Discussion #6
Friday, February 21, 2014
Milestone 1 due this Monday, 2/24
- 1 major component (e.g. RS, ROB) plus stellar, fully automated testbench
- Should synthesize (and work!)
- Will be graded on functionality, style, and coverage of testbench

Milestone 2 due 3/26
- Recommended to have mult_no_lsq.s working
Administrative

- Last lab
- But we will be having another discussion on Friday, 3/14, about the memory system
Administrative

- Midterm
  - Median: 80
  - Standard Deviation: 13.61
Agenda

- Miscellaneous SystemVerilog
  - Multidimensional arrays
  - “unique” / “priority” case
  - Assertions
- Generic designs
  - “generate” blocks
  - “for” loops
  - Synthesis quirks
Motivation

- We think some of this stuff will be useful in your projects
- Some constructs are easy to use, some have a steep learning curve
- Spend some time thinking about what would be worth investing your time in to incorporate
What is SystemVerilog?

- 1995 – Verilog HDL
- 2001 – Verilog 2001
- 2005 – SystemVerilog

- Emphasis on creating a combined Hardware Description Language and Hardware Verification Language
- Ability to debug at the “system” level
### Multidimensional Arrays

- **[127:0] and [63:0]** are referred to as “packed dimensions”
- **[3:0]** is referred to as an “unpacked” dimension, and is usually used to represent memory
- Read from right to left, outside in

```verilog
logic [127:0] [63:0] multi_d_array [3:0];
assign multi_d_array [3][101] = 64'hFFFF_FFFF;
```
Multidimensional Arrays

- Old Verilog only allows one packed dimension
- SystemVerilog allows as many as you need
- You probably only need packed arrays

```verilog
logic [127:0] [63:0] multi_d_array [3:0];
assign multi_d_array [3][101] = 64'hFFFF_FFFF;
```
Multidimensional Arrays

- Packed arrays are represented internally as a contiguous set of bits, which means you can easily move data between differently configured arrays.

```vhdl
logic [31:0] one_d_array;
logic [15:0] [1:0] two_d_array;
assign two_d_array = one_d_array;
```
Unique/priority if/case

How will the synthesis tools convert this design to hardware?

```vhdl
always_comb
  case(sel)
  2'b00: z = a;
  2'b01: z = b;
  2'b10: z = c;
  endcase
```

A latch will be generated, since a value for $z$ was not specified for $\text{sel} == 2\text{'}b11$.

```verilog
always_comb
  case(sel)
    2'b00: z = a;
    2'b01: z = b;
    2'b10: z = c;
  endcase
```
Unique/priority if/case

- What if we know sel will never equal 2’b11?
- We could add a dummy default state, but that would add unnecessary logic and potentially mask errors
- SystemVerilog has a “priority” construct for if/case statements
  - Tells synthesis to not generate a latch
  - Checks at run-time that at least one of the branches is true
Unique/priority if/case

```verilog
always_comb
priority case(sel)
  2'b00: z = a;
  2'b01: z = b;
  2'b10: z = c;
endcase
```

If, during behavioral simulation, `sel` is ever 2'b11, a warning will be generated:

```
RT Warning: No condition matches in 'priority case' statement.
"case.v", line 7, for top.DUT, at time 3.
```
Unique/priority if/case

- What hardware will be generated by this code?

```vhdl
input [1:0] sel;
output logic [1:0] z;
if (sel[1])
    z = 2'b00;
else if (sel[0])
    z = 2'b01;
else
    z = 2'b11;
```
Unique/priority if/case

- Hardware will give priority to higher bits, since it assumes multiple could be high.

```plaintext
input [1:0] sel;
output logic [1:0] z;
if (sel[1])
    z = a;
else if (sel[0])
    z = b;
else
    z = c;
```
Unique/priority if/case

- What if we’re using one-hot encoding?
SystemVerilog has “unique” if/case statement

- Tells synthesis to ignore priority logic and to not generate any latches
- Generates run time warning if zero or multiple branches are taken

```verilog
data
code
case
   
in [1:0] sel;
   output logic [1:0] z;
   unique if (sel[1])
      z = a;
   else if (sel[0])
      z = b;
   else
      z = c;
endcase
```
Unique/priority if/case

- Unique/priority can be used for both “if” and “case” statements
- Replace “full_case” and “parallel_case” pragmas from Verilog
  - These pragmas are undesirable, as they can cause mismatch behavior between behavioral and synthesized simulations
- Useful for simplifying logic and (more importantly) making designs more explicit
Assertions

- Statements declaring some invariant
- Can be inserted in testbenches or RTL (however, it will be ignored by synthesis)
- Great for automated testing, periodically check that certain conditions are being met
- Two types:
  - Immediate: directly called in code
  - Concurrent: running in parallel in the background
Immediate Assertions

- Nothing too new
- Saves some coding space

```verilog
adder a1(a, b, c);
initial begin
  if((a+b) != c)
    $display("Error with summation at time %h", $time);
end
```

Assertion failure: “lab6.v”, 4: top.assertion_1: started at 0s failed at 0s
Immediate Assertions

- Immediate assertions can be placed anywhere in procedural code
Concurrent Assertions

- Much more interesting (and challenging)
- Describe high level properties of your design and have the simulator periodically check them
- SystemVerilog supports an entire assertion language for this
Concurrent Assertions

● Implication

\[ s1 \implies s2; \]
  - If \( s1 \) is true, than \( s2 \) must also be true

● Timing windows

\[ @(\text{posedge} \ clock) \ (a \ \&\& \ b) \implies \#\#[1:3] \ c; \]
  - On the posedge of the clock, if both \( a \) and \( b \) are true, then between 1 and 3 clocking cycles later, \( c \) will be true
Concurrent Assertions

queue DUT (.data_in, .data_out, .push, .pop, .size, .clock, .reset, .empty, .full);

property p1;
    @(posedge clock)
        reset |=> (size == 0) && !full && empty;
endproperty
assert property(p1) else $finish;
 Concurrent Assertions

queue DUT (.data_in, .data_out, .push, .pop, .size, .clock, .reset, .empty, .full);

property p2(logic[$clog2(QUEUE_SIZE):0] cur_size);
   logic [31:0] data;
   disable iff (reset || cur_size != size)
   @(posedge clock)
   (push && !full, data = data_in) ->
      pop[->cur_size] ##1 (data_out == data);
endproperty
Concurrent Assertions
Concurrent Assertions

For more info on how to use SystemVerilog’s assertion language, check out A Practical Guide for SystemVerilog Assertions by Srikanth Vijayaraghavan, or Google it.
Generic Designs

- We want to make designs where we can easily and succinctly make changes to certain features (like # ROB entries).
- For example, the multiplier in P2 could be modified to use a different number of stages in just a few lines.
- We can do better! Ideally we’d like to change just one parameter.
Recall that many complex designs can be built from several simpler designs.

```plaintext
one_bit_adder addr_8 [7:0] (
    .a(a), .b(b), .cin({carries, cin}),
    .sum(sum), .cout({cout, carries}));
```
Generate blocks

- An adder is simple, just a 1-d array of smaller adders
- What about more complicated structures, like a tree (the priority selectors from P1 were trees of smaller selectors)?
Generate blocks

- Generate blocks allow us to use procedural statements ("if" or "for") outside of procedural blocks.
- Can be used to generate generically sized, complex structures.
- Use "generate/endgenerate" keywords.
Generate blocks

generate
  genvar i;
  for (i=0; i<N; i++) begin
    one_bit_addr ( .a(a[i]), .b(b[i]),
                  .cin(carries[i]),
                  .sum(sum[i]),
                  .cout(carries[i+1]));
  end
endgenerate
generate
  genvar i;
  for (i=1; i<N; i++) begin
    localparam left_right = i[0];
    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
Generate blocks

- Before compilation, the tools will “elaborate” the design, evaluating “if” statements and unroll “for” statements.
- All conditions must be deterministic at compile time.
Generate blocks

- Check the webpage for more details: http://www.eecs.umich.edu/eecs/courses/eecs470/labs/lab6_ex/AMI.pdf
Procedural “for” loops

- Generate blocks allow us to generate arbitrarily large logic structures near the structural level.
- What about at the procedural level? We know about “if” statements, but what about “for” loops? Can we use those?
- Yep!
Procedural “for” loops

- Didn’t we say earlier “for” loops were bad outside of testbenches?
- Not necessarily, but easy to mess up
Procedural “for” loops

- When we use “for” loops in software (or in testbenches), we think about sequential steps
  - Iteration 1 happens, THEN iteration 2, THEN iteration 3… etc
- But this is hardware, things are usually happening in parallel, i.e. at the same time
Procedural “for” loops

- Does this Verilog make sense for actual hardware?

```verilog
logic parity = 0;
for(int i=0; i<32; i++) begin
    if(in[i])
        parity = ~parity;
end
```
Procedural “for” loops

- “for” loops can be very valuable, just not how you would necessarily think from a software perspective
- Great for condensing repetitive code
Procedural “for” loops

if(reset) begin
    entry[0] = 0;
    entry[1] = 0;
    ...
    entry[31] = 0;
end

if(reset) begin
    for(int i=0; i<32; i++)
        entry[i] = 0;
end
Procedural “for” loops

- Keys for synthesis
  - All iterations will be done in parallel
  - Watch out for circular logic!
  - You should be able to mentally unroll a loop and think about how it would be implemented in hardware at compile time, as this is essentially what the compiler will do.
Procedural “for” loops

- Keys for synthesis
  - Remember that only the “last” assignment to a particular variable in a procedural block takes effect
Procedural “for” loops

```verilog
always_comb begin
  a=0;
  if(en)
    a=1;
end
```

- If “en” is true, a=1 takes precedence since it’s “lower” (priority logic generated)
Procedural “for” loops

```plaintext
for (int i=0; i<32; i++)
    a=i;
```

- In this case “a” will be assigned 31, since if we expanded the loop out, 31 would be the “lowest”
Procedural “for” loops

for (int i=0; i<32; i++) begin
    a=i;
    if(cond[i]) break;
end

- Break statements are synthesizable, but be cautious! Lot’s of hidden priority logic could hurt clock period
Procedural “for” loops

- Golden rule: Don’t use Verilog as a way of not thinking about actual hardware
  - Will result in designs that don’t synthesize or become overly complex
- First think about how you would build this in hardware at a high-level, then think about how Verilog constructs can make that easy to describe
Procedural “for” loops

- Design compiler sets a maximum number of loop iterations to prevent infinite loops
- 1024 by default
- If you need more, add this line to your .tcl file (replacing [max_iterations] with appropriate integer)
  ```tcl
  set hdlin_while_loop_iterations [max_iterations]
  ```
Synthesis quirks

- Structs and parameterized modules are very useful for generic designs (lab 3)

```verilog
testbench #(parameter NUM_ENTRIES=32) DUT {
  input IN_STRUCT_t in,
  output OUT_STRUCT_T out
};
```

- Design compiler has some quirks when interfacing with modules using SV constructs
Synthesis quirks
Synthesis Quirks

Post-synthesis simulation... uhh...
SystemVerilog Wrapper Files

- We need wrapper files so that testbenches using SystemVerilog constructs can connect to structural .vg files
SystemVerilog Wrapper Files
SystemVerilog Wrapper Files

- In .tcl file,
  
  ```
  write -format svsim -output $svsim_file
  $design_name
  ```

- In testbench,
SystemVerilog Wrapper Files

`ifdef SYNTH_TEST
`define DUT(mod) mod``_svsim
`else
`define DUT(mod) mod
`endif

...

`DUT(DUT) #(NUM_ENTIRES(`SIZE)) (.in, .out)