EECS 470 Lab 3
SystemVerilog Style Guide

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

Friday, 24th January, 2014
Overview

Administrivia

Verilog Style Guide
  Brevity
  Indentation and Alignment
  SystemVerilog Features

Finite State Machines

Project 2

Assignment
Homework

- Homework 2 is due Monday, 3rd February at the beginning of lecture.

Projects

- Project 2 is due Monday, 27th January at 9:00PM.

Help

We are available to answer questions on anything here. Office hours can be found in the course google calendar.

Due to room scheduling conflicts, office hours might be in 1620 or 1695. Please check both before posting to Piazza asking where we are.
Verilog Style

What is good style?
- Easy to read
- Easy to understand
- Mostly a matter of personal preference

Why should I use good style?
- Easier to debug
- Important for group work
- Mandatory in projects 1-3
Verilog Style Rules

Goal: Clarity

- For any problem, one solution is the clearest
- We want to approximate this solution
- By creating a set of rules to follow

Format

- We’ll look at a bad example
- Then how to fix it
- Derive a rule
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
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

Example Reformatted

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

```verilog
logic [5:0] shift;

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

**Example Reformatted**

```verilog
logic [5:0] shift;

always_ff @(posedge clock)
begin
    if (reset)
    begin
        shift <= #1 6'b0;
    end else begin
        shift <= #1 {shift[4:0], foo};
    end
end
```
Example

```
logic [5:0] shift;

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

Example Reformatted

```
logic [5:0] shift;

always_ff @(posedge clock)
begin
    if (reset)
        begin
            shift <= #1 6'b0;
        end
    else begin
        shift <= #1 {shift[4:0], foo};
    end
end
```
Brevity Rule

Rule
Brevity is strongly correlated with the optimal solution. Be brief, where you can.
General Requirements

Clarity Rules

- Use meaningful names for signal; wire wire; is confusing
- Comment your designs; (a ^ b ~^ c) | (&d) is unintelligible without an explanation
- Conceptualize what you need to build before you start writing Verilog. A state machine diagram will be make the Verilog much easier...
Why are we interested in indentation?

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

```verilog
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
```

Example Reformatted

```verilog
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
```
Indentation by Example

Example

```verilog
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
```

Example Reformatted

```verilog
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.
Indentation Rule

Rule
Items within the same scope should have the same indentation.
Alignment

Why are we interested in alignment?

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

Example

```verilog
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
```

Example Reformatted

```verilog
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.
Alignment by Example

Example

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

Example

\[
\text{assign } \text{mux\_out} = (\text{cond1}) \? (\text{foo1} \& \text{bar}) : (\text{cond2}) \? (\text{foo2} + \text{cnt3}) : \\
(\text{cond3}) \? (\text{foo3} \& \neg \text{bar2}) : 0;
\]

Example Reformatted

\[
\text{assign } \text{mux\_out} = (\text{cond1}) \? (\text{foo1} \& \text{bar}) : \\
(\text{cond2}) \? (\text{foo2} + \text{cnt3}) : \\
(\text{cond3}) \? (\text{foo3} \& \neg \text{bar2}) : 0;
\]
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.
User-defined Types

- Useful for cleaning repeated declarations, specifically bundling connections
- Types can be named informatively, e.g. arch_reg_t
Structs

About struct

- A package of signals (wire or logic)
- Basically follow C conventions
  - List of signal declarations
  - Named with _t ending
- Only struct packed is synthesizable, more in lab 6

Syntax

- struct
- List of signals between braces ({})
- Name after braces, followed by a semicolon (;)
Structs

Example

typedef struct packed {
    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

About enum

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

Syntax

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

### Example

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

### Example

```verilog
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

```verilog
arb_state state, n_state;
assign n_arb_state = IDLE;
```
Enums

DVE Example
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

- typedef
- by any signal declaration or struct or enum declaration
- Name for the type followed by a semicolon (;)
Typedef by Example

Example: Typedef’d Enum

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

Example: Type Synonym

```verilog
//typedef, then definition, then name;
typedef logic [63:0] addr;
```
Procedural FSM Design

FSM Process

- All states should be `typedef enum`
- All next state logic should go into a combinational block, following all combinational rules
- All resets should be synchronous (to the clock)
- All output assignments should go in their own combinational block
- The only logic in the sequential block should be the state assignment (to the computed next state)
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 <= #1 DEFAULT;
    end else begin
        state <= #1 next_state;
    end
endmodule
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 <= #1 LOCKED;
    else state <= #1 next_state;
end
endmodule
Project 2 Overview

Part 1: Pipelined Multiplier
- Change the pipeline depth
- Synthesize at each size

Part 2: Integer Square Root
- Finite state machine implementation
- Synthesis
Partial Products

- Multiply the first $n$ bits of the two components
- Multiply the next $n$ bits, etc.
- Sum the partial products to get the answer
Pipelined Multiplication by Example

**Binary Multiplication**

\[
\begin{array}{c}
0 & 0 & 0 & 1 & 1 & 1 \\
\times & 0 & 1 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

**Decimal Multiplication**

\[
\begin{array}{c}
7 \\
\times & 5 \\
\hline
3 & 5 \\
\end{array}
\]
Example: 4-stage Pipelined Multiplication

multiplicand: 00001011
multiplier: 00000111
partial product: 00000000

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\times & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Example: 4-stage Pipelined Multiplication

- **Multiplicand:** 00001011 $\ll 2$
- **Multiplier:** 00000111 $\gg 2$
- **Partial Product:** 00100001

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\times & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}
\]
Example: 4-stage Pipelined Multiplication

multiplicand: 00101100
multiplier: 00000001
partial product: 00100001

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
\times & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
Example: 4-stage Pipelined Multiplication

| multiplicand: | 00101100  << 2 |
| multiplier:   | 00000001  >> 2 |
| partial product: | 01001101 |

```
 0 0 1 0 1 1 0 0
× 0 0 0 0 0 0 0 1
 0 0 1 0 1 1 0 0
```
Part 1 Hints

Pipeline Depth

- Figure out the sizes of all the structures as a function of pipeline depth
- This would be an optimal place to use `define

Synthesis

- Clock Periods ($T_{clk}$)
  - Optimize inner module (mult_stage.v) first, then outer (mult.v)
  - Understand set_dont_touch
- Synthesis is time consuming; $\approx 10$ minutes for both mult_stage and mult
Part 2 Hints

ISR Algorithm

- Guess-and-check
- Loop from the top bit of the guess to the bottom
- Basically binary search for a solution

Hardware Implementation

- How do we implement this in hardware?
Part 2 Hints

ISR State Machine
Computing: $\sqrt{\text{value}}$

- On a reset
  - guess initialized to 32’h8000_0000
  - value is clocked into a register
- guess gets the next bit set each time we cycle through the FSM again
- Square guess (multiply it with itself)
  - Wait until the multiplier raises its done
- if guess $\leq$ value
  - Keep the current bit
- else
  - Clear the current bit
- Move to the next bit
- After the last bit, raise done
Part 2 Hints

Reminders

▷ Remember to declare bitwidths for signals, e.g. 64’hFFFF_FFFF_FFFF_FFFF
▷ It must take less than 600 cycles to compute a square root
▷ Remember to use the 8-stage multiplier for this
▷ Remember to check for proper reset behavior
▷ Remember the reset pragma (/\synopsys\ sync_set_reset “reset”)
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 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 next lab session.