out_bool_and, out_bool_or, out_bool_not,
    out_reduction_and, out_reduction_or,
    out_mux
    out_eq, out_ne, out_gt, out_lt, out_ge, out_le);

input [7:0] input1, input2;  // 8-bit inputs for simplicity
input sel_in;  // selector for ternary operator

// Outputs will be explained below
output [8:0] out_sum, out_dif;   // extra bit for carry
output [15:0] out_shl;
output [7:0] out_shr;
output [7:0] out_bit_and, out_bit_or, out_bit_xor, out_bit_not;
output out_bool_and, out_bool_or, out_bool_not;
output out_reduction_and, out_reduction_or;
output [7:0] out_mux;
output out_eq, out_ne, out_gt, out_lt, out_ge, out_le;

// Add and sub are simple enough.  In Verilog, if the sum were a
// sub-expression, its size (google for "natural size") would be 8 bits,
// but since it's being assigned to a 9-bit wire, it'll correctly produce
// the extra carry bit.
// The data types are unsigned from the point of view of Verilog, so
// if we care about signs, we have to add extra logic.

assign out_sum = input1 + input2;
assign out_dif = input1 - input2;

// We've given the shift-left some extra bits just for fun.  If you
// shift right or left too far, you'll just get zero as a result.  It's
// often useful to do a bit slice on the shift amount just to save logic,
// so if your input and output are 8-bit, you only need a 3 or 4 bit
// shift, unless you want to be able to shift more and get zero.

assign out_shl = input1 << input2;
assign out_shr = input1 >> input2[2:0];  // Demonstrating the bit-slice

// Bit-wise operators are just like in C.  Corresponding bits are
// operated on in parallel.

assign out_bit_and = input1 & input2;
assign out_bit_or = input1 | input2;
assign out_bit_xor = input1 ^ input2;
assign out_bit_not = ~input1;

// Boolean operators are a bit different from the bit-wise operators.
// Just like in C, the whole bus is considered to be zero (false) or any
// non-zero value (true).

assign out_bool_and = input1 && input2;
assign out_bool_or = input1 || input2;
assign out_bool_not = !input1;

// Verilog has reduction operators.  They take all bits in the bus and
// reduce them to a single bit that indicates if they're all 1's (and) or
// any of them are 1's (or).

assign out_reduction_or = |input1;
assign out_reduction_and = &input1;

// Note:  ~|input1 means no bits are 1, and ~&input1 means some bits are zero.
```

```
// C has this ?: operator that takes a boolean and chooses between two
// other expressions.  Verilog has the same thing.

assign out_mux = sel_in ? input1 : input2;

// How about some comparison operators

assign out_eq = input1 == input2;
assign out_ne = input1 != input2;
assign out_gt = input1 > input2;   // all of these are unsigned comparisons
assign out_lt = input1 < input2;
assign out_ge = input1 >= input2;
assign out_le = input1 <= input2;

endmodule
```

I'm sure I've missed a few, but I have the basics covered. Others can point out anything I didn't think of. There are also multiply (*), divide (/), and I think mod (%) operators, but you avoid them for synthesis. They are, however, quite useful for simulation. Some chips have built-in multipliers, and the synthesis tools will infer them from *, but I tend to manually instantiate them because then I have more control over additional features that the blocks tend to provide.

I don't know about Lattice, but Xilinx's synthesizer doesn't handle a carry-in correctly for an add, so if you do this:

```
wire [7:0] x, y;
wire c;
wire [8:0] z = x + y + c;
```

You'll get very poor results. The solution I use is to subtract the 1's compliment:

```
wire [8:0] z = c ? (x - (~y)) : (x + y);
```

The analogue for borrowing for subtract is to add the 1's compliment.

For synthesis, I would structure it a bit differently to get the synthesizer to infer a unified addsub block. (The extra parentheses generally are not necessary; I just like to make things clear to the reader.)

Created by: hamish last modification: Saturday 06 of October, 2007 [13:34:57 UTC] by asdf

source history similar **4 comments**

RSS Wiki RSS Blogs

Übersetzen Sie diese Seite ins Deutsche
Traduzca esta paginación a español
Traduisez cette page en français
Tradurre questa pagina in italiano
Traduza esta página em portuguêses
翻译这页成汉语 **(CN)**
日本語にこのページを翻訳しなさい **(Nihongo)**
한국인으로 이 페이지를 번역하십시요 **(Hangul)**

AltaVista Babel Fish
To **translate this page**, click a flag!