

Verilog by Example, errata

The verilog code on the opposite page, although also functionally equivalent to the previous code, now reflects a different and more compact way of representing the multiplexer function. We here introduce verilog's combinatorial conditional construct, eliminating the "in_3_bus" intermediate signal of the previous example in the process. The assign statement reads as such: "when ~~in_3_bus~~^{in_3} (the select control) is high, select in_2, else select in_1." This works very much like a limited version of the familiar IF/THEN statement of other languages.

Verilog by Example, errata

All of the combinatorial and bus reconstruction shown in the module above is implemented in one assignment in the code on the opposite page. Here we introduce bus concatenation, which is defined by a single set of braces. I have arranged the concatenation elements vertically on separate lines for clarity, but they could all be included on the same (albeit somewhat long) line, still separated by commas. Note that the MS element is always first (i.e., next to the left-most brace), while the LS element is always last (next to the right-most brace). Notice also that the first and last elements here comprise two bits, and that the two middle elements (each one bit) are the result of combinatorial operations.



“most significant”



“least significant”

Verilog by Example, errata

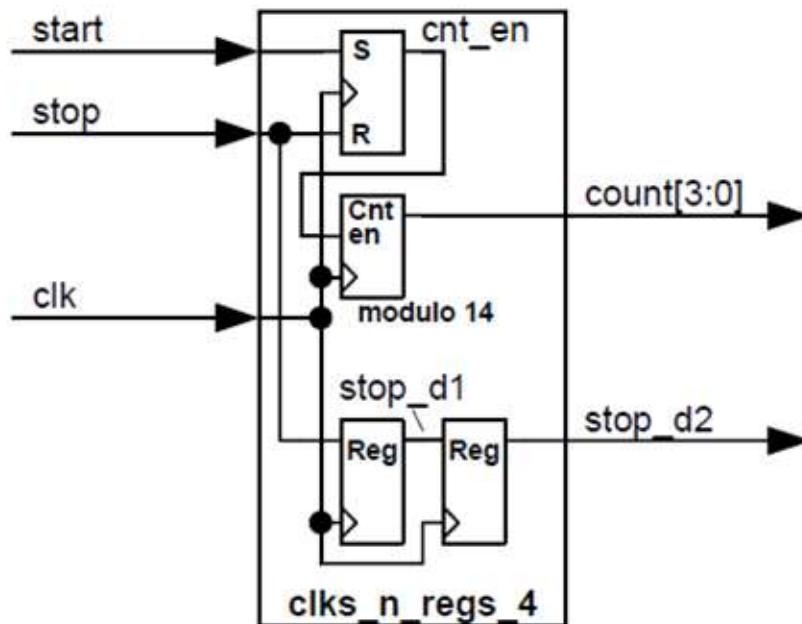
The body of the always-block has now become more complicated as we introduce if/else conditional statements to accommodate the reset. Any time “reset” ~~is~~^{is} high, “out_1” is forced to zero. Since this happens as soon as reset goes active (reset is part of the sensitivity list), and at every rising clock edge, you can see that this effects an asynchronous clear. When reset is

Verilog by Example, errata

Verilog by Example

We now introduce a few common state-type operations to show how increasingly sophisticated register-based functions are implemented in always-blocks. A four-bit counter is enabled by a “start” event, and stopped by a “stop” event. The SR flop allows the start and stop events to be short, e.g. one-clock pulses, rather than a continuously enabling flag. Additionally, for further illustration, we delay the ~~start~~ ^{stop} signal two clocks and send it out.

You’ll notice that we have not shown the asynchronous reset. This is done for clarity; from this point forward it is assumed. It is implemented in the code, and always will be (in this book).



SR flop and counter

Verilog by Example, errata

```
////////////////////////////////////  
// SR flop and counter  
////////////////////////////////////  
  
module srflop_n_cntr (  clk,  
                        reset,  
                        start,  
                        stop,  
                        count  
                        );  
  
    input                clk;  
    input                reset;  
    input                start;  
    input                stop;  
    output [3:0]         count;  
    output                stop_d2;  
    reg                 cnt_en;  
    reg [3:0]           count;  
    reg                 stop_d1;  
    reg                 stop_d2;  
  
    // ----- Design implementation -----  
  
    // SR flop  
    always @( posedge clk or posedge reset )  
        begin  
            if ( reset )  
                cnt_en <= 1'b0;  
            else if ( start )  
                cnt_en <= 1'b1;  
            else if ( stop )  
                cnt_en <= 1'b0;  
        end
```

Verilog by Example, errata

```
    end  
endmodule
```

SR flop and counter

The last always-block implements the two sequential delays. The points to note here are that multiple register signals can be grouped into the same always-block (when it makes sense), and that additional begin/end block boundaries are needed around each pair of signal assignments. Without these, the synthesis software might interpret, for example, that “stop_d2 <= stop_1” is not associated with the “else,” but stands alone.

Finally, we should note that the three always-blocks could be collected together into one. This is shown on the next page.

Verilog by Example, errata

```
////////////////////////////////////  
// Single-port Memory  
  
module single_port_mem  
    (clk,  
     reset,  
     data_io,  
     address,  
     wr_en,  
     rd  
    );  
  
    input          clk;  
    input          reset;  
    inout [15:0]   data_io; //new I/O type  
    input [9:0]    address;  
    input          wr_en;  
    input          rd;  
  
    reg [15:0]     memory[0:1023];  
    reg [15:0]     dat_out; data_out  
    reg            rd_d1;  
  
    // ----- Design implementation -----  
  
    // Memory  
    //  
    always @(posedge clk )  
    begin  
        if (wr_en)  
            memory[address] <= data_io;  
        dat_out := memory[address];  
        data_out rd_d1 <= rd;  
    end  
  
    assign data_io = rd_d1 ? data_out : 16'bz;  
  
endmodule
```

Single-port Memory

Verilog by Example, errata

```
////////////////////////////////////  
// Clock Buffer  
  
module clock_buffer  
    ( reset,  
      clk_in,  
      dat_in,  
      dat_out  
    );  
  
    input      reset;  
    input      clk_in;  
    input      dat_in;  
    output     dat_out;  
  
    wire      clk;  
    reg       dat_out;
```

60

Verilog by Example, errata

signed values: Verilog-2001 added “signed” types to regs and constants. This defines the value as signed, two’s-complement. A signed reg declaration might look like:

```
reg signed [31:0] data val;
```

and a signed constant might be:

```
parameter signed [7:0] WIDTH = 8'h56;
```

Inputs and outputs associated with signed regs would be:

```
input signed [31:0] data val;
```

```
output signed [31:0] data val;
```