ECE4700J Lab 1 - Getting Started with Vivado and SystemVerilog

Xinting Jiang

UM-SJTU JI

evans_jiang@sjtu.edu.cn

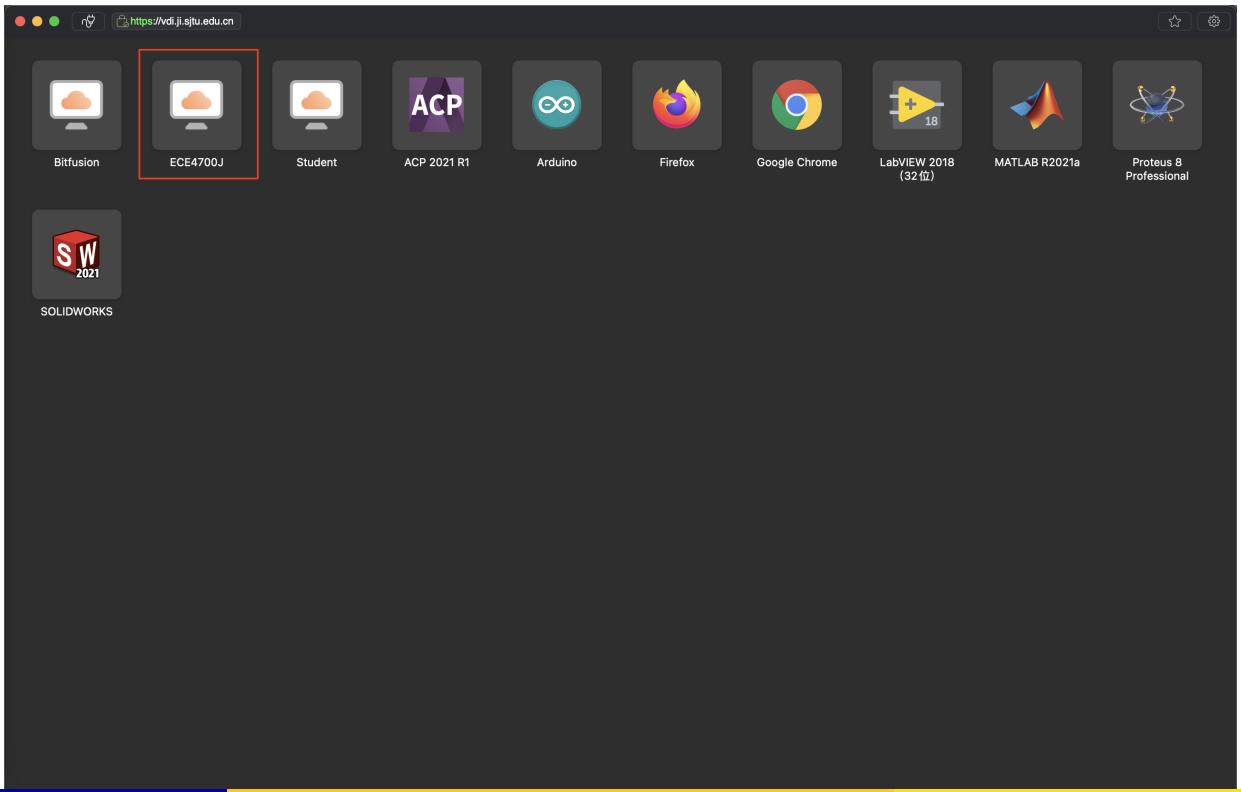
May 19, 2024

Overview

Lab Logistics

ECE4700J Lab Logistics

- Mode: In person only lecture/lab + lab assignment due in 1 or 2 weeks later each lab
- Attendance: Not required but highly recommended
  - Participation points: lecture 3% and lab 3%
- JI virtual desktop (VMware Horizon Client) is provided to access Vivado and Gem5



Need Help?

- Refer to Canvas→Files→Reading Materials for more info about SystemVerilog and Vivado
- ECE4700J Piazza (highly recommended)
- TA's Office Hours:
  - Xinting Jiang: Tue 20:30-21:30 DXY4-406
  - Mingjian Li: Thu 20:00-21:00 LBL326F
- TA's emails
  - Xinting Jiang: evans_jiang@sjtu.edu.cn
  - Mingjian Li: lmjhshxhc@sjtu.edu.cn

Advice

- Start early (Installation, Verilog debugging...)
- Make good use of Piazza and office hours

Lab 1 Administrivia

- Due time: 23:59 on May 28th (Beijing time)
- Late Penalty: 10% / day
- Contents
  - Vmware horizon or Vivado installation (step by step in the manual:)
  - SystemVerilog implementation of a Round Robin arbiter
  - Testbench writing
  - Debugging in SystemVerilog

From Verilog to SystemVerilog

Verilog Recap

- Variables: reg vs. wire
  - multiple bits: `reg [3:0] var1`, `wire [4:0] var2`
  - reg: can store values
  - wire: can change immediately and cannot store values; input and output are initialize as wire
- Blocking assignment ($=$) vs. Non-blocking assigment ($<=$)
- Modules: A part of circuits and can be instantiated multiple times.
  - Best way to instantiate a module:
    `my_module m1(.a_in(a), .b_in(b), .c_out(c))`
- Refer to https://hdlbits.01xz.net/ for more detailed tutorial and practice!

Verilog Recap

- Basic syntax
  - Code in blocks: Start with `begin`, end with `end`
  - `assign`: Assign values to `wire` variables.
  - `always`: `always @(cond/var)`, When `cond` is satisfied or `var` is changed, the code in this always block will be executed
  - `initial`: Commonly used in testbench file to initialize stimuli. Write sequential code here.
  - `if ... else ...`: Similar to C
  - `case (var) begin ... endcase`: Similar to C

Verilog Recap

- Examples

```
module 1b_register(
    input rst,
    input clk,
    input in,
    output out
);
    always @(posedge clk, posedge rst) begin
        if (rst) begin
            out <= 1'b0;
        end
        else begin
            out <= clk;
        end
    end
endmodule
```

Figure: Simple 1 bit register

```
always @(*) begin
    case (var)
        3'b0: begin
            // do sth
        end
        3'b101: begin
            // do sth
        end
        3'b111: begin
            // do sth
        end
        default: begin
            // do sth
        end
    endcase
end
```

Figure: "case" example

From Verilog To SystemVerilog

- `wire` - `wire/logic`
  - Also called nets. Cannot hold states.
- `reg` - `logic`
  - Holds state. Turns into flip-flops
- `always` - `always_comb`, `always_ff`
  - always_comb: Used for combinational logic. Replace always @(*)
  - always_ff: Used for sequential logic.
- `enum` - really useful!
  - improve code readability and maintainability by using meaningful names instead of raw integer values
- More advanced features will be covered in the next lab:)

Changes in SystemVerilog

- Examples

```
module 1b_register(
    input rst,
    input clk,
    input in,
    output out
);
    always_ff @(posedge clk, posedge rst) begin
        if (rst) begin
            out <= 1'b0;
        end
        else begin
            out <= clk;
        end
    end
endmodule
```

Figure: 1b register in SystemVerilog

```
always_comb begin
    case (var)
        3'b0: begin
            // do sth
        end
        3'b101: begin
            // do sth
        end
        3'b111: begin
            // do sth
        end
        default: begin
            // do sth
        end
    endcase
end
```

Figure: "case" example in SystemVerilog

Implement a Finite State Machine (Moore)

- Step 1: Assign output based on the current state .
- Step 2: Identify state transfer function and implement in a always comb block.
- Step 3: Complete sequential logic, remember to use non-blocking assignment.

A Simple Arbiter

```
module arbiter (
    input clock, reset,
    input req_a, req_b,  // request lines
    output gnt_a, gnt_b  // grant lines
);
```

Figure: Module declaration for the simple 2-bit arbiter

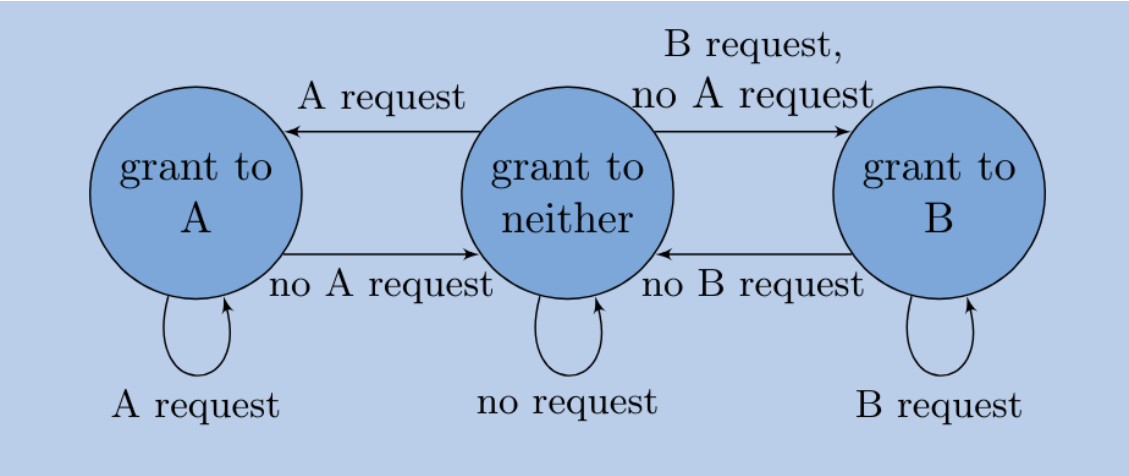


Figure: FSM for the simple 2-bit arbiter

A Simple Arbiter

- Define state variable using enum type:

```
// Define states using enum for readability
enum logic [1:0] { IDLE, GRANT_A, GRANT_B } State;
```

- Note that we still need two bits for this type.

A Simple Arbiter

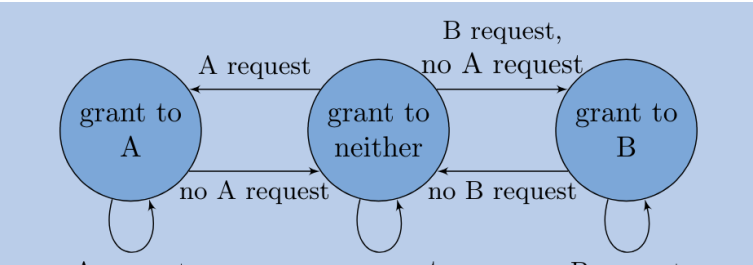
- Step 1: Assign output based on the current state.

```
module arbiter (
  input clock, reset, req_a, req_b,
  output gnt_a, gnt_b
);
  State current_state, next_state;
  assign gnt_a =(current_state==GRANT_A) ? 1'b1 : 1'b0;
  assign gnt_b =(current_state==GRANT_B) ? 1'b1 : 1'b0;
```

A Simple Arbiter

- Step 2: Implement the state transfer function.

```
always_comb begin
    case (current_state)
      IDLE: begin
        // Prioritize A if both request
        next_state=req_a?GRANT_A:(req_b?GRANT_B:IDLE);
      end
      GRANT_A: next_state = req_a ? GRANT_A : IDLE;
      GRANT_B: next_state = req_b ? GRANT_B : IDLE;
      // Handle unexpected state
      default: next_state = IDLE;
    endcase
end
```



A Simple Arbiter

- Step 3: complete sequential logic.

```
always_ff @(posedge clock) begin
    if (reset) begin
      current_state <= IDLE;
    end else begin
      current_state <= next_state;
    end
end
```

A Simple Arbiter

```
module arbiter (
  input clock, reset, req_a, req_b,
  output gnt_a, gnt_b
);
State current_state, next_state;
assign gnt_a =(current_state==GRANT_A) ? 1'b1 : 1'b0;
assign gnt_b =(current_state==GRANT_B) ? 1'b1 : 1'b0;

always_comb begin
    case (current_state)
      IDLE: begin
        next_state=req_a?GRANT_A:(req_b?GRANT_B:IDLE);
      end
      GRANT_A: next_state = req_a ? GRANT_A : IDLE;
      GRANT_B: next_state = req_b ? GRANT_B : IDLE;
      default: next_state = IDLE;
    endcase
end
```

A Simple Arbiter

```
...
always_ff @(posedge clock) begin
    if (reset) begin
      current_state <= IDLE;
    end else begin
      current_state <= next_state;
    end
end
endmodule
```

Synthesizable Design

Synthesis

- Definition: Given specific hardware modules library, the process that changes the behavioral Verilog into fully structural Verilog.

Synthesis Tips - Latches

- Try to avoid latches.

```
always_comb begin
    if (cond) begin
        next_x = y;
    end
end
```

$\Rightarrow$

```
always_comb begin
    if (cond) begin
        next_x = y;
    end
    else begin
        next_x = x;
    end
end
```

```
always_comb begin
    next_x = x;
    if (cond) begin
        next_x = y;
    end
end
```

Synthesis Tips - Combinational Logic

- Avoid feedback (combinatorial loops)
- Always blocks for combinational logic should
  - Be `always_comb` blocks
  - Use the blocking assignment operator =
- All variables assigned on all paths
  - Default values
  - `if (...)` paired with an `else`
  - `case` needs to have `default`

Synthesis Tips - Sequential Logic

- Avoid clock- and reset-gating
- Always blocks for sequential logic should be `always_ff @(posedge clock)` block
- No path should set a variable more than once
- Reset all variables used in the block

Synthesis Tips - Flow Control

- Synthesizable flow control: `if...else...`, `case`
  - `for` ?
- Unsynthesizable flow control: `while`, `repeat`, `forever`

Post-Synthesis Simulations

- Simulations of the synthesized netlist
- Take account for FPGA setup time and delays within netlist
- Will be different from your behavioral simulation results
- Unsynthesizable tricks will not work (e.g. delays #5)
- You need to do post-synthesis simulation for a part of your Lab 1 assignment and for most of your future lab assignments:)

Testbench Writing

Features of Testbenches

- Unsynthesizable: So you can use the unsynthesizable constructs introduced above!
- Programmatic: More like doing a software programming. We have `function`, `task`, `classes`...
- There are good examples in Lab1 starter file, please have a look!

Steps to Setup Your Testbenches

1. Declare inputs and outputs for modules being tested
2. Instantiate the modules
3. Setup a clock driver (if needed)
4. Setup a correctness checking function (if necessary/possible)
5. Inside a `initial` block,
   - Assign default values to all inputs, including any available reset signals
   - Print out import signals
   - Describe changes in input, using good testing practice

Useful System Task or Functions

- `$random`: Generate an 8-bit random number
  `$urandom_range(100, 0)`: Generate a number within range 0-100
- `$monitor`: Used in testbenches. Prints every time an argument changes. Bad for large projects.
  e.g. `$monitor("format", signal_name, ...)`
- `$display`: Used both in testbenches or designs, but not after synthesis. Print only once.
  e.g. `$display("format", signal_name, ...)`
- `$time`: The current simulation time in a 64-bit integer.
- `$reset`: Reset the simulation to the beginning.
- `$finish`: Exit the simulator, return to the terminal

Write Your Own Tasks and Functions!

- `task`: can have delays (e.g. `#5`), can have timing info (e.g. `@(negedge clk)`)

  e.g.

```
task exit_on_error;
begin
    $display("@@@ Incorrect at time %4.0f", $time);
    $display("ENDING TESTBENCH : ERROR !");
    $finish;
end
endtask
```

Write Your Own Tasks and Functions!

- `function`: No delays, no timing. Can return values, unlike `task`

  e.g.

```
function [2:0] EXPECTED_OUT;
    begin
        EXPECTED_OUT = (a1_state==2'b01 ? 3'b100 :
            (a1_state==2'b10 ? 3'b010 :
            (a1_state==2'b11 ? 3'b001 : 3'b000)))
            & request;
    end
endfunction
```

Reference

[1] Haoyang Zhang, ECE4700J SU 2022 Lab1 & Lab2
[2] Runxi Wang, ECE4700J SU 2023 Lab1
[3] Jon Beaumont, EECS470 Lab1 & Lab2 & Lab3, University of Michigan

Thank you!