EECS 470 Lab 5

SystemVerilog

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

Thursday, February 8th, 2024

Overview

Crazy times are here

Homework

- HW 3 is due is next Wednesday, February 14th

Exam

- Exam is next Thursday, February 15th

Project

- Project proposals due Tuesday, February 13th
- Group meetings Thursday and Friday, February 15th -16th
- Then 3 weeks (one of which is Spring break) until milestone 1 meetings

Project 2 Feedback

Implementations + Style

- Give every signal a default value in `always_comb` blocks
  - Do this before you write anything else
  - This makes avoiding latches easy
- Be very aware of how your Verilog translates to hardware
  - Ex: If you use "`*`", that makes a (very expensive) multiplier

Free Response

- Each pipeline stage requires overhead (setup + hold time, wiring)
- You should be able to analyze the latency of your module
  - Understand the latency associated with your design choices
  - Don't rely on Verdi to tell you how long something takes to compute

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
- Related to how the wires get laid out in hardware

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

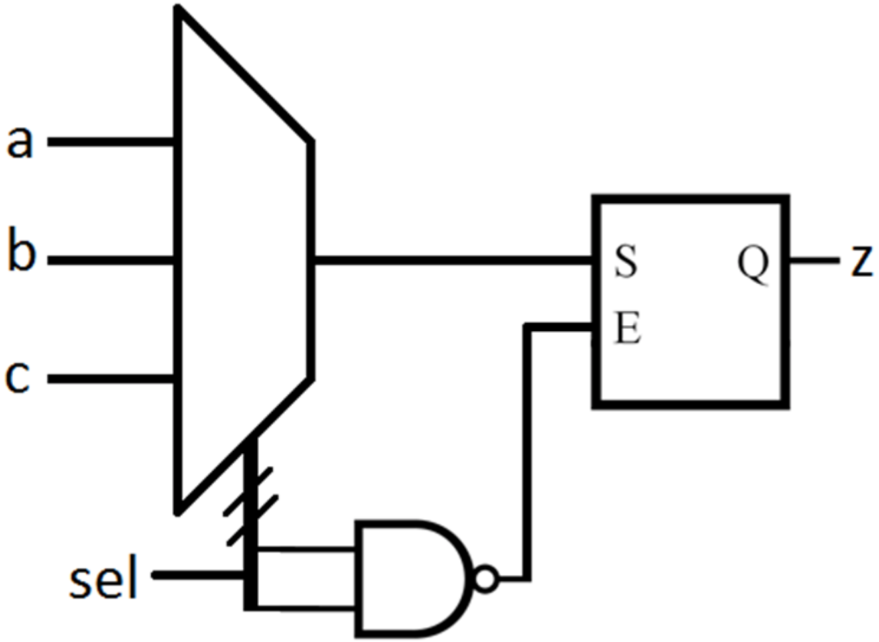
```
input a, b, c;
input [1:0] sel;
output z;
case (sel)
    2'b00: z = a;
    2'b01: z = b;
    2'b10: z = c;
endcase
```

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


a
b
c
sel
S
Q
E
z

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?

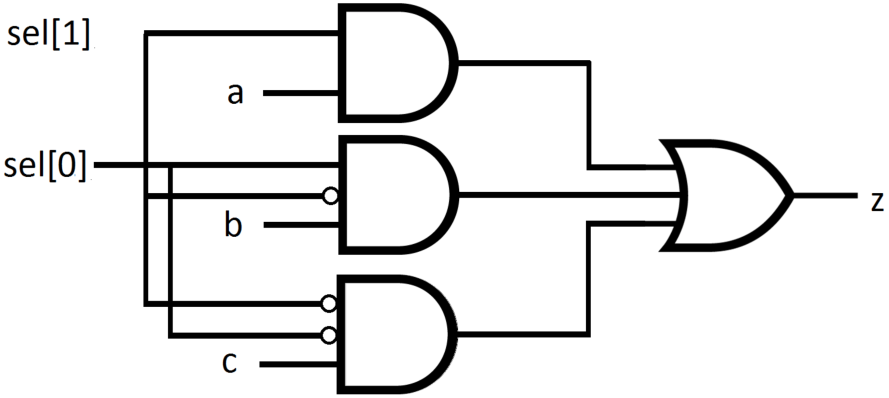
Unique/priority if/case

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?


sel[1]
a
sel[0]
b
c
z

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

- Idea: Embed invariants about your design in the module itself
- Verify your design automatically every clock cycle!
    - Describe high-level functional correctness of your design...
    - ...and have simulator check these invariants in the background
- Very helpful for documenting your design
    - Describe what should be true at all times
    - Lay out sequences of events that should occur
- Makes writing a testbench much simpler
- SystemVerilog supports an entire assertion language (!)
    - Very powerful, but can be confusing

Assertion Syntax Basics

- Clocked behavior
  - `@(posedge clock) disable iff(reset)` : examine at every positive clock edge, except if reset is high
  - `clocking cb @(posedge clock) ... endclocking` : Everying in this block should be evaluated at the positive clock edge
- Implication
  - `s1 |-> s2`: If s1 is true, then s2 must also be true *at the same time*
  - `s1 |=> s2`: If s1 is true, then s2 must also be true the *next cycle*
- Timing windows
  - `(a && b) |-> ##[1:3] c;`
    - If a and b are true, then 1-3 cycles later, c must be true
  - `int x; (a, x=data) ##[1:$] c`
    - when a is true, save data to x and then wait $[1, \infty)$ cycles until c

Assertions

Assertions

- For more information on assertions, check out "A Practical Guide to SystemVerilog Assertions"
- An example set of assertions is used to test lab 5. See the source code for the example

"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

wor and wand

- For loops lend themselves to another type of wire: `wor` and `wand`
- Those wire types allow you to do multiple assignments to one signal
- The final value will be the logical OR (for wor) or AND (for wand) of all assignments.

Index Part-Select

- "-:" and "+:" are shorthand for selecting based by an index and width
- Syntax: `signal[<start>+:<width>]` or `signal[<end>-:<width>]`
- Example: `array[WIDTH*i+:WIDTH]` takes the bits from WIDTH*i to WIDTH*(i+1)-1

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 add_8 (
            .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:

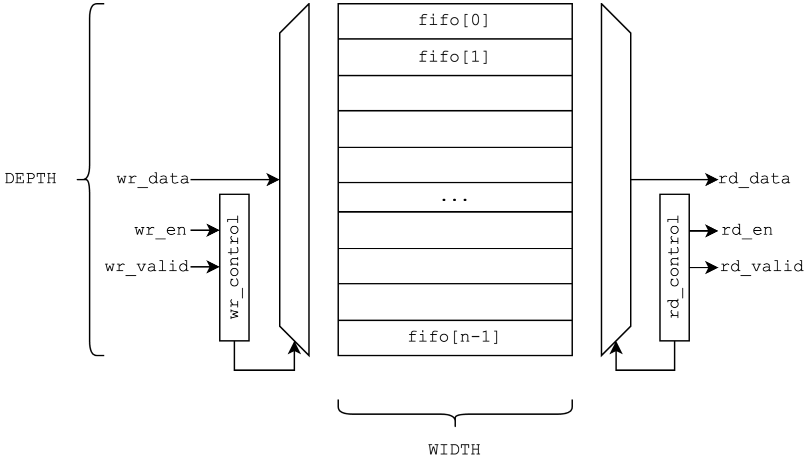
```
generate
    genvar i;
    for (i=0; i<N; i++) begin
        localparam left_right = i[0];

        ps2 ps_i (
            .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 Tips

- Circular Buffers aka FIFOs are used all over the place in HW design
  - Any time you need to buffer data from source to destination
  - Storing stalled requests until hazards are resolved
  - Also a very common interview question
- Goal: design a generic FIFO that you can reuse in your final project
- Key parameters:
  - Depth: how many entries are in the FIFO
  - Width: # bits in each entry
  - Alert depth: when to send an alert that the buffer is almost full
- Nuances:
  - Head and tail need to wrap around ("circular" buffer)
  - How do you track empty, full, and almost full?

Lab Diagram


DEPTH
wr_data
wr_en
wr_valid
wr_control
fifo[0]
fifo[1]
...
fifo[n-1]
WIDTH
rd_data
rd_en
rd_valid
rd_control

Design Decisions

- One-hot vs encoded read/write pointers:
  - Encoded is more space efficient and easier to read/use
  - One-hot is faster (addition is just a shift) but requires more area
- Options for differentiating empty and full states:
  - Keep a separate count for number of entries
  - Track which entries are valid or not
  - Have one extra bit for full vs empty
  - Make it so you always have at least one empty slot
- How do you drive the read data?
  - Does this need to be combinational or sequential?
  - Does it need to be 0 when data is invalid?
- What if you need to do multiple reads and writes per cycle?
  - This isn't part of the lab, but may be needed for superscalar processors

Lab Assignment

- Assignment is posted to the course website as Lab 5 Assignment.
- **NOTE:** we are giving you two weeks for this one because it is more challenging
- 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/IA in office hours *before* the end of next week.