ECE4700J Lab 1
Verilog: Hardware Description Language

Haoyang Zhang

UM-SJTU Joint Institute

zhy-sjtu-jc@sjtu.edu.cn

May 18, 2022

Overview

What?

- Lab 1 - Lab 3: SystemVerilog Design Tutorial and Environment Setup
- Lab 4: Introduction to RISC-V ISA and Pipeline
- Lab 5: Introduction to Chisel Language
- Lab 6: Introduction to Gem5 Simulator

Where? When?

- Due to the completely online nature
  - Lab assignments will be released shortly before the start of the first lab each week.
  - A recording of the lab will be released after the lab lecture.
  - Please refer to the slides and recording for demonstrations and tips!
- Lab attendance is optional but strongly recommended!
  - Lab lectures will be at Thursday 10:00 am to 11:30 am (Beijing Time).
  - The first three Lab assignments must be checked off during a live meeting with the TA (Lab live demo). They are scheduled on Friday 8:00 pm - 10:00 pm (Beijing Time) of the week after they are assigned.
  - Labs are due by the end of lab the week after they are assigned.

Help?

- ECE4700J Piazza (Recommended)
  - Most of your lab/project related questions should be asked here so that other people can benefit from the answer.
  - Reminder: Please do not post program code in public questions.
- ECE4700J OH: on Friday 8:00 pm - 9:00 pm Beijing Time. (First 3 OHs can be used for live demo)
- ECE4700J Staff Email

Lab Assignments

- All lab assignments are individual assignments.
  - Lab1 - Simple Arbitor Design (10%)
  - Lab2 - Some Verilog Debugging (10%)
  - Lab3 - A Pipelined Integer Square Root Module (15%)
  - Lab4 - RISCV Five-Stage Pipeline Optimization (30%)
  - Lab5 - FIFO and CAM Design (15%)
  - Lab6 - Gem5 Simulation of Spectre V1 Attack (20%)
- Some lab assignments also contain optional assignments. If you implement them correctly, you can get up to 10% bonus points to the total score of lab assignments.

Advice

- These lab assignments will take a non-trivial amount of time, especially if you're not a Verilog guru.
- You should start them early. Seriously...

Administrivia

- Lab 1 is due Friday 11:59pm May 27th, 2022 (Beijing Time)

Intro to Verilog

What is Verilog?

- Hardware Description Language - IEEE 1364-2005
  - Superseded by SystemVerilog - IEEE 1800-2009
- Two Forms
  1. Behavioral
  2. Structural
- It can be built into hardware. If you can't think of at least one (inefficient) way to build it, it might not be good.

Why do I care?

- We use Behavioral Verilog to do computer architecture here.
- Semiconductor Industry Standard (VHDL is also common, more so in Europe)

Verilog to SystemVerilog

- wire - wire/logic
- reg - logic
- always (*) - always_comb
- always (posedge clock) - always_ff @(posedge clock)
- More on SystemVerilog later...

The Difference Between Behavioral and Structural Verilog

Behavioral Verilog

- Describes function of design
- Abstractions
  - Arithmetic operations (+, -, *, /)
  - Logical operations (&, |, ^, ~)

Structural Verilog

- Describes construction of design
- No abstraction
- Uses modules, corresponding to physical devices, for everything

Suppose we want to build an adder?

Structural Verilog by Example

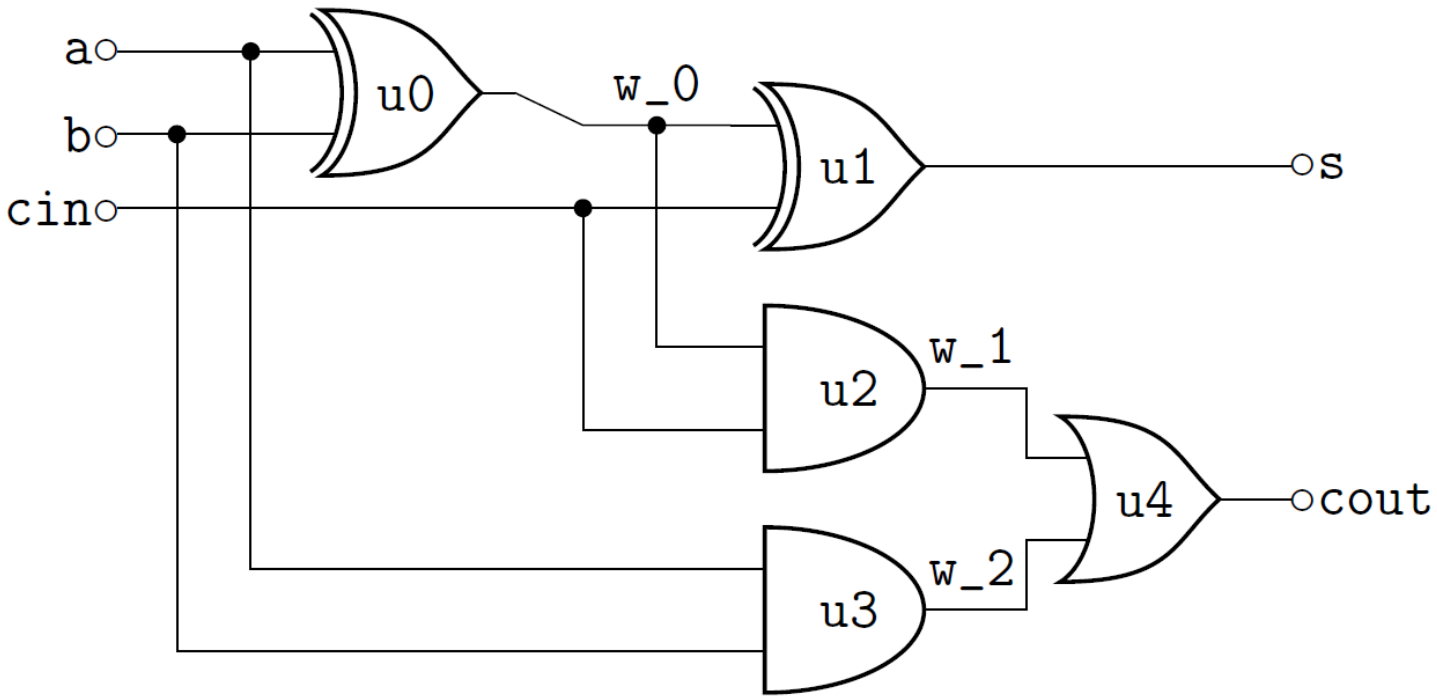

Figure: 1-bit Full Adder

Structural Verilog by Example

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    wire w_0,w_1,w_2;
    xor u0(w_0,a,b);
    xor u1(sum,w_0,cin);
    and u2(w_1,w_0,cin);
    and u3(w_2,a,b);
    or u4(cout,w_1,w_2);
endmodule
```

Behavioral Verilog by Example

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

Behavioral Verilog by Example

```
module one_bit_adder(
    input logic a,b,cin,
    output logic sum,cout);

    always_comb
    begin
        sum = a ^ b ^ cin;
        cout = 1'b0;
        if ((a ^ b) & cin) | (a & b))
            cout = 1'b1;
    end
endmodule
```

Synthesis

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

Data Types

Synthesizable Data Types

`wires` Also called nets

```
wire a_wire;
wire [3:0] another_4bit_wire;
```

- Cannot hold state

`logic` Replaced `reg` in SystemVerilog

```
logic [7:0] an_8bit_register;
reg a_register;
```

- Holds state, might turn into flip-flops
- Less confusing than using `reg` with combinational logic (coming up...)

Data Types

Unsynthesizable Data Types

`integer` Signed 32-bit variable

`time` Unsigned 64-bit variable

`real` Double-precision floating point variable

Types of Values

Four State Logic

- `0` False, low
- `1` True, high
- `Z` High-impedance, unconnected net
- `X` Unknown, invalid, don't care

Values

Literals/Constants

- Written in the format <bitwidth>'<base><constant>
- Options for <base> are...
  - b Binary
  - o Octal
  - d Decimal
  - h Hexadecimal

```
assign an_8bit_register = 8'b10101111;
assign a_32bit_wire = 32'hABCD_EF01;
assign a_4bit_logic = 4'hE;
```

Verilog Operators

Arithmetic	
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulus
**	Exponentiation
Bitwise	
~	Complement
&	And
|	Or
~|	Nor
^	Xor
~^	Xnor
Logical	
!	Complement
&&	And
||	Or

Shift	
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift
Relational	
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!=	Inequality
!==	4-state inequality
==	Equality
===	4-state equality
Special	
{,}	Concatenation
{n{m}}	Replication
?:	Ternary

Setting Values

assign Statements

- One line descriptions of combinational logic
- Left hand side must be a wire (SystemVerilog allows assign statements on logic type)
- Right hand side can be any one line verilog expression
- Including (possibly nested) ternary (?:)

Example

```
module one_bit_adder(
    input wire a,b,cin,
    output wire sum,cout);
    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

Setting Values

`always` Blocks

- Contents of `always` blocks are executed whenever anything in the sensitivity list happens
- Two main types in this class...
  - `always_comb`
    - implied sensitivity lists of every signal inside the block
    - Used for combinational logic. Replaced `always @*`
  - `always_ff @(posedge clk)`
    - sensitivity list containing only the positive transition of the `clk` signal
    - Used for sequential logic
- All left hand side signals need to be `logic` type.

Always Block Examples

Combinational Block

```
always_comb
begin
    x = a + b;
    y = x + 8'h5;
end
```

Sequential Block

```
always_ff @(posedge clk)
begin
    x <= #1 next_x;
    y <= #1 next_y;
end
```

Blocking vs. Nonblocking Assignments

Blocking Assignment

- Combinational Blocks
- Each assignment is processed in order, earlier assignments block later ones
- Uses the = operator

vs.

Nonblocking Assignment

- Sequential Blocks
- All assignments occur "simultaneously," delays are necessary for accurate simulation
- Uses the <= operator

Blocking vs. Nonblocking Assignments by Example

Blocking Example

```
always_comb
begin
    x = new_val1;
    y = new_val2;
    sum = x + y;
end
```

- Behave exactly as expected
- Standard combinational logic

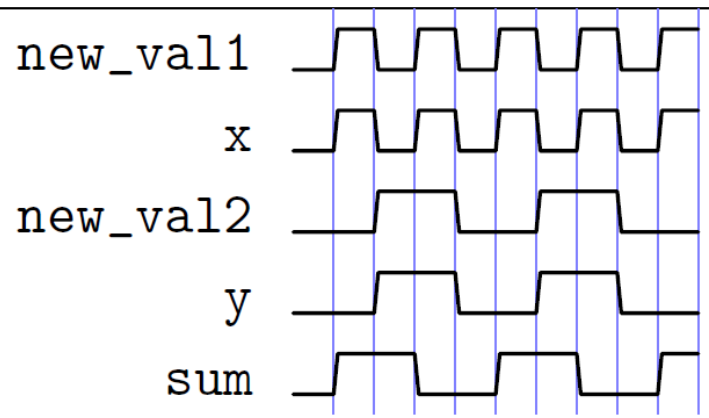

Figure: Timing diagram for the above example.

Blocking vs. Nonblocking Assignments by Example

Nonblocking Example

```
always_ff @(posedge clock)
begin
    x <= #1 new_val1;
    y <= #1 new_val2;
    sum <= #1 x + y;
end
```

- What changes between these two examples?
- Nonblocking means that sum lags a cycle behind the other two signals

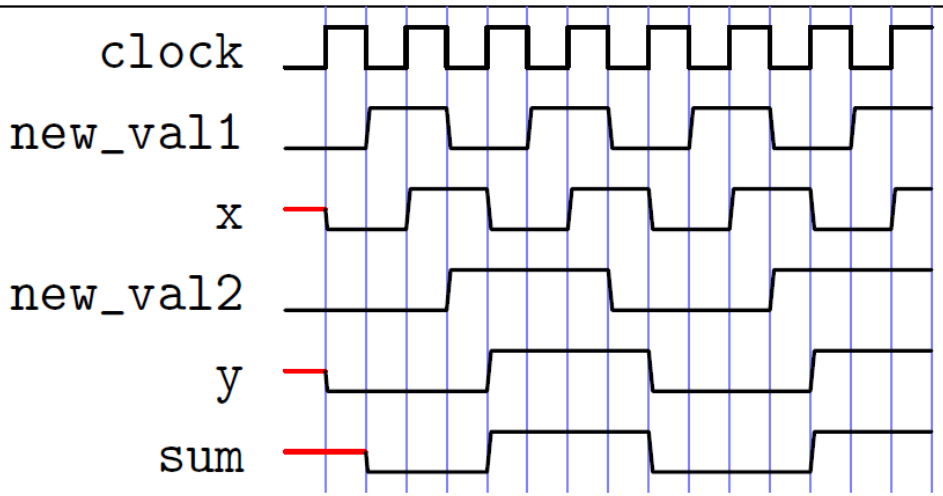

Figure: Timing diagram for the above example.

Blocking vs. Nonblocking Assignments by Example

Bad Example

```
always_ff @(posedge clock)
begin
    x <= #1 y;
    z = x;
end
```

- z is updated after x
- z updates on negedge clock

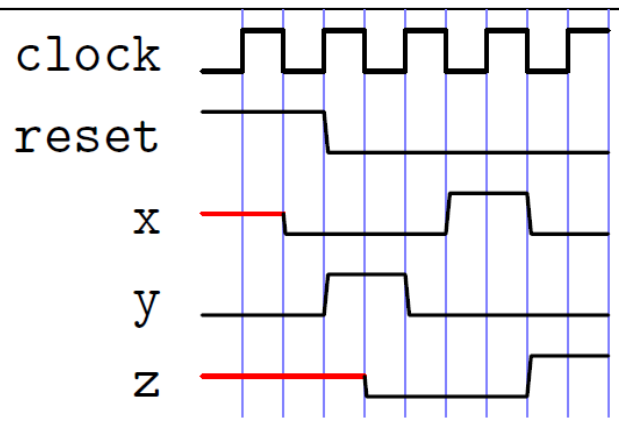

Figure: Timing diagram for the above example.

Synthesis Tips

Latches

- What is a latch?
  - Memory device without a clock
- Generated by a synthesis tool when a net needs to hold state without being clocked (combinational logic)
- Generally bad, unless designed in intentionally
- Unnecessary in this class

Synthesis Tips

Latches

- Always assign every variable on every path
- This code generates a latch
- Why does this happen?

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

Synthesis Tips

Possible Solutions to Latches

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

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

Modules

Intro to Modules

- Basic organizational unit in Verilog
- Can be reused

Module Example

```
module my_simple_mux(
    input wire select_in, a_in, b_in; //inputs listed
    output wire muxed_out); //outputs listed
    assign muxed_out = select_in ? b_in : a_in;
endmodule
```

Modules

Writing Modules

- Inputs and outputs must be listed, including size and type
  format: `<dir> <type> <[WIDTH-1:0]> <name>;`
  e.g. `output logic [31:0] addr;`
- In module declaration line or after it, inside the module

Instantiating Modules

- Two methods of instantiation
  1. e.g. `my_simple_mux m1(.a_in(a),.b_in(b),`
     `.select_in(s),.muxed_out(m));`
  2. e.g. `my_simple_mux m1(a,b,s,m);`
- The former is much safer...
- Introspection (in testbenches): `module.submodule.signal`

How to Design with Verilog

- Remember – Behavioral Verilog implies no specific hardware design
- But, it has to be synthesizable
- Better be able to build it somehow

Keys to Synthesizability

Combinational Logic

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

Keys to Synthesizability

Sequential Logic

- Avoid clock- and reset-gating
- Always blocks should
  - Be `always_ff @(posedge clock)` blocks
  - Use the nonblocking assignment operator, with a delay `<= #1`
- No path should set a variable more than once
- Reset all variables used in the block
- `//synopsys sync_set_reset "reset"`

Flow Control

All Flow Control

- Can only be used inside procedural blocks (`always`, `initial`, `task`, `function`)
- Encapsulate multiline assignments with `begin...end`
- Remember to assign on all paths

Synthesizable Flow Control

- `if/else`
- `case`

Flow Control

Unsynthesizable Flow Control

- Useful in testbenches
- For example...
  - `for`
  - `while`
  - `repeat`
  - `forever`

Flow Control by Example

Synthesizable Flow Control Example

```
always_comb
begin
    if (muxy == 1'b0)
        y = a;
    else
        y = b;
end
```

The Ternary Alternative

```
wire y;
assign y = muxy ? b : a;
```

Testing

What is a test bench?

- Provides inputs to one or more modules
- Checks that corresponding output makes sense
- Basic building block of Verilog testing

Why do I care?

- Finding bugs in a single module is hard...
- But not as hard as finding bugs after combining many modules
- Better test benches tend to result in higher project scores

Intro to Test Benches

Features of the Test Bench

- Unsynthesized
  - Remember unsynthesizable constructs? This is where they're used.
  - In particular, unsynthesizable flow control is useful in testbenches (e.g. `for`, `while`)
- Programmatic
  - Many programmatic, rather than hardware design, features are available e.g. functions, tasks, classes (in SystemVerilog)

Anatomy of a Test Bench

A good test bench should, in order...

1. Declare inputs and outputs for the module(s) being tested
2. Instantiate the module (possibly under the name `DUT` for Device Under Test)
3. Setup a clock driver (if necessary)
4. Setup a correctness checking function (if necessary/possible)
5. Inside an `initial` block...
   - 5.1 Assign default values to all inputs, including asserting any available `reset` signal
   - 5.2 `$monitor` or `$display` important signals
   - 5.3 Describe changes in input, using good testing practice

Unsynthesizable Procedural Blocks

`initial` Blocks

- Procedural blocks, just like `always`
- Contents are simulated once at the beginning of a simulation
- Used to set values inside a test bench
- Should only be used in test benches

Unsynthesizable Procedural Blocks

`initial` Block Example

```
initial
begin
    @(negedge clock);
    reset = 1'b1;
    in0 = 1'b0;
    in1 = 1'b1;
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    in0 = 1'b1;
    ...
end
```

List of System Tasks and Functions

`$monitor` Used in test benches. Prints every time an argument changes. Very bad for large projects.
e.g. `$monitor("format",signal,...)`

`$display` Can be used in either test benches or design, but not after synthesis. Prints once. Not the best debugging technique for significant projects.
e.g. `$display("format",signal,...)`

`$strobe` Like display, but prints at the end of the current simulation time unit.
e.g. `$strobe("format",signal,...)`

`$time` The current simulation time as a 64 bit integer.

`$reset` Resets the simulation to the beginning.

`$finish` Exit the simulator, return to terminal.

Test Benches by Example

Test Bench Setup

```
module testbench;
    logic clock, reset, taken, transition, prediction;

    two_bit_predictor(
        .clock(clock),
        .reset(reset),
        .taken(taken),
        .transition(transition),
        .prediction(prediction));

    always
    begin
        clock = #5 ~clock;
    end
```

Test Benches by Example

Test Bench Test Cases

```
initial
begin
    $monitor("Time:%4.0f clock:%b reset:%b taken:%b trans:%b"
        "pred:%b", $time, clock, reset, taken,
        transition, prediction);
    clock = 1'b1;
    reset = 1'b1;
    taken = 1'b1;
    transition = 1'b1;
    @(negedge clock);
    @(negedge clock);
    reset = 1'b0;
    @(negedge clock);
    taken = 1'b1;
    @(negedge clock);
    ...
    $finish;
end
```

Test Benches Tips

Remember to...

- Initialize all module inputs
- Then assert reset
- Use `@(negedge clock)` when changing inputs to avoid race conditions

Vivado Synthesis

- Learn to see Vivado Synthesis Reports
- Avoid Latch
- Avoid timing violation
- Demo

References


Jon Beaumont (2021)
EECS470 Lab1

Thanks