

# *Verilator 101*

---

*Presenter: Hai Cao Xuan*

*Computer Architecture 203B3*



# objectives

- Understand how to use Verilator to compile SystemVerilog source codes
- Understand how to write C++ testbench to use with Verilator
- Understand how to write Makefile to manage projects
- **Make a simple 8-bit ALU**
- **Make a button buffer**

# Introduction

---

# What is Verilator?

Verilator is a tool to compile Verilog and SystemVerilog source codes to optimized C++ or SystemC code.

→ Verification or Modeling



# Why bother using Verilator?

## **Speed**

Cycle-based → Extremely fast, but only be used for synchronous circuits.

## **Code quality**

Not fully support SystemVerilog **but not accept most non-synthesizable** code → have to write **better code**

## **Price**

Free and open-source

# Why bother using Verilator?

**Question** Is it enough for us?

**Answer** Yes :)

**Question** What's wrong with Quartus, ModelSim, etc.?

**Answer** It's slow and limit your imagination.

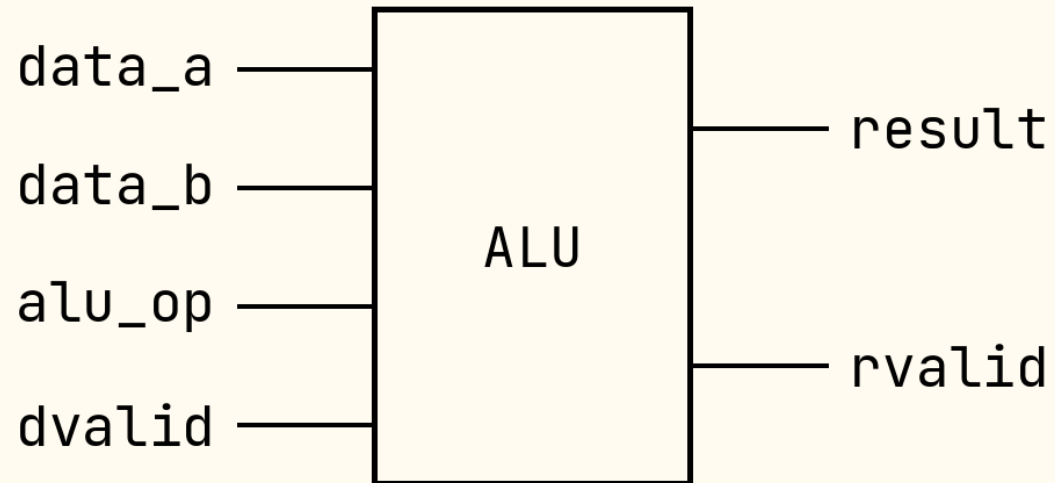
**Question** Can I use Quartus, ModelSim, Vivado, etc. instead in this course?

**Answer** Yes, you can. But why fleeing from something just because the fun is not yet to come?

# Let's build an ALU

---

# Specification





# Specification

## Input

<b>data_a</b>	8-bit data
<b>data_b</b>	8-bit data
<b>alu_op</b>	2-bit for ALU operation
<b>dvalid</b>	1-bit data valid: if this signal is 0, the data are invalid, there is no result.

## Output

<b>result</b>	8-bit data
<b>rvalid</b>	1-bit result valid: this signal is 1 if the result is valid.

# Specification

Input

`alu_op`      4 operators: NOP, ADD, OR, AND

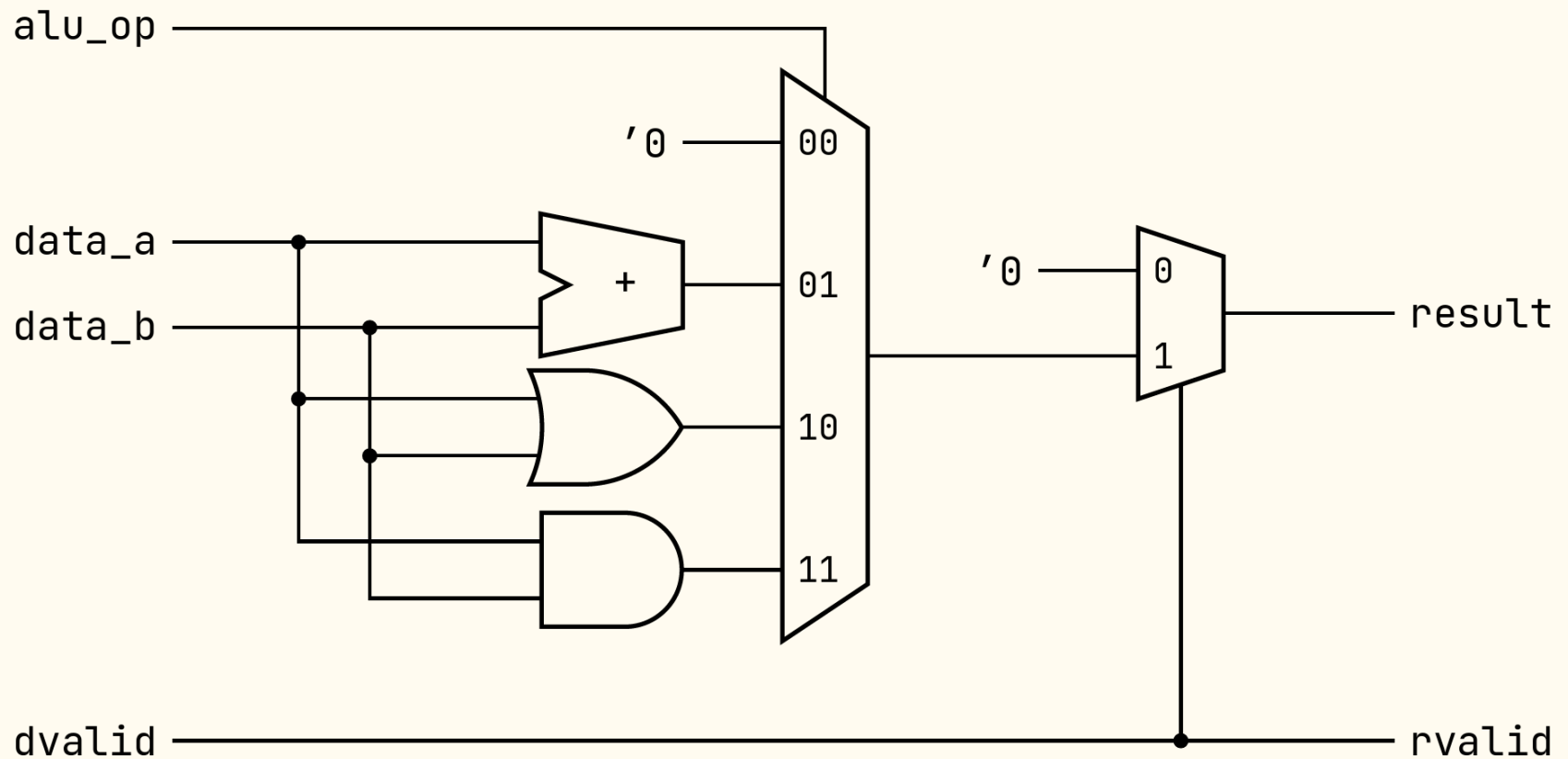
<code>valid(i)</code>	<code>alu_op</code>	<code>result</code>	<code>valid(out)</code>
0	xx	'0	0
1	00 NOP	'0	1
1	01 ADD	<code>data_a + data_b</code>	1
1	10 OR	<code>data_a   data_b</code>	1
1	11 AND	<code>data_a &amp; data_b</code>	1

# Analyze the Specification

- a) How many input, output signals?
- b) The relationship between them?
- c) Possible modules in the design?
- d) Possible algorithms to implement the design?

→ Sketch the diagram

# Sketch the Diagram



## ◇ *Remember the Guideline*

→ *Suffixes*

Input

**data\_a\_i**

**data\_b\_i**

**alu\_op\_i**

**dvalid\_i**

Output

**result\_o**

**rvalid\_o**

# Time to Code

Actually, you could open the folder to see the sample code.

ex01/

# Structure

```
~/r/s/02-verilator/ex01 $ tree
```

```
.
├── include
│   └── my_pkg.svh
├── makefile
├── src
│   └── alu.sv
├── tb_top.cpp
├── top.sv
└── top.js
```

```
2 directories, 6 files
```

# Structure

1. `include/:` contains package files
2. `*.svh:` SystemVerilog package files
3. `makefile:` makefile
4. `src/:` contains RTL source code
5. `*.sv:` SystemVerilog files
6. `top.sv:` Top for test
7. `top.js:` Yosys file



# **It's Verilator's showtime**

---

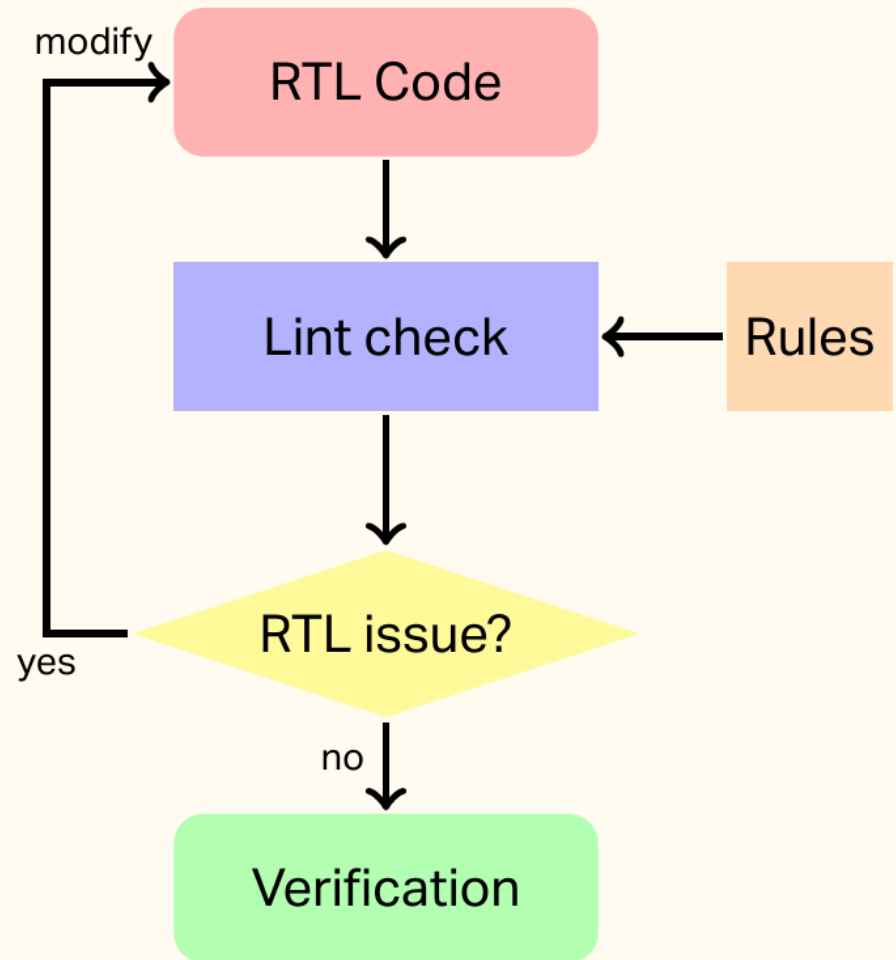
Lint

# Lint

A process of static code analysis checks on RTL design to find violations based on sets of guidelines or rules.

Rules:

- Verilator
- Spyglass (VCS)
- ...



# Lint Basic Goals

1. Basic connectivity issues (floating inputs, width mismatch...)
2. Simulation issues
  - a. Incomplete sensitivity list
  - b. Incorrect use of blocking/non-blocking assignments
  - c. Potential functional errors
  - d. Possible simulation hang cases and race cases
3. Structural issues that affect the post implementation functionality or performance
  - a. Multiple drivers
  - b. High fan-in mux
  - c. Synchronous/asynchronous use of resets
4. Unsynthesizable constructs, RTL vs. gate simulation mismatch

# Verilator Lint

```
$ verilator -Wall -sv --lint-only {files} --top-module {top module}
```

```
$ verilator -Wall -sv --lint-only include/my_pkg.svh src/alu.sv top.sv --  
top-module top
```

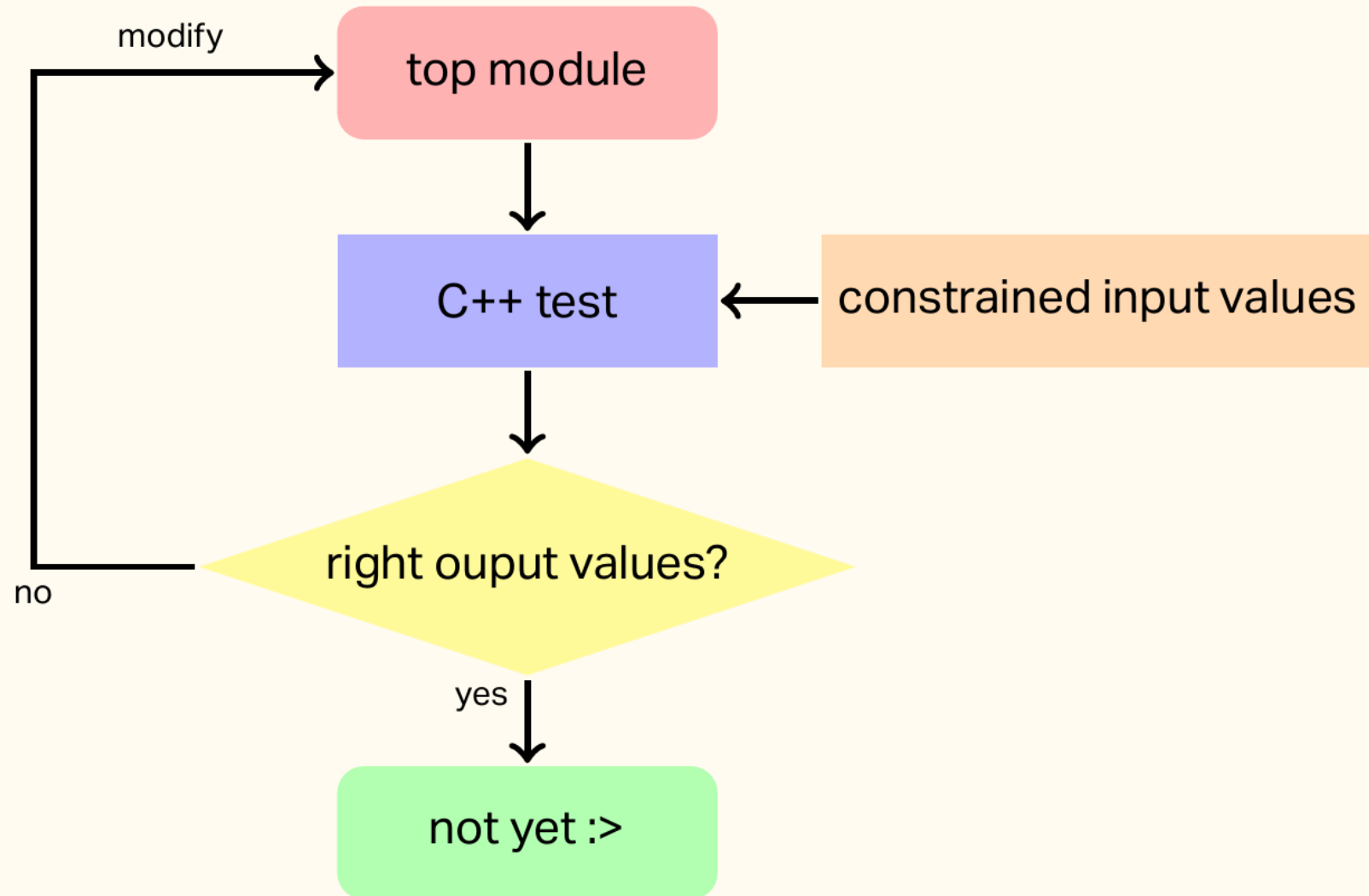
\* You must put reused modules/files before the modules/files using them.

When warnings or errors occur, fix them or go to this link to figure out how to fix them.

<https://verilator.org/guide/latest/warnings.html>

Verification

# Time-based Verification



# Why using C++?

1. Easy to code
2. Utilize its huge libraries



# Time-based Verification – basic

## Basic Procedure

1. Set initial values
2. Set series of test values
3. Monitor output values: **compare**

# Time-based Verification – basic

It's playtime

ex01/

# Time-based Verification – basic

## 1. Verilate the top module

```
$ verilator -Wall -sv -cc {files} --top-module {top-module} --exe {test-files}
```

```
$ verilator -Wall -sv -cc include/my_pkg.svh src/alu.sv top.sv --top-module top --exe tb_top.cpp
```

## 2. Build the top module

```
$ make -C obj_dir -f V{top-module}.mk V{top-module}
```

```
$ make -C obj_dir -f Vtop.mk Vtop
```

## 3. Simulate

```
$ ./obj_dir/V{top-module}
```

```
$ ./obj_dir/Vtop
```

# Why are there thousands of commands?

## How could I remember all?

That's why makefile is created, to ease the tedious of typing.

Depend on your projects, modify the variables: TOPMODULE and FILES

To know how to run, type: **make help**

# Time-based Verification – a little bit advanced

## Random-Value Procedure

1. Set initial values
2. Set test values: **RANDOMLY**
3. Get expected output values
4. Monitor output values: **compare**
5. Repeat step 2. N times

→ Random values help check the correctness of the design by driving it in some cases which designers might not or even cannot think or imagine

# Time-based Verification – a little bit advanced

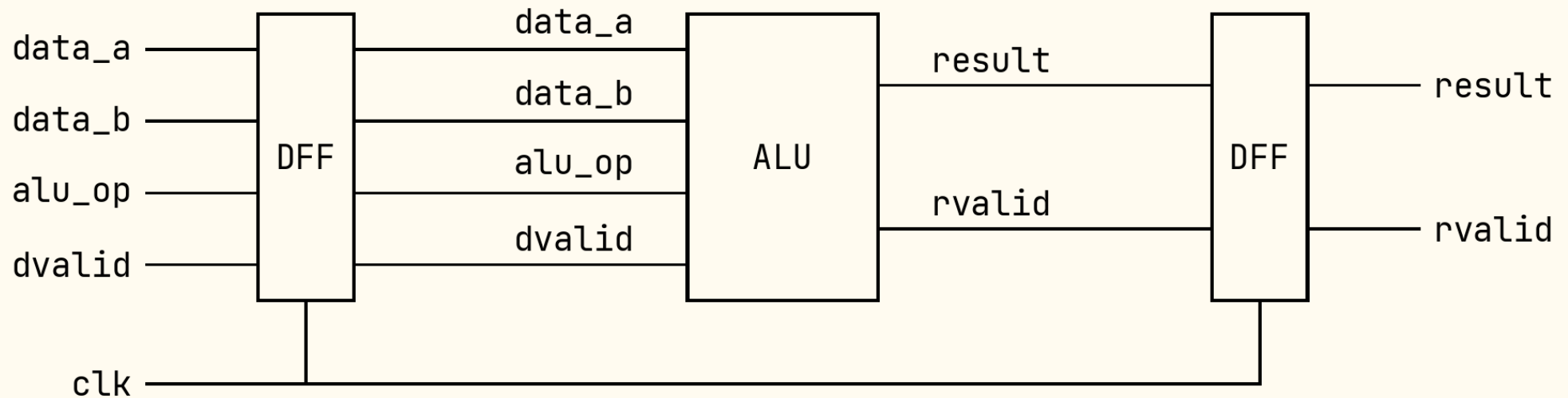
Another one

ex02/

Pay attention to how the code was rearranged to make it simpler and easier to maintain and modify

It's CLK realm

# Time-based Verification – a little bit advanced





# Time-based Verification – a little bit advanced

## Recommended Procedure for Clock and Reset in Verilator

1. Set initial values
2. Model 1 clock cycle → Monitor output values: **compare**
3. Set test values: **RANDOMLY**
4. \*Extra processes
5. Get expected output values
6. Model reset
7. Repeat step 2. N times

Where's Waveform?

# Time-based Verification – a little bit advanced

ex03/

Pay attention to how the code was rearranged to make it simpler and easier to maintain and modify

# Time-based Verification – a little bit advanced

## Verilate the top module

```
$ verilator -Wall -sv -cc --trace-fst {files} --top-module {top-module} --exe {test-files}
```

```
$ verilator -Wall -sv -cc --trace-fst include/my_pkg.svh  
src/alu.sv top.sv --top-module top --exe tb_top.cpp
```

In the test file, take a closer look to some extra lines of generating waveforms or trace.  
After simulating, a \*.vcd file will be in the current folder, use this command to view:

```
$ gtkwave {vcd file}
```

```
$ gtkwave top.vcd
```

**It's your showtime**

---

# Problem

The FPGA in which you implement a design often has high frequency, so when a button is pressed, which you intend it to press in 1 cycle, the design will consider the signal is active in multiple cycles.

→ What is your solution?

# Problem

Step 1. Analyze the problem

Step 2. Write the specification

Step 3. Sketch the waveform

Step 4. Draw the FSM

Step 5. Design

# Analyze the Problem

What is your solution?



# Write and Analyze the Specification

- a) How many input, output signals?
- b) The relationship between them?
- c) Possible modules in the design?
- d) Possible algorithms to implement the design?

# Specification

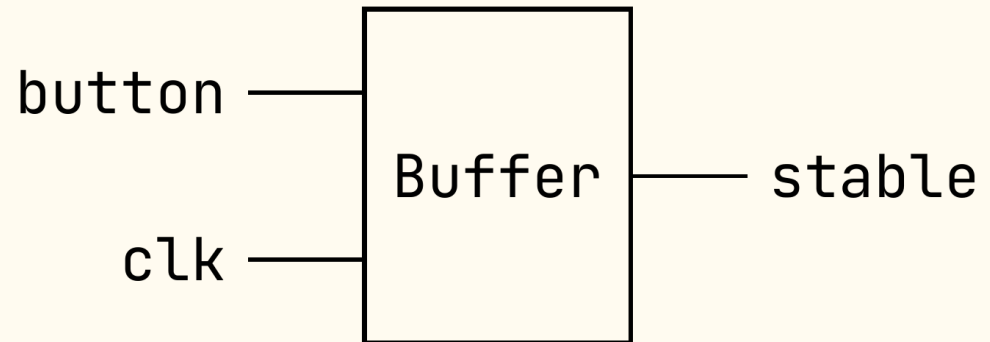
Input

**clk**      ...

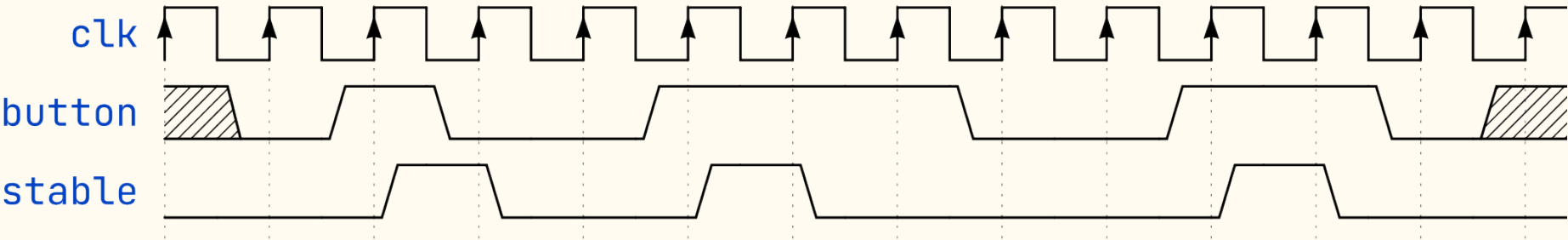
**button**    ...

Output

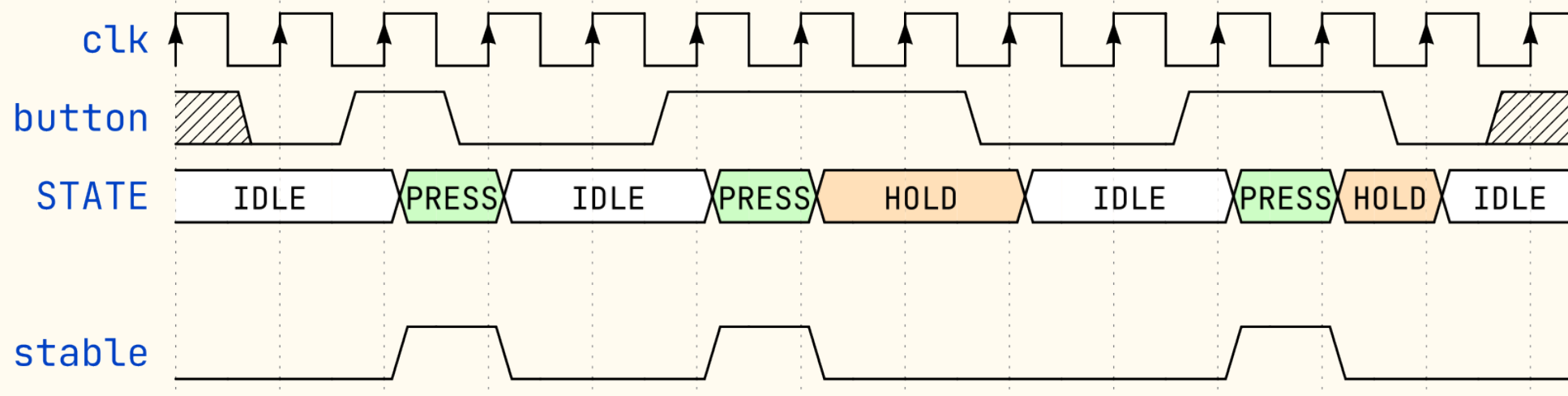
**stable**    ...



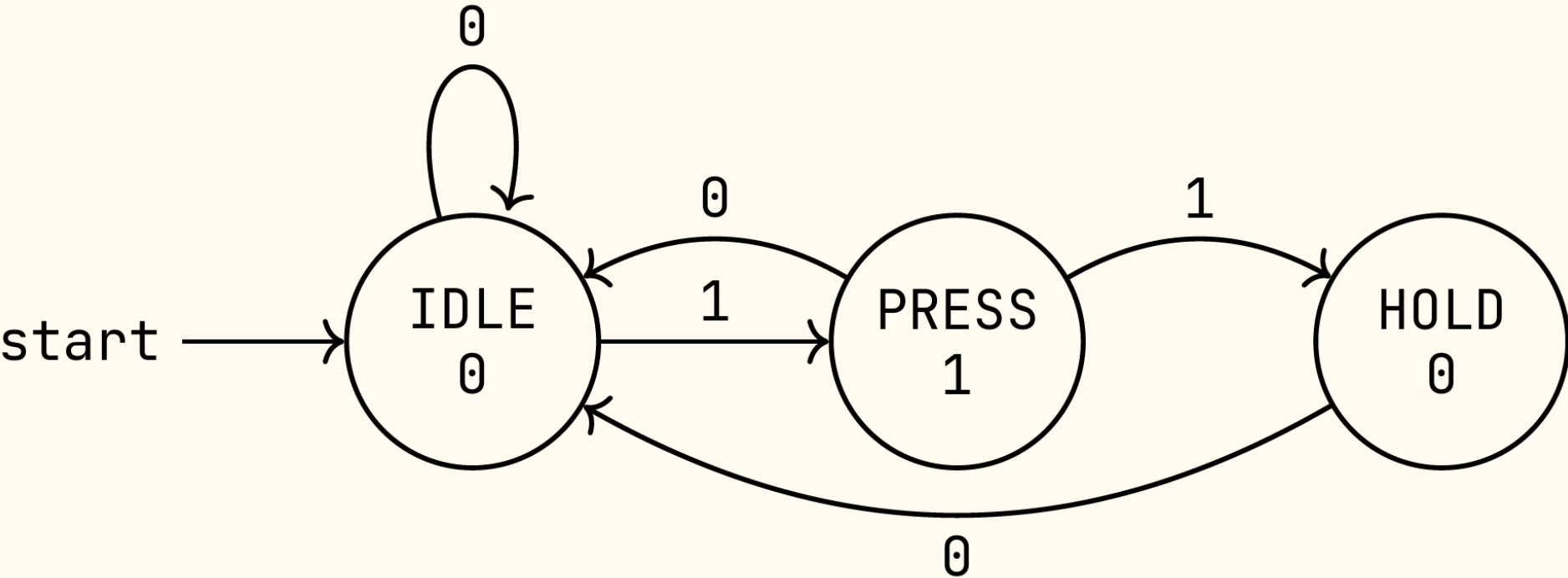
# Waveform



# Waveform



# FSM



Let's get your hands  
dirty...

# Questions?

---

# References

1. <https://www.veripool.org/verilator/>
2. <https://verilator.org/guide/latest/>
3. <https://zipcpu.com/blog/2017/06/21/looking-at-verilator.html>
4. [https://www.itseembedded.com/dhd/verilator\\_1/](https://www.itseembedded.com/dhd/verilator_1/)