EECS 470 Lab 3

SystemVerilog Style Guide

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

Thursday, 11th September 2025

Overview

Administrivia

Custom Types

Finite State Machines

Verilog Style

Testing
- Coverage
- VCD Files

Appendix

Administrivia

Homework (due on Gradescope)

- Homework 2 is due Wednesday, 17th September at 11:59PM

Projects (due via submission script)

- Project 2 is due Monday, 15th September at 11:59PM
- Project 3 will be released Monday
- Project 3 Milestone is due Monday, 22nd September at 11:59PM
- Project 3 is due Monday, 29th September at 11:59PM

Help

We are available to answer questions on anything here. Office hours can be found on the course web site.

SystemVerilog Types

User-defined Types

- Useful for simplifying repeated declarations, bundling connections
- Types should be named informatively, e.g. `arch_reg_t`

About `struct`

- A group of signals (`wire` or `logic`)
- Basically follow C conventions
- Use `packed` structs so wires are contiguous

About `enum`

- List of possible values, but named instead of numbered
- Good for state machine states
- Can be shown in Verdi with the name instead of the associated value

Typedef

About `typedef`

- Necessary for reuse of a `struct` or `enum`
  - Without a `typedef`, a `struct`/`enum` must be redefined at each instance declaration
- Also useful in clearly naming commonly sized buses

Syntax

1. `typedef`
2. then any signal declaration or `struct` or `enum` declaration
3. Lastly, name for the type followed by a semicolon (`;`)

Typedef by Example

Example: Typedef'd Enum

```
//typedef, then definition, then name;
typedef enum logic [3:0] {
    IDLE,
    GNT[0:7],
    RESET
} arb_state;
```

Example: Type Synonym

```
//typedef, then definition, then name;
typedef logic [63:0] addr;
```

Procedural FSM Design

FSM Process

1. Define states in a `typedef enum`
2. Describe next state logic in a combinational block, following all combinational logic rules
3. All resets should be synchronous (to the clock)
4. Make output assignments in their own combinational block
5. The only logic in the sequential block should be the state assignment (to the computed next state)

Finite State Machine Skeleton

```
typedef enum logic [(NUM_STATES-1):0] { STATES } fsm_state;

module fsm(
    input wire inputs,
    output logic outputs
    );

    fsm_state state, next_state;

    always_comb begin
    /* Transitions from a diagram go here */
    /* next_state = f(inputs, state) */
    end

    always_ff @(posedge clock) begin
        if(reset) begin
            state <= DEFAULT;
        end else begin
            state <= next_state;
        end
    end
endmodule
```

Finite State Machine Example

```
typedef enum logic { LOCKED, UNLOCKED } ts_state;

module turnstile(
    input wire coin, push,
    input wire clock, reset,
    output ts_state state
    );

    ts_state next_state;

    always_comb begin
        next_state = state;
        if (state==LOCKED   && coin) next_state = UNLOCKED;
        if (state==UNLOCKED && push) next_state = LOCKED;
    end
    always_ff @(posedge clock) begin
        if (reset) state <= LOCKED;
        else       state <= next_state;
    end
endmodule
```

Common Project 2 issue

Where is the latch? - Discuss with a neighbor

```
// Cycle through an 8 bit history, return if the state changed from 8 cycles ago
module history (input clock, reset, update, value,
                output state_changed);
    logic [7:0] state, next_state;
    logic [2:0] idx, next_idx;
    always_comb begin
        next_idx        = update ? idx + 1 : idx;
        next_state[idx] = update ? value   : state[idx];
        state_changed   = next_state[idx] != state[idx];
    end
    always_ff @(posedge clock) begin
        if (reset) begin
            idx   <= '0;
            state <= '0;
        end else begin
            idx   <= next_idx;
            state <= next_state;
        end
    end
endmodule
```

Verilog Style

What is good style?

- Easy to read, understand, and maintain
- High reusability

Why should I use good style?

- Easier to debug: save your sanity
- Important for group work: give your teammate(s) an easier life :)
- Mandatory in projects 1-3

3 Main Guidelines

1. Clarity: make your point clear
2. Brevity: less is more
3. Alignment: simplify parsing

Clarity

Clarity Rules

- Use meaningful names for signals
  - Defining structs and enums often makes signals and states clearer
  - Ex: `assign flag = state == FLAG_STATE;`
- Comment your designs
  - Ex: `(a ^ b ~^ c) | (&d)` is unintelligible without an explanation
  - Writing comments before implementations can help keep yourself organized
- Something more verbose may be much easier to read
  - Making the text shorter without changing the number of operations does not decrease hardware size

Brevity Rule

Rule

- Brevity is (often) strongly correlated with the optimal solution
- Be brief, where you can
- Nobody wants to read a whole essay

Exception

Is brevity always better? Not necessarily... You also need to be clear

Example

```
assign shift[15:1] = shift[14:0];
assign shift[0]    = new_data;
```

Vs

```
assign shift = {shift[14:0], new_data};
```

Alignment Rule

Rule

- Assignments should be aligned by column.
- Ternary statements should have the conditionals aligned, and each "if" should be on a new line.
- Try to align bit width declaration and indices when possible
- Spaces are preferable over tabs for portability across IDEs

Example

```
case (size)
    2'd0: x_n[ 1:0] = in[ 9:8];
    2'd1: x_n[ 3:0] = in[10:7];
    2'd2: x_n[ 7:0] = in[12:5];
    2'd3: x_n[15:0] = in[15:0];
endcase
```

Coverage

What is Coverage?

- Quantitatively measure quality of a testbench
- Check if you test all desired states
- Gives a percentage of how many states you tested

Why is Coverage?

- Want to make sure you test all edge cases
  - Just like your targeted tests in lab 2!
- A testbench does nothing if it doesn't test anything useful
- Formal measure to check your own competence
- We will ask you to report your coverage as part of milestone 1 in the final project

VCD Files

What Are VCD Files?

- **V**alue **C**hange **D**ump file
- Saves the values of internal signals to a file
- Allows you to view and analyze them after simulation

How do I use them?

- Add the following lines in the `initial` block of your testbench

```
$dumpfile("<filename>.vcd");
$dumpvars(0);
```

- Install a VCD viewing vscode extension and open the produced vcd file
- . . . Or write your own GUI debugger to parse it and visualize your processor

Lab Assignment

- Assignment is posted to the course website as Lab 3 Assignment.
- If you get stuck...
  - Ask a neighbor, quietly
  - Put yourself in the help queue
- When you are finished, put yourself on help queue to get checked off.
- If you are unable to finish today, the assignment needs to be checked off by next Friday.

Appendix

Below are further examples and details about the topics covered in this presentation.

They are excluded from the presentation to maintain brevity but included here for clarity.

Clarity by Example

Example

```
logic enable;

always_comb
begin
    enable =
    (op[0] ^ op[1] |
     op[2] & op[3]) ?
            1'b1 : 0'b0;
end
```

vs.

Example Reformatted

```
logic enable;

always_comb
begin
    if(op[3])
    begin
        if(op[2])
        begin
            enable = 1'b1;
        end else if(op[0] ^ op[1])
        begin
            enable = 1'b1;
        end else begin
            enable = 1'b0;
        end
    end else begin
        enable = 1'b0;
    end
end
```

Brevity by Example

Example

```
always_comb
begin
    if (foo[3] == 1'b1)
    begin
        bar[3] = 1'b1;
        bar[2] = 1'b0;
        bar[1] = 1'b1;
        bar[0] = 1'b1;
    end else if (foo[2] == 1'b1)
    begin
        bar[3] = 1'b0;
        bar[2] = 1'b1;
        bar[1] = 1'b0;
        bar[0] = 1'b0;
    end
end
```

vs.

Example Reformatted

```
always_comb
begin
    if      (foo[3]) bar = 4'b1011;
    else if (foo[2]) bar = 4'b0100;
end
```

Brevity by Example

Example

```
logic [5:0] shift;

always_ff @(posedge clock)
begin
    if (reset)
    begin
        shif_reg <= 6'b0;
    end else begin
        shift[0] <= foo;
        shift[1] <= shift[0];
        shift[2] <= shift[1];
        shift[3] <= shift[2];
        shift[4] <= shift[3];
        shift[5] <= shift[4];
    end
end
```

vs.

Example Reformatted

```
logic [5:0] shift;

always_ff @(posedge clock)
begin
    if (reset)
    begin
        shift <= 6'b0;
    end else begin
        shift <= {shift[4:0], foo};
    end
end
```

Brevity & Clarity Rules

- Use common sense and intuition.
- Ask your teammate(s) to peer-review.
- Use the names to describe what is happening
- For example:
  - "_d" suffix for combinational wires
  - "_q" suffix for sequential registers
  - "_n" for next state logic or active low signals
  - Macros or parameters for describing names of states.

Indentation

Why are we interested in indentation?

- Readability – easier to trace down
- Clarity – easier to check what is in a given scope

Why are we interested in alignment?

- Readability – easier to trace down
- Clarity – easier to check that everything is assigned

Indentation and Alignment Example

Example

```
always_comb
begin
    if (cond)
    begin
        n_state = `IDLE;
        n_gnt = `NONE;
    end else begin
        n_state = `TO_A;
        n_gnt = `GNT_A;
    end
end
```

vs.

Example Reformatted

```
always_comb
begin
    if (cond)
    begin
        n_state  = `IDLE;
        n_gnt    = `NONE;
    end else begin
        n_state  = `TO_A;
        n_gnt    = `GNT_A;
    end
end
```

Alignment by Example

Example

```
assign mux_out = (cond1) ? (foo1&bar): (cond2) ? (foo2+cnt3) :
(cond3) ? (foo3&~bar2) : 0;
```

Example Reformatted

```
assign mux_out = (cond1) ? (foo1 &  bar)  :
                 (cond2) ? (foo2 +  cnt3) :
                 (cond3) ? (foo3 & ~bar2) : 0;
```

Structs

Example

```
typedef struct {
    logic [7:0] a; //Structs can contain
    logic       b; //other structs, like
    arch_reg_t  c; //<-- this line
} example_t; //named with _t
```

Example

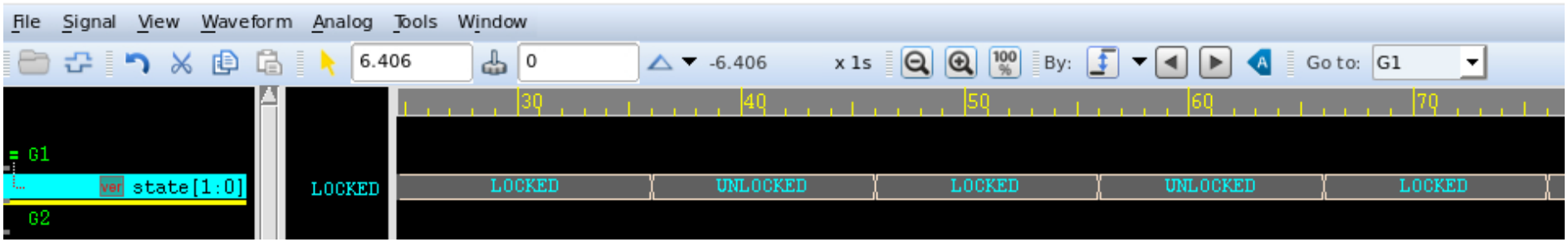
```
typedef struct packed {
    addr_t  pc;
    logic   valid;
} prf_entry_t;
```

Usage Example

```
prf_entry_t [31:0] prf;
assign prf[1].valid = 1'b0;
```

Enums

Verdi Example



Syntax

- `enum`
- List of values between braces ({})
- Name after braces, followed by a semicolon (;)

Enums

Example

```
typedef enum logic [3:0] {
    IDLE,      //=0, by default
    GNT[0:7], //Expands to GNT0=1,...GNT7=8
    RESET      //=9
} arb_state;
```

Example

```
typedef enum logic [1:0] {
    ADD   = 2'b00,  //The value associated with
    MULT  = 2'b10,  //a particular name can be
    NOT   = 2'b11,  //assigned explicitly.
    AND   = 2'b01
} opcode;
```

Usage Example

```
arb_state state, n_state;
assign n_state = IDLE;
```

Coverage Cont.

Code Coverage Types

- **Line Coverage** — measures statements in your HDL code that have been executed in the simulation
- **Toggle Coverage** — measures the bits of logic that have toggled during simulation. A toggle simply means that a bit changes from 0 to 1 or from 1 to 0.
  - One of the oldest metrics of coverage in hardware designs and can be used at both the register transfer level (RTL) and gate level.
  - May give low numbers if you have wide signals (ex: adding 64 bit numbers)
- **Condition Coverage** — measures how the variables or sub-expressions in the conditional statements are evaluated during simulation. It can find errors in the conditional statements that cannot be found by other coverage analysis.

Coverage Cont.

Code Coverage Types Cont.

- **Branch Coverage** — measures the coverage of expressions and case statements that affect the control flow (such as if-statement and while-statement) of the HDL. It focuses on the decision points that affect the control flow of the HDL execution.
- **FSM Coverage** — verifies that every legal state of the state machine has been visited and that every transition between states has been covered.
- For more information about coverage, feel free to ask us or search online!

Coverage Cont.

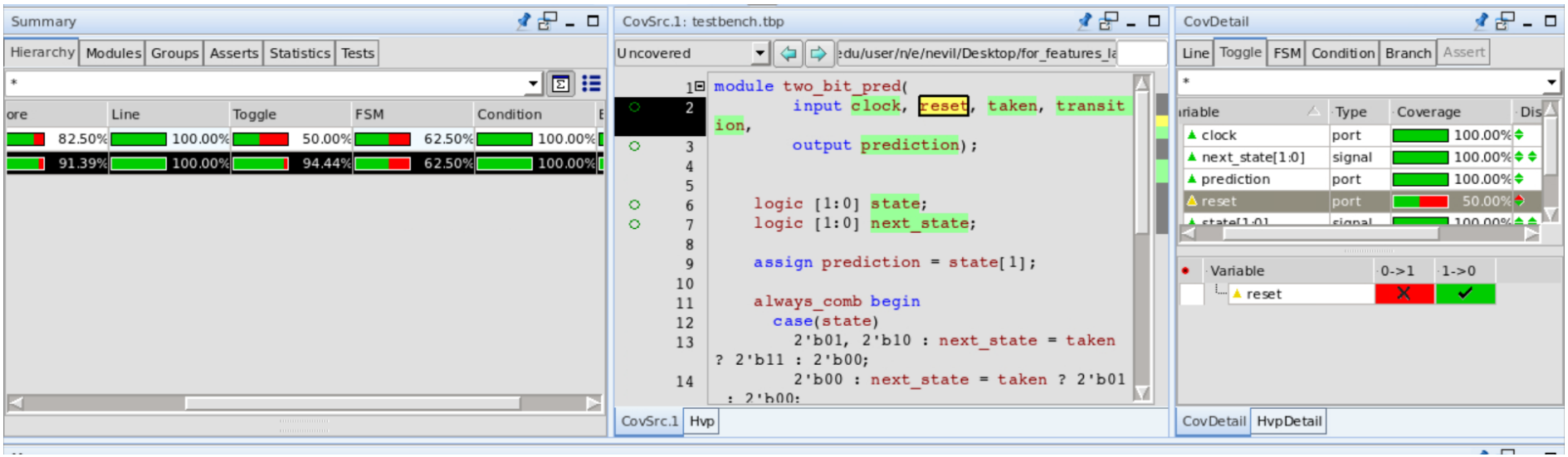
Coverage compile time options

- `-cm line|cond|fsm|tgl|branch|assert`
- The arguments specifies the types of coverage:
  - `line` - Compile for line or statement coverage.
  - `cond` - Compile for condition coverage.
  - `fsm` - Compile for FSM coverage.
  - `tgl` - Compile for toggle coverage.
  - `branch` - Compile for branch coverage
  - `assert` - Compile for SystemVerilog assertion coverage.
- If you want VCS to compile for more than one type of coverage, use the plus (+) character as a delimiter between arguments, for example:
  - `vcs -sverilog -xprop=tmerge +vc -Mupdate -line -full64 -kdb -lca -nc -debug_access -cm fsm+tgl mydesign.sv mytest.sv -o simv`
  - `./simv -cm fsm+tgl`

Coverage example

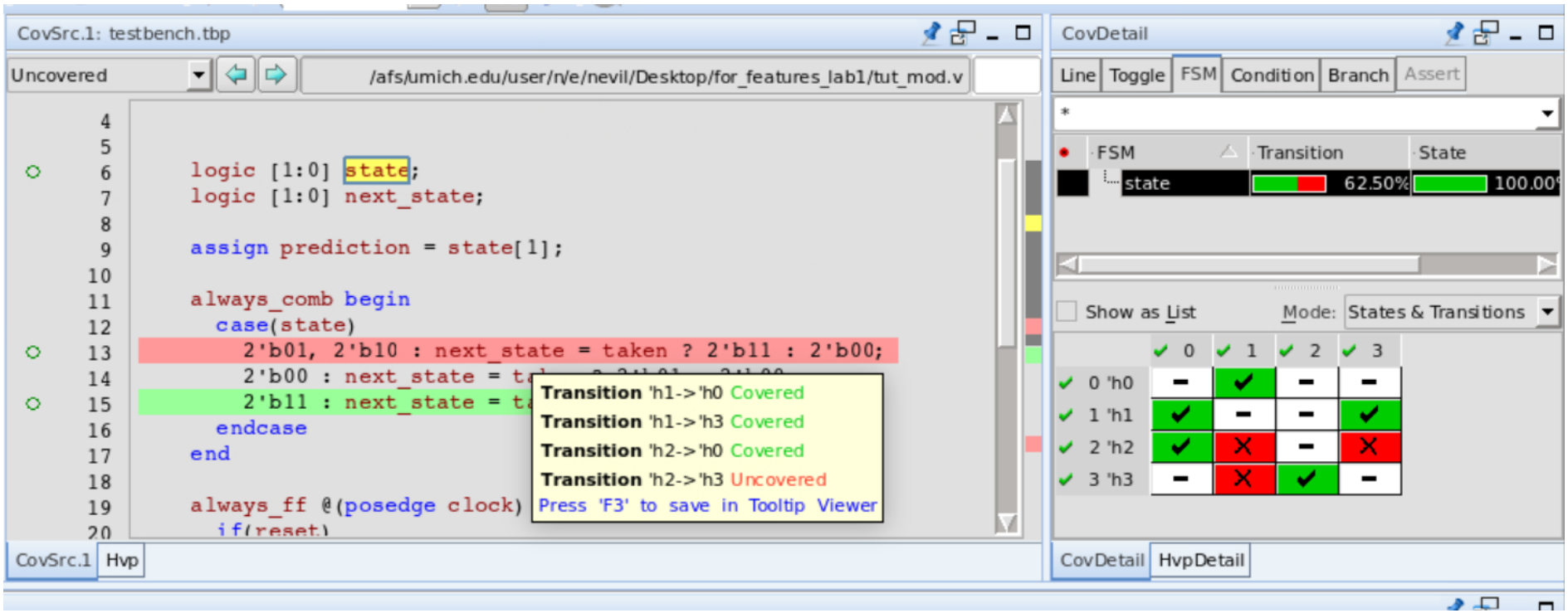
Run `make cov.verdi` in the lab 3 scripts to open Verdi in coverage mode.

Verdi Coverage



Coverage example Cont.

Verdi Coverage



Coverage Cont.

We will come back to this later in the final project.

- As part of milestone 1, you will need to submit a module along with a testbench and we will grade you based on your coverage percentage
- In the meanwhile... Give it a try on your project 2 testbench!