EECS 470 Lab 6

SystemVerilog

Department of Electrical Engineering and Computer Science
College of Engineering
University of Michigan

Thursday, October. 10th, 2019

Overview

Crazy times are here

Project

- Milestone 1 is due Wednesday, October 23rd!
  - You should have at least one module fully completed and debugged
  - Turn in: progress report, and one module and testbench for us to grade
  - Some resources (like fast priority selector) are available on the course site!

Motivation

Why SystemVerilog? Why now?

- Extra features that will be useful in your projects
- Not all features are easy to use: many have a steep learning curve
- The goal isn't to go wild and try to use everything...
- Instead, think about which are worthwhile to incorporate

What is SystemVerilog?

1. 1995 – Verilog HDL
2. 2001 – Verilog 2001
3. 2005 – SystemVerilog

- Emphasis on creating a combined Hardware Description Language and Hardware Verification Language
- Ability to debug at the "system" level
- Provides the basis for very powerful, object-oriented testbenches
- The framework for industry verification tools, e.g. UVM

Multidimensional Arrays

Example

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

Explanation

- "[127:0]" and "[63:0]" are called "packed" dimensions
- "[3:0]" is an "unpacked" dimension
- When referencing for read/write, unpacked dimensions come first, then packed dimensions

Multidimensional Arrays

Example

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

Explanation

- Old Verilog only allows one packed dimension
- SystemVerilog allows as many as you need
- We recommend packed arrays for most designs

Multidimensional Arrays

Example

- `logic [31:0] one_d_array;`
- `logic [15:0] [1:0] two_d_array;`
- `assign two_d_array = one_d_array;`

Explanation

- Packed arrays are laid out as a contiguous set of bits
- Allows easy copying from one array to another

Unique/priority if/case

```
input a, b, c;
input [1:0] sel;
output z;
case (sel)
        2'b00: z = a;
        2'b01: z = b;
        2'b10: z = c;
endcase
```

How will the synthesis tool convert this design to hardware?

Unique/priority if/case

```
input a, b, c;
input [1:0] sel;
output z;
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 when sel == 2'b11

latch.png

Unique/priority if/case

What if you know sel will never equal 2'b11?

- You could add a dummy state, but that adds unnecessary logic and potentially hides errors
- SystemVerilog has a "priority" construct for exactly this problem
  - Tells synthesis tool not to generate a latch
  - Checks at run-time that each state is reachable

Unique/priority if/case

```
input a, b, c;
input [1:0] sel;
output z;
priority case (sel)
        2'b00: z = a;
        2'b01: z = b;
        2'b10: z = c;
endcase
```

During behavioral simulation, if sel is 2'b11, a warning will be generated:

RT Warning: No condition matches in priority case statement.

Unique/priority if/case

Another code example:

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

What hardware will be generated by this code?

Unique/priority if/case

Another code example:

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

Tool will give priority to higher bits, since it assumes multiple bits could be high?

- But what if we're using one-hot encoding?

multiple_bits.png

Unique/priority if/case

SystemVerilog has "unique" if/case statement

```
input [1:0] sel;
output logic [1:0] z;
unique if (sel[1])
        z = a;
else if (sel[0])
        z = b;
else
        z = c
```

- Tells synthesis tool to assume one-hot encoding
- Ignores priority logic and doesn't generate any latches
- Generates simulation warning if multiple bits are high

Unique/priority if/case

Unique & Priority used for both if and case statements

- Replaces "full_case" and "parallel_case" pragmas from old Verilog
- Useful for simplifying logic and clarifying design choices

Assertions

Assertions

- Strategy for automated testing: check that certain conditions are true
- Statements declaring some kind of invariant
- Can be inserted in testbenches or RTL (ignored by synthesis)
- Two types:
  - Immediate: directly called in code
  - Concurrent: running in background

Immediate Assertions

Need to check that some expression is true...

```
adder a1(a, b, c);
initial begin
        if ((a+b) != c)
        $display("Error!");
end
```

Better done by immediate assertion...

```
adder a1(a, b, c);
initial begin
        assert ((a+b) == c);
end
```

Concurrent Assertions

- Much more interesting (and challenging)
  - Describe high-level functional correctness of your design...
  - ...and have simulator check these invariants in the background
- SystemVerilog supports an entire assertion language (!)
  - (beyond the scope of what we will do in class)
- Implication
  - `s1 |-> s2`
    - If s1 is true, then s2 must also be true
- Timing windows
  - `(a && b) |-> ##[1:3] c;`
    - On the posedge of the clock, if a and b are true, then 1-3 cycles later, c will be true

Assertions

Assertions

- For more information on assertions, check out "A Practical Guide to SystemVerilog Assertions"

"For" loops

"You want 'for' loops? You can't handle 'for' loops!"

- We told you earlier in the semester that "for" loops are not a thing
- We lied, sort of... but they don't work the way they do in software
- In software we think about iterations of loops
  - Iteration 1, then Iteration 2, then Iteration 3... etc...
- In hardware, loops need to unroll completely at design time
  - Self-modifying hardware is still not a thing...
  - So either everything runs in parallel (good)
  - Or loop can "break" when a certain condition is true (can get ugly)

For loops

Does this make sense for actual hardware?

```
parity = 0;
for (int i=0; i<32; i++) begin
        if (in[i])
                parity = ~parity;
end
```

For loops

Designing synthesizable "for" loops

- "For" loops can be valuable, just different than software
  - Just another way of doing combinational logic, not a replacement for sequential logic
  - Very limited ability to change signals referenced in the loop
- Great for condensing repetitive code, because everything will be done in parallel
- Visualize how a loop can be built into hardware at synthesis time

For loops

Blocking assignment in loops

```
always_comb begin
        for (int i=0; i<32; i++)
                a = i;
end
```

- What will a equal?
- 31, because if we unrolled the loop, the assignment to 31 would be last

For loops

Break Statements

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

- Effect: break out of loop once condition is true

For loops

Max loop iterations

- Design Compiler sets a maximum number of loop iterations to prevent infinite loops
  - This is configured to be 1024 by default
  - If you need more, add this line to your .tcl file:
    `set hdlin_while_loop_iterations (iterations)`

Final advice

- Remember: don't use Verilog as a way to avoid thinking about actual hardware
  - This results in synthesis problems or overly complex designs
- First think about how to build the hardware, then think about the Verilog constructs that can allow you to describe your design easily

Generic Designs

Goal: complex designs with a single parameter

- Want to make designs where we can easily change certain features
  - For example, the number of ROB entries
- The multiplier in P2 could be modified using parameters
- We can build complex designs... remember module arrays?

```
one_bit_adder add_8 [7:0] (
        .a(a), .b(b), .cin({carries, cin}),
        .sum(sum), .cout({cout, carries}));
```

- What if we couldn't condense everything to a single parameter?
  - An adder is simple, just an array of smaller adders
  - What about more complex structures like the priority selectors from P1 that are trees of smaller selectors?

Generate Blocks

Generate blocks give control

- Using a generate block to build hardware:

```
generate
    genvar i;
    for (i=0; i<N; i++) begin
        one_bit_adder (
            .a(a[i]), .b(b[i]),
            .cin(carries[i]),
            .sum(sum[i]),
            .cout(carries[i+1]));
    end
endgenerate
```

Generic Designs

Goal: complex designs with a single parameter

- How does this work?
  - The tool will "elaborate" the design
  - Evaluate "if" statements and unroll "for" loops
- Important: all conditions must be deterministic at compile time

Generate Blocks

Another example: the Priority selectors from P1:

```
generate
    genvar i;
    for (i=0; 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
```

Lab Assignment

- Assignment is posted to the course website as Lab 6 Assignment.
- If you get stuck...
  - Ask a neighbor, quietly
  - Put yourself in the help queue
- When you finish the assignment, sign up in the help queue and mark that you would like to be checked off.
- If you are unable to finish today, the assignment needs to be checked off by a GSI in office hours *before* the end of next week.