EECS 470 Lab 5

SystemVerilog

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

Thursday, October 2nd, 2025

Overview

Administrivia

SystemVerilog Constructs
- Multidimensional Arrays
- For Loops
- Generate Blocks

Verification
- Assertions, Assumptions, Covers
- Formal Verification

Lab Assignment

Appendix
- Unique and Priority
- For Loops

Schedule Updates

Homework

- HW 3 is due Friday, October 3rd

Exam

- Wednesday, October 8th (next week)

Project

- Project proposals due before your meeting
- Group meetings this Thursday and Friday
- Then 2 weeks until milestone 1

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...
  - It's good to have an general overview of what tools are available
  - Think about which are worthwhile to incorporate

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 from left to right
- Related to how the wires get laid out in hardware
- **We recommend only using packed dimensions**

Multidimensional Array Equivalence

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
- Also allows for easier resetting of signals

For Loop Tricks

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

Fancy For Loop Example

```
module RdyVldMerge
  #(parameter SOURCES = 2, parameter WIDTH   = 16
  )(input          [SOURCES-1:0] srcVld,
    input   [WIDTH*SOURCES-1:0] srcData,
    input                       dstRdy,
    output logic [SOURCES-1:0] srcRdy,
    output wor                  dstVld,
    output wor       [WIDTH-1:0] dstData);
wor [SOURCES-1:0] srcGnt;
psel_gen #(.WIDTH(SOURCES),.REQS(1)) psel_gen(.req(srcVld),.gnt(srcGnt));
generate
    for (genvar i = 0; i < SOURCES; i++) begin
        assign srcRdy[i] = dstRdy;
        assign dstVld    = srcGnt; // dstVld and dstData use wor
        // Need to alternatively assign to '0 to not impact wor
        assign dstData   = srcGnt[i] ? srcData[i*WIDTH +: WIDTH] : '0;
    end
endgenerate
endmodule
```

Generic Designs

Goal: Adjust complex designs with a single parameter

- Want to make designs where we can easily change certain features
  - Ex: number of ROB entries
- The multiplier in P2 could be modified using parameters (stages)
- What if parametrization doesn't just mean repetition?
  - 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

What Do They Do?

- Let you conditionally generate blocks of hardware
  - Modules, assign statements, procedural blocks
- Without `generate`, loops in `always_comb` blocks can only parameterize assignment operations (create muxes)

How do I do it?

- All conditions must be deterministic at compile time
- Loop variables must be “genvar” type

Generate Block Example

Simple Adder Example

```
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
```

Generate Block Example 2

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
```

Generate Block Example 3

Choose which design to use

```
generate
    if (USE_PSEL_GEN) begin
        psel_gen #(.WIDTH(8), .REQS(1))
        psel_gen  (
            .req   (reqs),
            .gnt   (gnts));
    end else begin
        ps8 my_ps (
            .req   (reqs),
            .en    (1'b1),
            .gnt   (gnts),
            .req_up());
    end
endgenerate
```

Packages

What if you don't want to everything globally defined?

- Packages provide a way to define macros, parameters, and structs for specific modules/uses
- Define the package in one place...
- And include it in all relevant modules

Example: wonka_p.sv

```
package wonka_p;
    parameter NUM_OOMPA_LOOMPAS = 42
    typedef enum {CHOCOLATE, GUM, ...} PRODUCT_TYPE;
    function binary2onehot();
        // ...
    endfunction
endpackage
```

Including Packages

```
module factory
  import wonka_p::*;      // import entire package
  #(parameter CLP = 3)    // candy level parallelism
   (input              clock,
    input PRODUCT_TYPE product_type,
    // ...
  );
    // implement module
endmodule
```

Why Verification?

Need to make sure it works

- Have you ever had something work the first time?
- . . . probably not

Mistakes are expensive

- An error in silicon can cost millions of dollars in losses
- Companies tend to have 2-3 verification engineers for every designer

Verification can be fun!

- Verification engineers have the best understanding of how things work
- Can build really elegant models of your design

Assertions

Assertions

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

Assumptions

- Similar to assertions, but describe what is assumed to be true
- Applies constraints on what the verification tool can do

Covers

- Embeds edge cases you want to ensure are verified

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

Shortcomings of Immediate Assertions

- Only used in testbench
- Often requires lots of manual reasoning / coding

Benefits of Concurrent Assertions

- Verify your design automatically every clock cycle!
- Better reflect nature of hardware we're simulating
- Makes writing a testbench much simpler
  - Or maybe you don't have to write a module-level testbench at all
- Very helpful for documenting your design

Concurrent Assertion Syntax Basics

- Clocked behavior
  - `@(posedge clk) disable iff(rst)` : examine at every positive clock edge, except if rst is high
  - clocking cb @(posedge clk) ... 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
  - `a |-> s_eventually c`
    - After a is true, c must eventually also be true (this is also called liveness)

Assertions

Best Practices

- At the very least, code extremely simple assumptions into assertions
  - Ex: signal a is one-hot
- Make error messages informative so they are easy to trace
  - Print module, line number, value of failing signal
- Encode assumptions about inputs of module
  - Helps with integration through verifying assumptions about interfaces
  - Ex: Two signals that can't be asserted together
  - Ex: CDB index must map to an in-use ROB entry

More Info

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

Formal Verification Motivation

How can you be sure your design *truly* works?

- It's very hard to manually test *every* edge case
- As designs become more complex, the number of states explodes exponentially

Why Formally Verify?

- Can use FV for quick check of functionality
- ***The tool gives a debug waveform describing the failure***
- Can also add cover properties to ensure sufficient state exploration

FV Details

FV vs DV

- Ideally, you do both formal verification and design verification
- Formal verification (FV): use math/state analysis to verify assertions
- Design verification (DV): write targeted test (module and system-level)
- FV is (exponentially) less feasible as modules become larger

FV Steps

1. Encode assertions (and covers) in design
2. Write FV wrapper with any extra assertions, assumptions, and covers
3. Run the verification tool (JasperGold)
4. Debug any failing asssertions using the generated waveform

FV Process

Details on how to run formal verification are in the lab 5 spec.

verify.tcl (session_0) - JasperGold Apps (.../source/jgproject) - Main

File Edit View Design Reports Application Window Help

Formal Property V...

Design Hierarchy

- FIFO_fv (FIFO_fv:(WIDTH=12,SIZE=31))
  - dut (FIFO:(SIZE=31,WIDTH=12))

Property Table

Type	Name	Engine	Bound	Time	Task
Cover	FIFO_fv.cov_full	B	24 - 72	0.8	<embedded>
Cover	FIFO_fv.cov_rd_vld	Hp	2	0.2	<embedded>
Cover	FIFO_fv.cov_wr_vld	Hp	1	0.2	<embedded>
Cover	FIFO_fv.cov_afull	B	25 - 54	0.2	<embedded>
Cover	FIFO_fv.cov_wr_wrap	B	32	0.0	<embedded>
Cover	FIFO_fv.cov_rd_wrap	B	32	0.1	<embedded>
Assert	FIFO_fv.dut.DUT_sva.ValidRd	Ht	22 - 5368...	12.9	<embedded>
Assert	FIFO_fv.dut.DUT_sva.ValidWr	N	26 - 5368...	0.9	<embedded>
Assert	FIFO_fv.dut.DUT_sva.ValidFull	N	30 - 5368...	0.9	<embedded>
Assert	FIFO_fv.dut.DUT_sva.ValidAFull	N	27 - 5368...	0.9	<embedded>
Assume	FIFO_fv.dut.DUT_sva.CountIn	?		0.0	<embedded>

Total: 25 Filtered: 25 Selected: 0 Validity: 15:0:9:1 Run: 1:0:9:15

session_0

```
INFO (IPF047): 0.0.Ht: The cover property "FIFO_fv.dut.DUT_sva.genblk1[1].cov_bit_i" was covered in 3 cycles in 0.00 s.
INFO (IPF047): 0.0.Ht: The cover property "FIFO_fv.dut.DUT_sva.genblk1[2].cov_bit_i" was covered in 5 cycles in 0.01 s.
INFO (IPF047): 0.0.Ht: The cover property "FIFO_fv.dut.DUT_sva.genblk1[3].cov_bit_i" was covered in 9 cycles in 0.23 s.
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.cov_wr_wrap" was covered in 32 cycles in 0.03 s.
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.dut.DUT_sva.genblk1[4].cov_bit_i" was covered in 17 cycles in 0.03 s by the incidental trace "FIFO_fv.cov_wr_wrap".
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.cov_rd_wrap" was covered in 32 cycles in 0.10 s.
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.dut.DUT_sva.genblk1[5].cov_bit_i" was covered in 33 cycles in 0.08 s.
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.dut.DUT_sva.genblk1[6].cov_bit_i" was covered in 65 cycles in 0.20 s.
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.cov_afull" was covered in 54 cycles in 0.20 s by the incidental trace "FIFO_fv.dut.DUT_sva.genblk1[6].cov_bit_i".
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.dut.DUT_sva.genblk1[7].cov_bit_i" was covered in 129 cycles in 0.79 s.
INFO (IPF047): 0.0.B: The cover property "FIFO_fv.cov_full" was covered in 72 cycles in 0.79 s by the incidental trace "FIFO_fv.dut.DUT_sva.genblk1[7].cov_bit_i".

[<embedded>] %
```

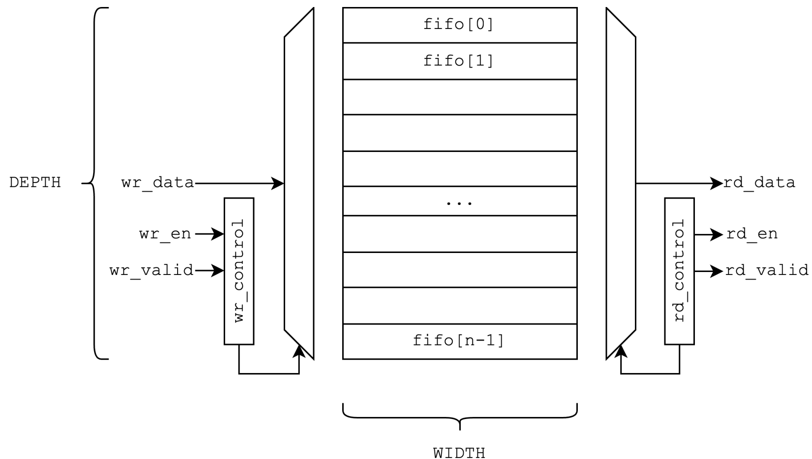
Console | Lint Messages | Warnings / Errors | Proof Messages

Console input ready

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
  - Max Count: max value of the "spots" output
- Nuances:
  - Head and tail need to wrap around ("circular" buffer)
  - How do you track empty, full, and almost full?

Lab Diagram



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

Debugging Your FIFO

Formal verification is new this semester, so you have 2 options to debug:

Option 1: Provided Testbench

- Was used last semester, should thoroughly test design

Option 2: Formal Verification

- Good practice use of JasperGold
- Everything should be set up, all you have to do is run
- Provides waveform on a failure
- Please give feedback on how you like it

Lab Assignment

- Assignment is posted to the course website as Lab 5 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/IA in office hours *before* the end of next week.

Appendix

Note

Everything from here to the end of the presentation is from an older version of this presentation, but kept here for reference.

Unique/priority if/case

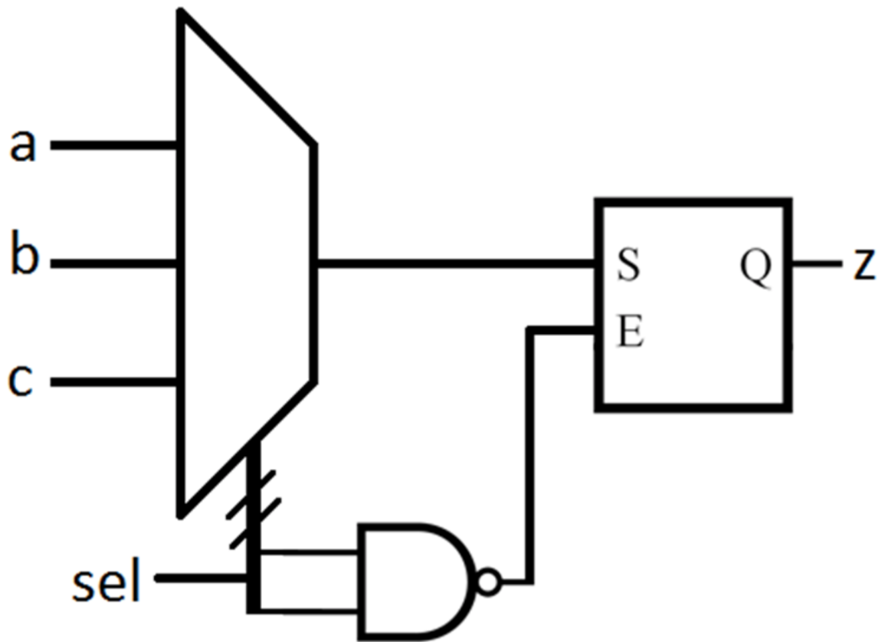
```
input a, b, c;
input [1:0] sel;
output z;
case (sel)
    2'b00: z = a;
    2'b01: z = b;
    2'b10: z = c;
endcase
```

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
E
Q
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.

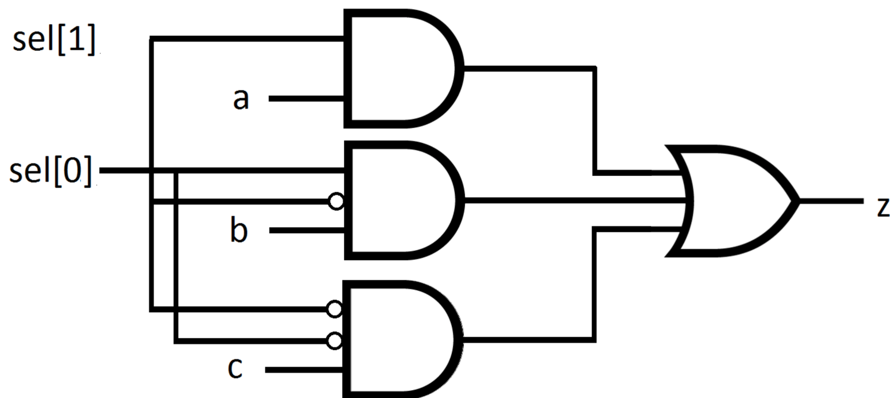
Unique/priority if/case

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



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

"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