

# ECE4700J Lab 3

## SystemVerilog and Tcl

Haoyang Zhang

UM-SJTU Joint Institute

*zhy-sjtu-jc@sjtu.edu.cn*

June 2, 2022

# Overview

- 1 Administrivia
- 2 SystemVerilog Programming
- 3 Advanced SystemVerilog Features
- 4 Basic Tcl Scripting for Vivado
- 5 Assignments

- Lab 2 Assignment's live demo will be on 8:00pm - 10:30pm, Friday Jun. 3rd (Beijing Time). Lab 2 Assignment code submission will be due on 23:59pm, Jun. 3rd (Beijing Time).
- Homework 1 will be due on 23:59pm, Jun. 6th (Beijing Time).
- Lab 3 Assignment will be released soon today. It will be due on Friday next week.
- Homework 2 will come soon.

- Start early on Lab 3 and Lab 4!
- The Lab 3 Assignment will be harder than former Lab assignments, and much easier than Lab 4 Assignment.
- We will cover hints on both of them in lab lectures.

- 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.

# Lab2 Part B Typo

I've made a typo in the lab2 partB when transforming the original buggy code from the open-source AXI-S repo to the prototype code in the lab assignments. The correct version has been updated on canvas (and also shown below). I'm really sorry for the inconvenience. (Please see Feishu)

- **Specification:** *B\_reg* should be initialized as 0. When *en&condition1* is true, *B\_reg* should be set as 1 (start from the next cycle), and value 1 will be kept until sometime *en&condition2* is true, then *B\_reg*'s value should be set back to 0 (start from the next cycle). It's guaranteed that condition1 and condition2 cannot both be true at the same time. During the time when the value of *B\_reg* is 1, the input *en* signal can be set as 0, but this should not affect the value of *B\_reg* to be 1.<sup>40</sup>
- **Code Implementation:**<sup>41</sup>

```
# B.sv
1  input clock, reset;
2  input en, condition1, condition2;
3  logic B_next;
4  logic B_reg;
5
6  always_comb begin
7      B_next = 1'b0;
8      if (en) begin
9          if (condition1 | B_reg) begin
10             B_next = 1'b1;
11             if (condition2) begin
12                 B_next = 1'b0;
13             end
14         end
15     end
16 end
17
18 always_ff @(posedge clock) begin
19     if (reset) begin
20         B_reg <= 1'b0;
21     end
22     else begin
23         B_reg <= B_next;
24     end
25 end
```

## User-defined Types

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

## About struct

- ▶ A package of signals (wire or logic)
- ▶ Basically follow C conventions
  - ▶ List of signal declarations
  - ▶ Named with `_t` ending

## Syntax

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



# 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;
```

---

## 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 (`;`)

## 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;
```

## 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

---

```
//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 (Good Style)

## 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)

# FSM 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 <= #1 DEFAULT;
        end else begin
            state <= #1 next_state;
        end
    end
endmodule
```

# FSM 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 <= #1 LOCKED;
        else      state <= #1 next_state;
    end
endmodule
```



# 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

# 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

# "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
```

## 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

## 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

# 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 (
            .a(a[i]), .b(b[i]),
            .cin(carries[i]),
            .sum(sum[i]),
            .cout(carries[i+1]));
    end
endgenerate
```

# Generate Blocks

- ▶ 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

## 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
```

# Advanced Assertions

- ▶ Much more interesting (and challenging)
  - ▶ Describe high-level functional correctness of your design...
  - ▶ ...and have simulator check these invariants in the background
- ▶ SystemVerilog supports an entire assertion language (!)
  - ▶ (beyond the scope of what we will do in class)
- ▶ Implication
  - ▶ `s1 |-> s2`
    - ▶ If s1 is true, then s2 must also be true
- ▶ Timing windows
  - ▶ `(a && b) |-> ##[1:3] c;`
    - ▶ On the posedge of the clock, if a and b are true, then 1-3 cycles later, c will be true

## Assertions

- ▶ For more information on assertions, look into the book *A Practical Guide to SystemVerilog Assertions*

# What is Tcl?

- Tcl is shortened form of Tool Command Language. John Ousterhout of the University of California, Berkeley, designed it. It is a combination of a scripting language and its own interpreter that gets embedded to the application, we develop with it.
- It aims at providing ability for programs to interact with other programs and also for acting as an embeddable interpreter.
- It's used commonly in EDA (Electronic Design Automation) tools like Vivado, VCS, Verilator...
- Tcl script files should be named as xxxx.tcl
- A guide on Vivado Tcl: <https://docs.xilinx.com/r/en-US/ug894-vivado-tcl-scripting/Tcl-Scripting-in-Vivado>



# Why we need Tcl?

- It can be used to configure complex variables and environments for your hardware design. (Actually, many tools are command line based, doesn't like Vivado, they don't have a GUI for users.)
- It can be used for batching simulations. (Suppose you have a lot of benchmarks... like lab 4 assignment)



TQL!



我 Tcl

# Basic Tcl Scripting for Vivado

```
# Set necessary variables, variables can be used as strings with syntax " $variable "  
set projectName project_1  
  
# Set Xilinx Vivado output directory  
set outputdir ./ProjectReports  
  
# Make the directory  
file mkdir $outputdir  
  
# Create the Vivado Project using a specific part  
create_project $projectName $outputdir -part xcvu35p-fsvh2892-3-e
```

# Basic Tcl Scripting for Vivado

```
# Add all the design files
add_files -norecurse -scan_for_includes design1.sv design2.sv
# 'Import files' means that you want to copy these files into your Vivado project's output directory
import_files -norecurse ram.v mem_controller.v design1.sv design2.sv

# Add simulation files (testbenches)
add_files -fileset sim_1 -norecurse -scan_for_includes testbench.v
import_files -fileset sim_1 -norecurse testbench.v
```

# Basic Tcl Scripting for Vivado

```
# Set your top module
```

```
set_property top your_top_module [current_fileset]  
update_compile_order -fileset sources_1
```

```
# Set you top simulation module
```

```
set_property top testbench [get_filesets sim_1]  
set_property top_lib xil_defaultlib [get_filesets sim_1]  
update_compile_order -fileset sim_1
```

```
# launch RTL synthesis and wait until it finished
```

```
launch_runs synth_1  
wait_on_run synth_1
```

---

# Basic Tcl Scripting for Vivado

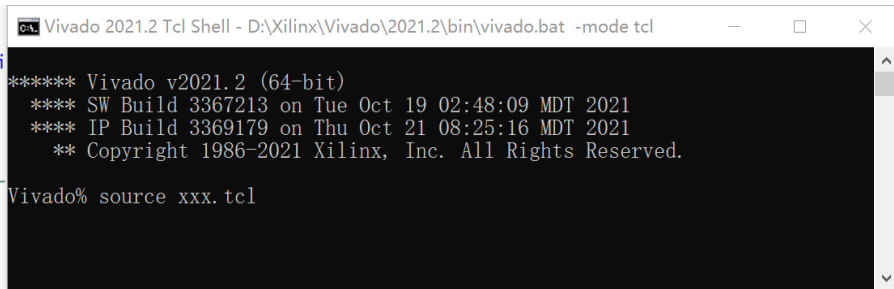
```
# Set your simulation time
set_property -name {xsim.simulate.runtime} -value {10000000ns} -objects [get_filesets sim_1]

# Launch your post-synthesis timing simulation
launch_simulation -mode post-synthesis -type timing

# Close your project and Exit the Vivado Tcl Shell
close_project
exit
```

# How to use Tcl commands?

- First you need to open Vivado Tcl Shell.
- Then you can use tcl commands directly in the shell.
- But how to run a .tcl file?
- Use "source xxxx.tcl" in the shell, or directly use the filename as an argument of Vivado executable in your Windows/Linux Command Shell. (More on Lab 4 lecture)



```
C:\> Vivado 2021.2 Tcl Shell - D:\Xilinx\Vivado\2021.2\bin\vivado.bat -mode tcl

***** Vivado v2021.2 (64-bit)
**** SW Build 3367213 on Tue Oct 19 02:48:09 MDT 2021
**** IP Build 3369179 on Thu Oct 21 08:25:16 MDT 2021
** Copyright 1986-2021 Xilinx, Inc. All Rights Reserved.

Vivado% source xxx.tcl
```

# Lab 3 Assignment Overview

## Part 1: Pipelined Multiplier

- ▶ Change the pipeline depth
- ▶ Synthesize at each size

## Part 2: Integer Square Root

- ▶ Finite state machine implementation
- ▶ Synthesis

# Pipelined Multiplication

## 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

$$\begin{array}{r} 000111 \\ \times \quad 0101 \\ \hline 000111 \\ 000000 \\ 011100 \\ + 000000 \\ \hline 100011 \end{array}$$

## Decimal Multiplication

$$\begin{array}{r} 7 \\ \times \quad 5 \\ \hline 35 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00001011

multiplier: 00000111

partial product: 00000000

$$\begin{array}{r} 00001011 \\ \times 00000111 \\ \hline 00000000 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00001011    << 2  
multiplier: 00000111    >> 2  
partial product: 00100001

$$\begin{array}{r} 00001011 \\ \times 00000111 \\ \hline 00100001 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

multiplicand: 00101100  
multiplier: 00000001  
partial product: 00100001

$$\begin{array}{r} 00101100 \\ \times 00000001 \\ \hline 00000000 \end{array}$$

# Pipelined Multiplication by Example

## Example: 4-stage Pipelined Multiplication

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

$$\begin{array}{r} 00101100 \\ \times 00000001 \\ \hline 00101100 \end{array}$$

## 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?

## 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

# References



Jon Beaumont (2021)

EECS470 Lab3, Lab6



# Thanks