EECS 470 Lab 1

Verilog: Hardware Description Language

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

August 28, 2025

Overview

EECS 470

HDL Introduction

Verilog Basics
- Assignments
- Modules
- Flow Control
- Synthesis

Testing

Misc Tips

Assignments

Appendix

Help?

- Contact Information
  - EECS 470 Staff Email - eecs470staff@umich.edu
  - Emails sent to the above address will go to all instructors so we can respond to your questions faster.
  - EECS 470 Piazza (click for link)
    - Most of your 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.
- See up-to-date office hours on the course website.

Where? When?

- Lab Logistics
  - Labs will be released shortly before the start of the first lab each week.
  - Due one week after the lab is released.
  - Please refer to the slides and recording for demonstrations and tips!
- Lab attendance is optional but strongly recommended!
  - Three Lab Sections
    - 011 Fridays, 10:30 am to 12:30 pm (134 NAME)
    - 012 Fridays, 12:30 pm to 2:30 pm (1620 BEYSTER)
    - 013 Thursdays, 4:30 pm to 6:30 pm (2517 GGBL)
  - You can attend any of the lab sections. If the lab becomes very busy, please try to attend your own lab section hours (within reason).
  - Labs assignments must be checked off during a live meeting with an instructor. If you are unable to get checked off during lab, you can also get checked off during any office hours.
  - Labs are due by the end of lab the week after they are assigned.

What?

- Lab 1 – Verilog Introduction
- Lab 2 – The Build System
- Lab 3 – Style
- Lab 4 – Scripting
- Bonus – Final Project Tips
- Lab 5 – SystemVerilog
- Bonus – Industry and Group Work
- Lab 6 – Memory (no assignment)
- Lab 7 – LSQ (no assignment)

Projects

Individual Verilog Projects

- Project 1 – Priority Selectors (2%)
- Project 2 – Pipelined Multiplier, Integer Square Root (3%)
- Project 3 – Verisimple 5-stage Pipeline (5%)

Group Project

- Final Project – Out-of-Order Processor (35%)

Advice

- These projects will take a non-trivial amount of time, especially if you're not a Verilog guru.
- You should start them early. Seriously...
- Especially Project 3!

Final Project

- RISCV-V ISA
  - An open source ISA that has commercial products
  - Better software support, education friendly and has various extensions that include additional functionality
- Groups of 5
  - Can choose groups or do random selection
  - You'll be spending hundreds of hours together this semester ...
- Heavy Workload
  - 100 hours/member, minimum
  - 150 to 300 hours/member, more realistically
  - This is a lower bound, not an upper bound...
- Class is heavily loaded to the end of the term

Administrivia

- Project 1 is due Monday 8th September, 2025 11:59 PM (turn in via submission script)
- Homework 1 is due Wednesday, 10st September, 2025 11:59 PM (turn in via Gradescope)
- Lab 1 is due Friday, 5th September, 2025 11:59 PM (turn in via Gradescope)

HDL Intro

Goal

The goal of a hardware description language (HDL) like Verilog is to describe hardware in an readable, and scalable manner.

Motivation

Perks of using HDLs:

- Complex logic gates can be abstracted into simple, readable "code"
- Can simulate and test hardware designs before layout
- Create modular / hierarchical designs

Before We Begin

Things to keep in mind

- **Everything you write turns into hardware**
  - Every operation you add has a cost
- Hardware needs to be **synthesizable**
- Each choice involves a power, performance, and area (PPA) tradeoff
  - Faster often (but not always) means more area and power
  - We are only grading on performance, but power and area are equally important in industry

Things you don't need to think about

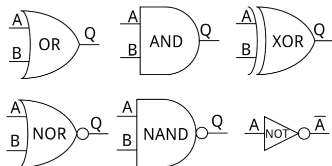
- Layout (take EECS 427 for that)
- Formal verification (although we may allude to this)
- Communication protocols (AXI, APB, PCIE, USB, ...)
- Clock trees, uniform power format (UPF), thermals, I/O, debug ...

Basic Hardware Building Blocks

Remember 270? Everything you code in 470 becomes those building blocks

Basic Blocks

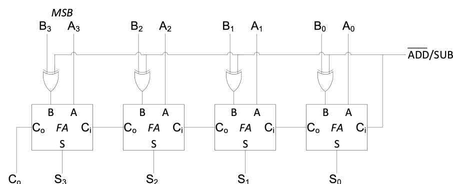
- Logic gates (NOT, AND, OR, XOR, ...)
- Muxes
- Flip-flops and registers



https://learn.sparkfun.com/tutorials/logicblocks–digital-logic-introduction/all

Arithmetic Blocks

- Adders
- Shifters
- Multipliers
- Encoders / Decoders



Verilog Basics

Most direct: Structural Verilog

- Exactly describe logic gates
- Can quickly become very difficult to understand

```
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);
```

What We Write: Behavioral Verilog

- Describes more high level behavior with operators and flow control
- Synthesizes into structural Verilog

```
assign sum = a ^ b ^ cin;
assign cout = ((a ^ b) & cin) | a & b;
```

Data Types

Wires and Registers

- Wires and registers are really the only two types in Verilog
- **Wires** transmit data and hold no state
  - Driven as *combinational* logic
- **Registers** (aka multi-bit flops) keep state to store values
  - Driven as *sequential* logic
- Let Verilog choose the correct one with the `logic` keyword
  - But it's very good to know what type you are actually dealing with

Larger Structures

- It is possible to create larger structures
- Structs and multidimensional arrays will be very helpful as you progress
  - Ex: `logic [7:0][31:0] array` is a, 8x32 array of wires/registers

Setting Values

assign statements

- Describes combinational logic driving wires
- Has to be on one line

```
assign a   = 1;
assign b   = c & d;
assign out = sel ? a : b;
```

Setting Values

always Blocks

- Can create more complex logic structures
- Two types generally used:
  - `always_comb`: Combinational Logic
    - Executed whenever any input to the block changes
    - Left-hand side becomes wires
  - `always_ff @(posedge clock)`: Sequential Logic
    - Left-hand side becomes flops updated on positive edge of clock
- All left hand side signals need to be `logic` type.

```
always_comb begin
    a   = 1;
    b   = c & d;
    out = sel ? a : b;
end
```

Blocking vs. Non-blocking

Two versions of “=”

- Why do we need both?
  - We are used to “code” happening sequentially
  - But we need a way to describe hardware doing things in parallel
- This tends to confuse people the most when learning Verilog

Blocking Assignment (=)

- Each assignment is processed in order
- Combinational Blocks
- Think: A series of combinational gates all following each other

vs.

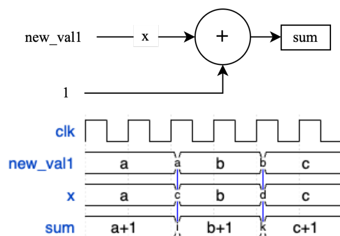
Nonblocking Assignment (<=)

- All assignments occur “simultaneously”
- Sequential Blocks
- Think: A bunch of flops all updating at the same time

Example

Blocking Assignment (=)

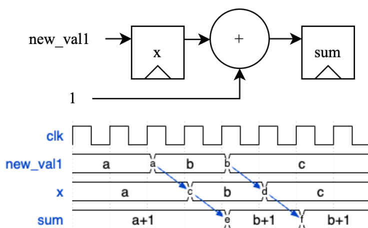
```
always_comb begin
    x   = new_val1;
    sum = x + 1;
end
```



Updates propagate instantly

Nonblocking Assignment (<=)

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



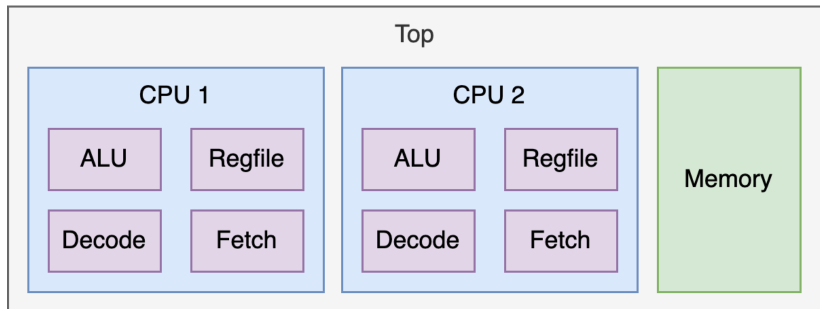
Updates happen at every clock edge

Modules

Why Modules?

- Keeps your code organized! Allows you to abstract functionality
- Can be parametrized and reused

Example Hierarchy



Module Example

Combinational

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

Sequential

```
module my_sequential_mux(
    input  logic clock, reset;
    input  logic select_in, a_in, b_in; //inputs
    output logic muxed_out);             //outputs
    always_ff @(posedge clock) begin
        if (reset) muxed_out <= 1'b0;
        else       muxed_out <= select_in ? b_in : a_in;
    end
endmodule
```

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 (avoids latches)

Synthesizable Flow Control

Helps abstract more complex structures

- `if/else`
- `case`
- `for` (but be careful)

Unsythesizable Flow Control

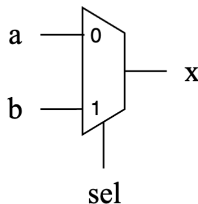
Useful in testbenches

- `while`
- `repeat`
- `forever`

Flow Control Examples

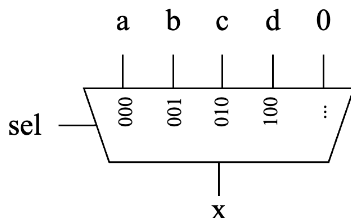
if/else

```
always_comb begin
    if (sel) x = b;
    else     x = a;
end
```



case

```
always_comb begin
    case(sel)
    3'b000:  x = a;
    3'b001:  x = b;
    3'b010:  x = c;
    3'b100:  x = d;
    default: x = 0;
    endcase
end
```



A Note on For Loops

Can we use them?

- For loops are allowed!
- Use them as a shorthand for duplicate hardware
- They expand when synthesized

Best Practices

- Bound on iterations must be constant
- When possible, keep iterations independent
  - Otherwise, you get long chains between loops
- Keep the body of the loop small
- Dependent for loops can often be replaced by priority selectors (see: Project 1)

Synthesis

Everything you design must be able to become hardware

- Although it looks like code, it becomes hardware
- Some things cannot be turned into hardware

Exception: test benches do not need to synthesize

- Will cover this in a few slides

More details in later labs

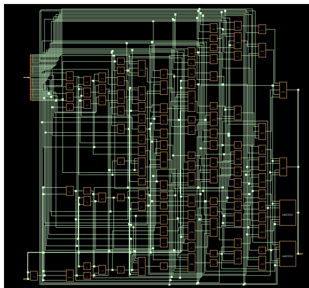
- This is just a quick intro

For Loops: Synthesized

Dependent For Loop

Creates **series** hardware

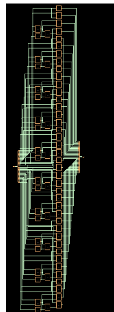
```
for (int i=0; i < LOOPS; i++)
begin
    out = x[i] + 1;
    if (i == a) break;
end
```



Independent For Loop

Creates **parallel** hardware

```
for (int i=0; i < LOOPS; i++)
begin
    out[i] = x[i] + 1;
end
```



Synthesis Tips

General

- Avoid feedback (combinatorial loops)
- Avoid clock- and reset-gating
- No path should set a variable more than once

Reset Values

- Flops need to be reset to a known state
- Otherwise, unknown (`x`) values will propagate

Latches

- Always assign every variable on every path
- Best practice: give every signal a default value at the start of an `always_comb` block

Understanding Check

Why Will This Not Synthesize?

Discuss with you neighbor for a few minutes

```
logic [15:0] x, en;
logic        x_new;

assign x_new = x[15];

always_comb begin
    for (int i = 0; i < 16; i++) begin
        if (i == 0) x[i] = x_new;
        else        x[i] = en[i] ? x[i-1] : x[i];
    end
end
```

Bonus: How can you make this synthesize?

Testing

*Note: Will discuss more in later labs

What is a test bench?

- Lets you test individual modules
- 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
  - Can use unsynthesizable constructs. For example:
    - Delays (ex: #5 to delay 5 ns)
    - `tasks` (for organization and reuse)
    - `while` loops (to simplify flow control)
    - `initial` blocks (for ordered test cases)
- Programmatic
  - Many programmatic, rather than hardware design, features are available e.g. functions, tasks, classes (in SystemVerilog)
  - Allows you to re-write logic in a different way
- Optional: Embed assertions to continually verify behavior

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

Test Benches by Example

```
module testbench;

    logic [3:0] in;
    logic out;
    logic tb_out;

    // dut stands for Device Under Test, the module we're testing
    and4 dut(
        .in  (in),
        .out (out)
    );

    // the prepend ampersand operator ANDs all bits of a variable
    assign tb_out = &in;

    assign correct = (out == tb_out);
```

Test Benches by Example

```
always @(correct) begin
    #2
    if (!correct) begin
        $display("@@@ Incorrect at time %4.0f", $time);
        $display("@@@ in:%b out:%b", in, out);
        $display("@@@ expected result=%b", tb_out);
        $finish;
    end
end

initial begin
    $monitor("Time:%4.0f in:%b out:%b", $time, in, out);

    in = 4'b0000; #5
    in = 4'b1100; #5
    <More test cases...>

    $display("@@@ Passed");
    $finish;
end
```

Misc Tidbits

Bit Widths

- Numbers default to 32 bit numbers
- Can manually set number of bits with apostrophes
  - Ex: `($(clog2(NUM_POINTS))'(42)`
- `'0` and `'1` automatically extends all 0's and 1's to width of target

System Tasks

- Any keyword beginning with $ is a system task
- Ex: `$clog2`, `$display`, `$time`, `$finish`
- `$clog2(WIDTH)` gives you enough bits to count to [0,WIDTH-1]
- `$clog2(WIDTH+1)` gives you enough bits to count to [0,WIDTH]

Project 1

Goals

- Read lots of Verilog to try to learn it quickly
- Learn to use a priority selector (will be helpful for later projects!)
- Analyze the costs of different implementations

Submission Script

- You will submit projects to the EECS 470 autograder by uploading your solution files to the `main` branch of your GitHub repository and running the project submission script on CAEN:
- `/usr/caen/misc/class/eecs470/submit project_num`

Lab Assignment

Goals

- Expose you to lots of Verilog
- Mostly reading, not too much writing
- Complete the worksheet and show it to us

How?

- Follow the setup tutorial, this is one of the most important documents in this class...
- Assignment on the course website.
- Submission: Place yourself on the help queue and we will check you off when you feel comfortable you can demonstrate what is required

Useful Links

- Assignment on the course website.
- Consider using VS Code with Remote SSH via Scott Smith's helpful guide
- Get comfortable using CAEN VNC if you can
- Read the Screen tutorial before synthesizing your projects.
- Review the GTKwave Waveform Viewer tutorial should VNC be too delayed
- Coming soon... helpful SystemVerilog tips
- Submission: Place yourself on the help queue and we will check you off.

Appendix

Note

Everything from here to the end of the presentation is from an older version of this presentation, but kept here for reference.

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)

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

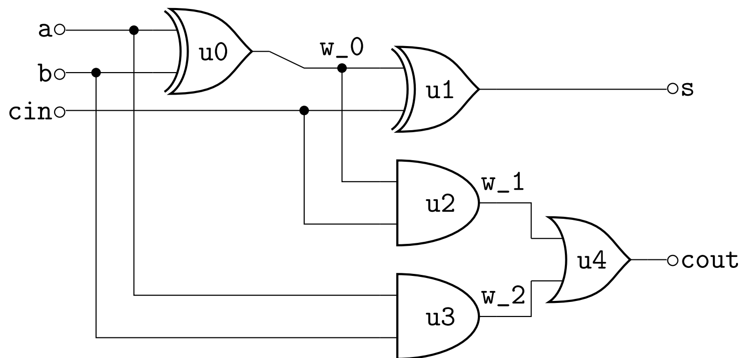


Figure: 1-bit Full Adder

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
```

Verilog Semantics

Lexical

- Everything is case sensitive.
- Type instances must start with `A-Z,a-z,_`. They may contain `A-Z,a-z,0-9,_,$`.
- Comments begin with // or are enclosed with /* and */.

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

<table>
<tr><th colspan="2">Arithmetic</th></tr>
<tr><td>*</td><td>Multiplication</td></tr>
<tr><td>/</td><td>Division</td></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Subtraction</td></tr>
<tr><td>%</td><td>Modulus</td></tr>
<tr><td>**</td><td>Exponentiation</td></tr>
<tr><th colspan="2">Bitwise</th></tr>
<tr><td>~</td><td>Complement</td></tr>
<tr><td>&</td><td>And</td></tr>
<tr><td>|</td><td>Or</td></tr>
<tr><td>~|</td><td>Nor</td></tr>
<tr><td>^</td><td>Xor</td></tr>
<tr><td>~^</td><td>Xnor</td></tr>
<tr><th colspan="2">Logical</th></tr>
<tr><td>!</td><td>Complement</td></tr>
<tr><td>&&</td><td>And</td></tr>
<tr><td>||</td><td>Or</td></tr>
</table>

<table>
<tr><th colspan="2">Shift</th></tr>
<tr><td>>></td><td>Logical right shift</td></tr>
<tr><td><<</td><td>Logical left shift</td></tr>
<tr><td>>>></td><td>Arithmetic right shift</td></tr>
<tr><td><<<</td><td>Arithmetic left shift</td></tr>
<tr><th colspan="2">Relational</th></tr>
<tr><td>></td><td>Greater than</td></tr>
<tr><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td><</td><td>Less than</td></tr>
<tr><td><=</td><td>Less than or equal to</td></tr>
<tr><td>!=</td><td>Inequality</td></tr>
<tr><td>!==</td><td>4-state inequality</td></tr>
<tr><td>==</td><td>Equality</td></tr>
<tr><td>===</td><td>4-state equality</td></tr>
<tr><th colspan="2">Special</th></tr>
<tr><td>{,}</td><td>Concatenation</td></tr>
<tr><td>{n{m}}</td><td>Replication</td></tr>
<tr><td>?:</td><td>Ternary</td></tr>
</table>

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
```

Always Block Examples

Combinational Block

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

Sequential Block

```
always_ff @(posedge clock)
begin
    x <= next_x;
    y <= next_y;
end
```

Blocking vs. Nonblocking Assignment by Example

Blocking Example

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

- Behave exactly as expected
- Standard combinational logic

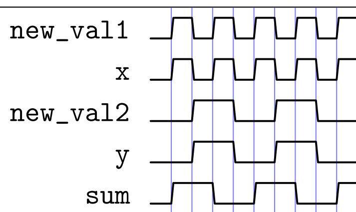


Figure: Timing diagram for the above example.

Blocking vs. Nonblocking Assignment by Example

Nonblocking Example

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

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

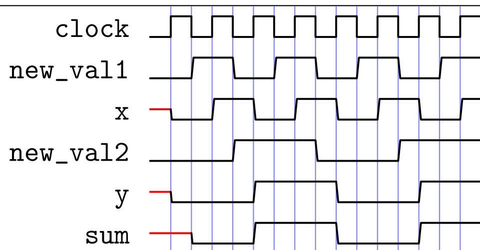


Figure: Timing diagram for the above example.

Blocking vs. Nonblocking Assignment by Example

Bad Example

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

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

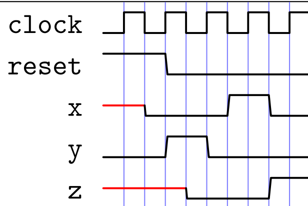


Figure: Timing diagram for the above example.

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

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

Wiring 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;`

Instantiating Modules

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

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;
```

Flow Control by Example

Casez Example

```
always_comb
begin
    casez(alu_op)
        3'b000: r = a + b;
        3'b001: r = a - b;
        3'b010: r = a * b;
        ...
        3'b1??: r = a ^ b;
    endcase
end
```

Unsythesizable 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

Unsythesizable 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
```

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)

Tasks and Functions by 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
```

Tasks and Functions by Example

function Example

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

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

Intro to System Tasks and Functions

- Just like regular tasks and functions
- But they introspect the simulation
- Mostly these are used to print information
- Behave just like `printf` from C

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.

More available at ASIC World.

Test Bench Tips

Remember to...

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

Macros And Parameters

Macros

- Replaced by their values
  - Used with the backtick (`)

Parameters

- Can configure a module at instantiation
- Localparameters are local to a module
- Like constants in C

Example

```
`define NUM_INPUTS 10        // Macro
module example
  #(parameter WIDTH = 30) // Parameter with default value
  (input        [`NUM_INPUTS-1:0][WIDTH-1:0] a,
   output logic [`NUM_INPUTS-1:0]            out)
   localparam WIDTH_LOG = $clog2(WIDTH); // Localparam
   ...
   logic [`WIDTH-1:0][WIDTH_LOG-1:0] b;
   ...
endmodule
```