ECE4700J Lab 2
Verilog: Design Tutorial

Haoyang Zhang

UM-SJTU Joint Institute

zhy-sjtu-jc@sjtu.edu.cn

May 26, 2022

Overview

Administrivia

- Lab 1 Assignment's live demo will be on 8:00pm - 10:30pm, Friday May 27th (Beijing Time). Lab 1 Assignment code submission will be due on 23:59pm, May 27th (Beijing Time).
- Homework 1 will come soon.
- Lab 2 Assignment will be released soon today. It will be due on Friday next week.

For Online Live Demo

- Please put yourself in the demo queue: `https://sjtu.feishu.cn/sheets/shtcn5u6ycQejXPQ2dQpLq24nhb`.
  - We will invite students into a discussion room to do the demo.
  - Others please wait in the main room.

Vivado Post-synthesis Simulation

Your buggy results of post-synthesis simulation can be caused by any one of these:

- **Badly described hardware**. (according to the synthesis guide), for example, your design generates some bad latches.
- **Timing violations**. Usually, timing violations will be printed to the log when you are simulating. (show as "WARNING")
- **FPGA can be in the configuration state for "some time"**. In post-synthesis simulation this time is modeled. Take a look at the waveforms later in time - they will probably behave as you expect. One good way is to insert a long latency (for example, 10x clock_period) doing nothing before your tests and after the reset, or simply reset for a long time.
- Other bugs - Try to look at your synthesis log file.

Tasks and Functions

`task`

- Reuse commonly repeated code
- Can have delays (e.g. #5)
- Can have timing information (e.g. `@(negedge clock)`)
- Might be synthesizable (difficult, not recommended)

`function`

- Reuse commonly repeated code
- No delays, no timing
- Can return values, unlike a `task`
- Basically combinational logic
- Might be synthesizable (difficult, not recommended)

task Example

task Example

```
task exit_on_error;
    input [63:0] A, B, SUM;
    input C_IN, C_OUT;
    begin
$display("@@@ Incorrect at time %4.0f", $time);
$display("@@@ Time:%4.0f clock:%b A:%h B:%h CIN:%b SUM:%h"
    "COUT:%b", $time, clock, A, B, C_IN, SUM, C_OUT);
$display("@@@ expected sum=%b", (A+B+C_IN) );
$finish;
    end
endtask
```

function Example

```
function check_addition;
    input wire [31:0] a, b;
    begin
        check_addition = a + b;
    end
endfunction

assign c = check_addition(a,b);
```

Verilog Macros

Definitions

- Syntax: `` `define <NAME> <value> ``
  - Note: That is a **tick**, not an apostrophe before define
  - The tick character is found with the ~, usually above tab (on US standard layout keyboards)
- Usage: `` `<NAME> ``
- Good for naming constants (can be used like wires)
- Convention dictates that macro names be in all caps
- Can also `` `undef <NAME> ``

Verilog Macros

Flow Control

Text inside flow control is conditionally included in the compiled source file.

- `` `ifdef `` Checks if something is defined
  - `` `else `` Normal else behavior
- `` `endif `` End the if
- `` `ifndef `` Checks if something is not defined

Verilog Macros by Example

```
`define DEBUG
`define LOCKED 1'b0
`define UNLOCKED 1'b1
module turnstile(
    input coin, push,
    input clock, reset
    `ifdef DEBUG
        ,output logic state
    `endif
    );
    `ifndef DEBUG
        logic state;
    `endif
    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
```

Verilog Headers

What is inclusion?

- Paste the code from the specified file where the directive is placed

Where will I use this?

- System defines separated out, e.g.

```
`define ALU_OP_ADD 5'b10101
`define DCACHE_NUM_WAYS 4'b0100
```

- Macro assertion functions, printing functions, etc.
- Note: headers are named with the `.vh` extension

Verilog Headers

Inclusion

- Syntax: `` `include <FILE_NAME> ``
- Pastes the contents of <FILE_NAME> wherever the include appears

Include Guards

- Inside the header...

```
`ifndef __FILE_NAME_H__
`define __FILE_NAME_H__

⋮

`endif
```

Parameters

What is a `parameter`?

- Constant defined inside a module
- Used to set module properties
- Can be overridden on instantiation

How do I use them?

- Definition

```
parameter NUM_CACHE_LINES = 8;
```

Setting Parameters

Overriding

- Set parameters for a module on instantiation
- Allows for different versions different places
- Usage:

```
cache #(.NUM_CACHE_LINES(4)) d_cache(...);
```

defparam

- Set parameters for a module
- Usage:

```
defparam dcache.NUM_CACHE_LINES = 4;
```

Macros v.s. Parameters

Macros

- Possibly globally scoped – namespace collision
- Use for modules that can change, but only have one instance
- Particularly for caches and naming arbitrary constants
- Needs the ` in usage

vs.

Parameters

- Locally scoped – no namespace collision
- Use for modules with many instances at different sizes
- Particularly for `generate` blocks (which are in a later lab)
- Does not need extra characters (like the `)

Array Connections

- Make a simple module a duplicate it several times
- Assume we have a module defintion:
  - `one_bit_addr(a, b, cin, sum, cout);`
- All ports are 1 bit, the first three inputs, last two outputs
- How do we build an eight bit adder?

The Bad Way

```
module eight_bit_addr(
    input en, cin,
    input [7:0] a, b,
    output [7:0] sum,
    output cout);

    wire [6:0] carries;

    one_bit_addr a0(en, a[0], b[0], cin, sum[0], carries[0]);
    one_bit_addr a1(en, a[1], b[1], carries[0], sum[1], carries[1]);
    one_bit_addr a2(en, a[2], b[2], carries[1], sum[2], carries[2]);
    one_bit_addr a3(en, a[3], b[3], carries[2], sum[3], carries[3]);
    one_bit_addr a4(en, a[4], b[4], carries[3], sum[4], carries[4]);
    one_bit_addr a5(en, a[5], b[5], carries[4], sum[5], carries[5]);
    one_bit_addr a6(en, a[6], b[6], carries[5], sum[6], carries[6]);
    one_bit_addr a7(en, a[7], b[7], carries[6], sum[7], cout);
endmodule
```

The Bad Way

- Lots of duplicated code
- Really easy to make mistake
- Now try building a 64-bit adder..., 256?
- There is a one line substitute

The Better Way

```
module eight_bit_addr(
    input en, cin,
    input [7:0] a, b,
    output [7:0] sum,
    output cout);

    wire [6:0] carries;

    one_bit_addr addr [7:0] (
        .en(en), .a(a), .b(b), .cin({carries,cin}),
        .sum(sum), .cout({cout,carries})
    );
endmodule
```

All of the ports in the one_bit_addr module are 1 bit wide. All of the busses we pass are 8 bits wide, so each instantiation of the module will get one, except en which is only 1 bit wide and thus copied to every module.

The (Even) Better Way

```
`define ADDR_WIDTH 8
module eight_bit_addr(
    input en, cin,
    input [(`ADDR_WIDTH-1):0] a, b,
    output [(`ADDR_WIDTH-1):0] sum,
    output cout);

    wire [(`ADDR_WIDTH-2):0] carries;

    one_bit_addr addr [(`ADDR_WIDTH-1):0] (
        .en(en), .a(a),.b(b),.cin({carries,cin}),
        .sum(sum),.cout({cout,carries})
    );
endmodule
```

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

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
```

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 <= #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
```

VS.

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...

Indentation

Why are we interested in indentation?

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

Indentation 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
```

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
```

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
```

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;
```

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.

References


Jon Beaumont (2021)
EECS470 Lab2, Lab3

Thanks