Gisselquist Technology, LLC

7. Data Coherency

Daniel E. Gisselquist, Ph.D.

Lesson Overview

Understanding why the button counter didn't work as expected

- It double counted button presses
- Sometimes it counted 2-4 times per button press
- Rarer observed effects
  - At one point, the counter counted *down*
  - Another time, it skipped 11 numbers at once

Objectives

- Understand data coherency issues
- Understanding bouncing
- Build and verify a button debouncer

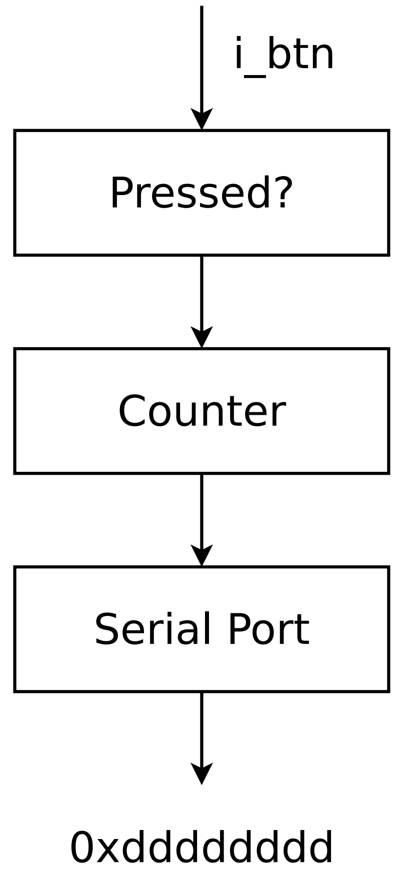
Last Lesson

This lesson picks up where the last lesson left off.

- If you didn't build the button counter, or implement it in hardware
  - You missed a valuable lesson
  - Go back and try it
  - Press the button several times, see what happens
- If it didn't work like you expected it should
  - Feel free to start this lesson

Button Press Counter

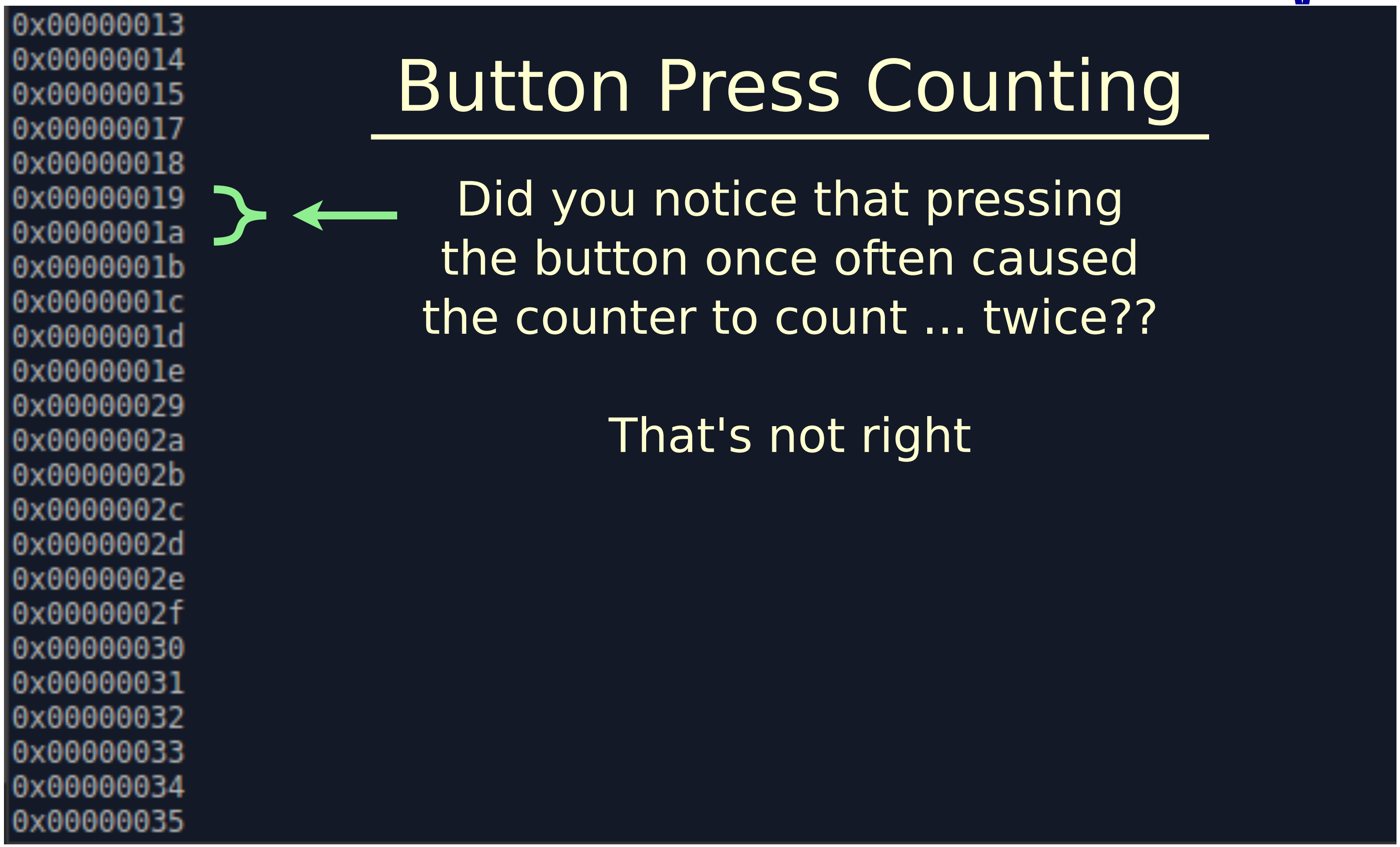
We built a button press counter in the last lesson



1. It detected a button press,
2. Incremented a counter,
3. Sent the value over the serial port as hexadecimal, and was
4. Witnessed at a terminal

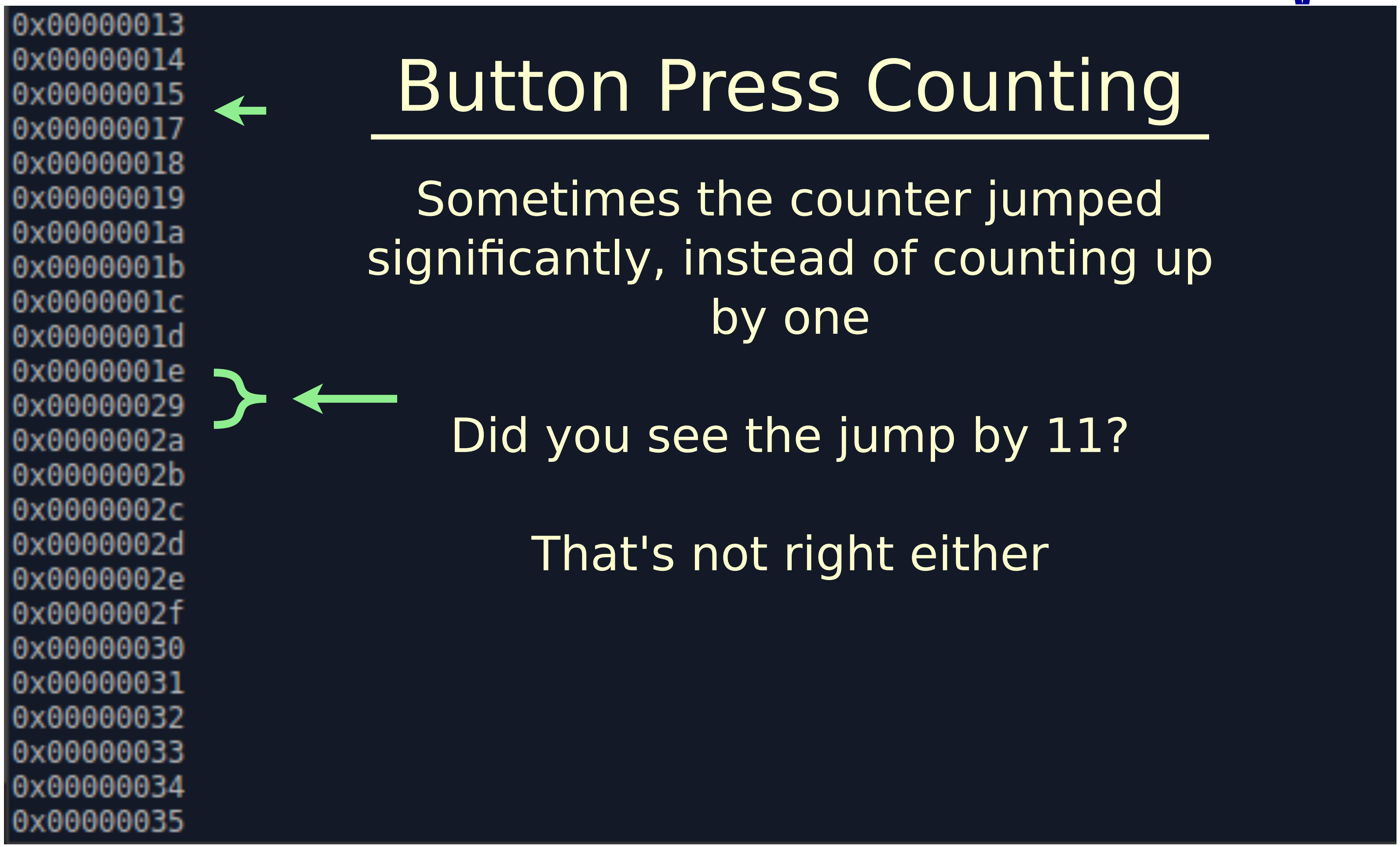
An easy way to count button presses, no?

Was this what you expected?



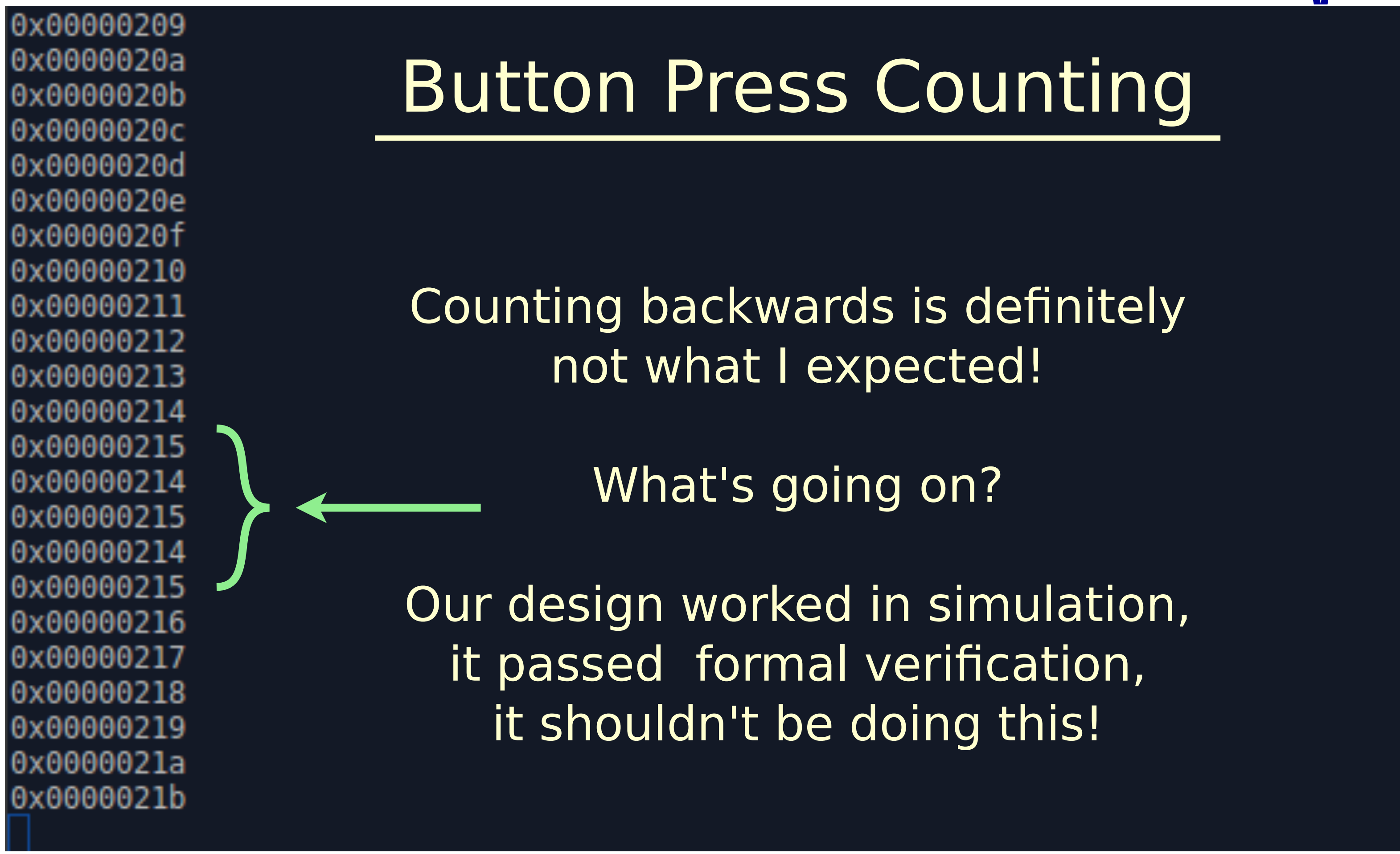
This looks like it could be fixed

Was this what you expected?



This might take some work to understand

Was this what you expected?



Now I'm really confused! What happened?

Logic takes time

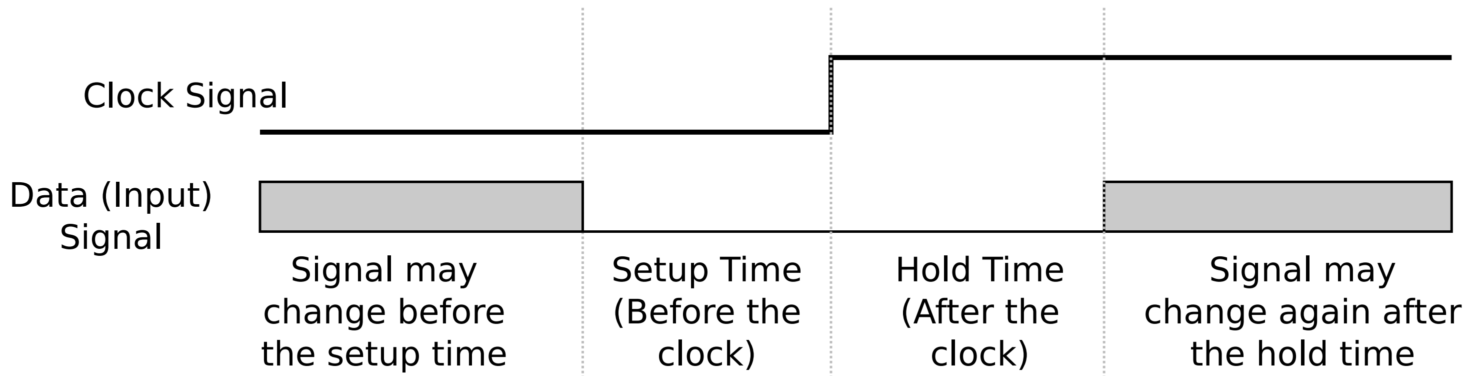
To understand what happened, you need to understand that . . .

- Logic takes time
- It takes time to go through a logic gate
- It takes time to move about the chip

All this work must be done in time for the next clock

Setup and Hold

Flip-Flops (FFs) (a.k.a. registers or **regs**) have two requirements

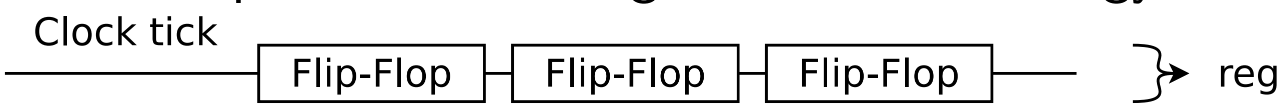


1. The incoming data must be constant for a *setup period* of time before the clock edge
2. It must also be constant for a *hold* time after the clock edge

If these criteria are not met, your design will not function as you expect

Caving Analogy

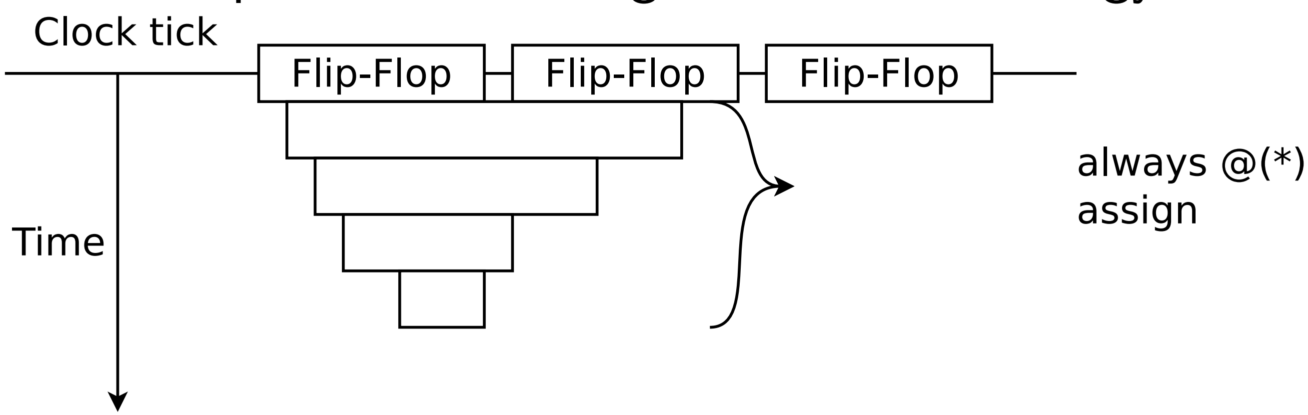
I like to explain clocks using caves as an analogy



It starts with the clock, and the FFs set using that clock

Caving Analogy

I like to explain clocks using caves as an analogy

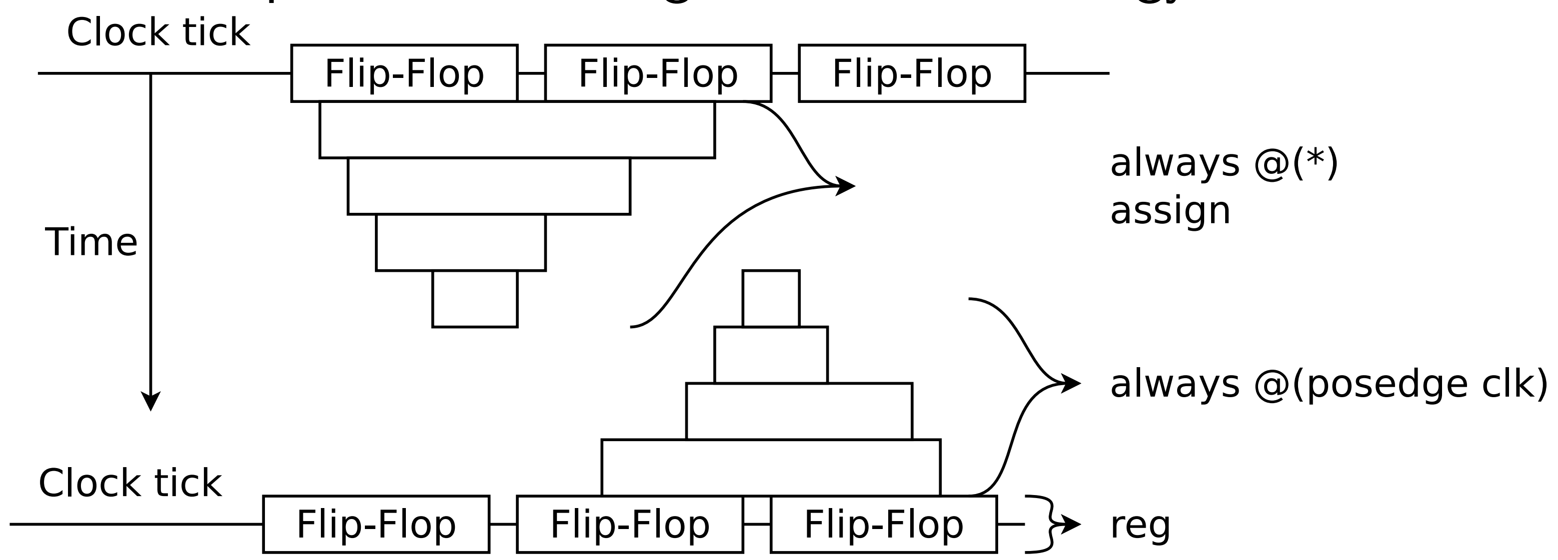


Adding logic creates stalagtites

- Stalagtites in this analogy are formed from **assign** statements and **always** `@(*)` blocks
  - Their timing is derived from the last clock tick

Caving Analogy

I like to explain clocks using caves as an analogy

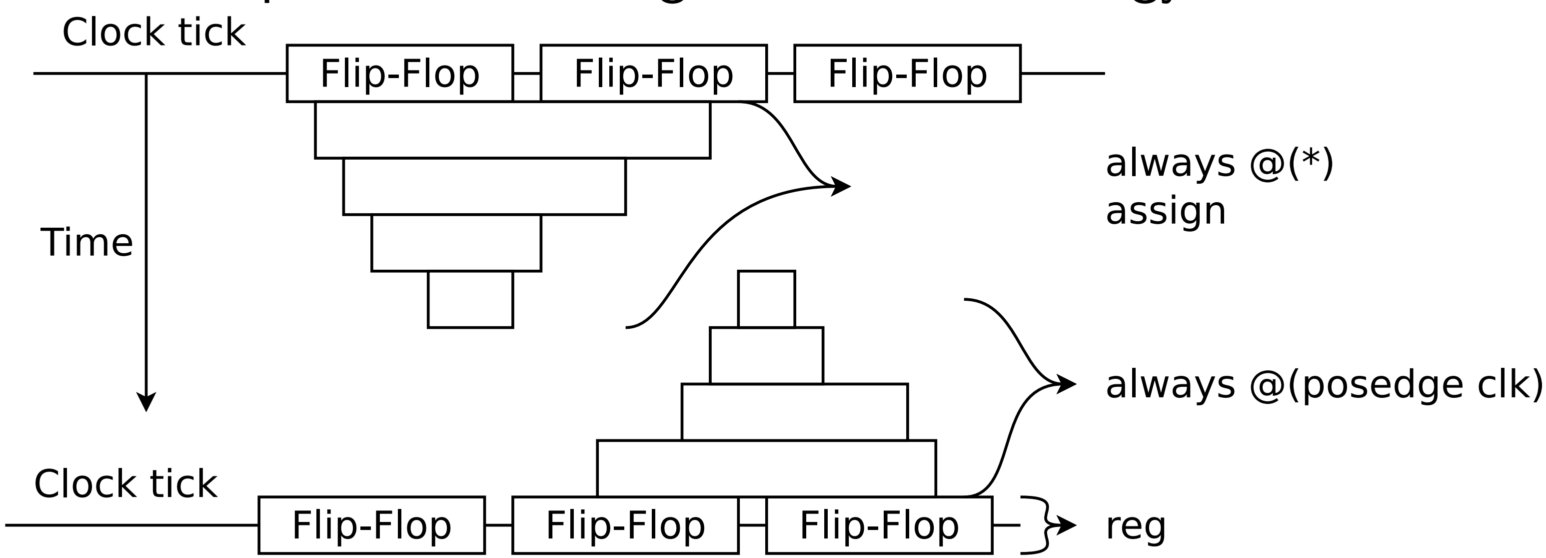


Adding logic creates stalagtites and stalagmites

- Stalagtites in this analogy are formed from **assign** statements and **always** `@(*)` blocks
- Stalagmites are formed from **always** `@(`**posedge** `i_clk)` blocks
  - Their timing is derived from the next clock tick

Caving Analogy

I like to explain clocks using caves as an analogy



Your goal as the designer is to make certain that there's extra space between stalagtites and the stalagmites

- This is your margin
- You need this margin for success

Did we guarantee any margin in our button press design?

What happened

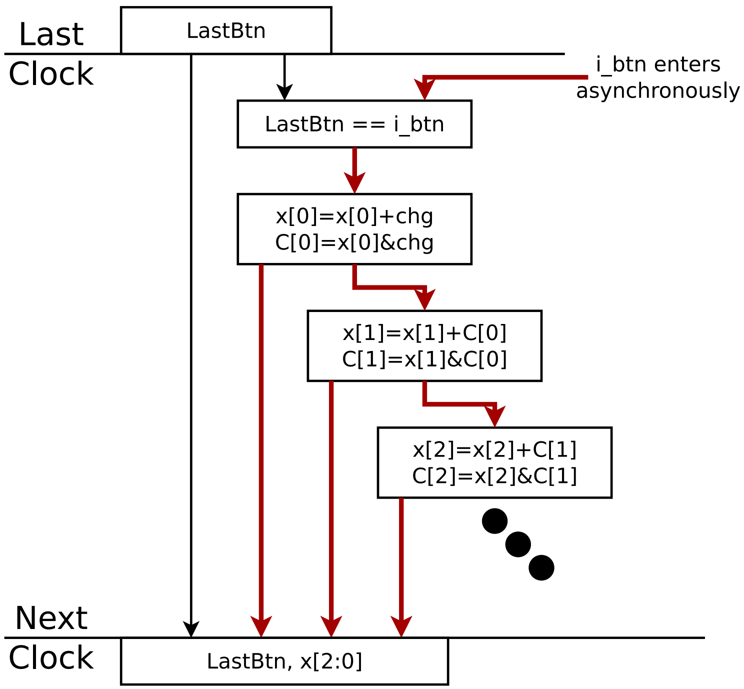
For reference, here was the basic problematic code:

```
initial o_count = 0;
always @(posedge i_clk)
if (i_reset)
        o_count <= 0;
else if ((i_btn)&&(!last_btn))
        o_count <= o_count + 1'b1;
```

See the problem?

No margin

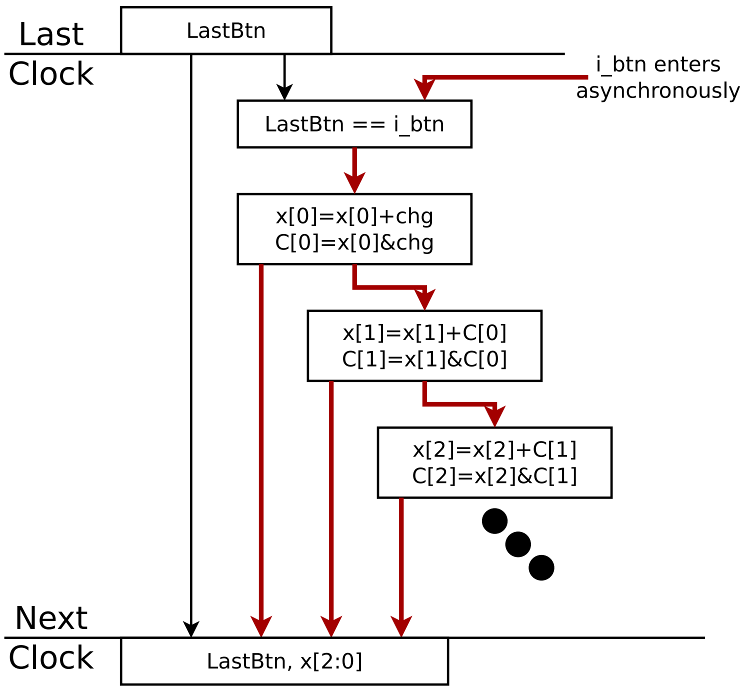
In our last design, . . .



- Timing analysis was based upon the time between FFs
- The 32-bit carry chain stretched out the logic
- The high clock rate I used just made this worse

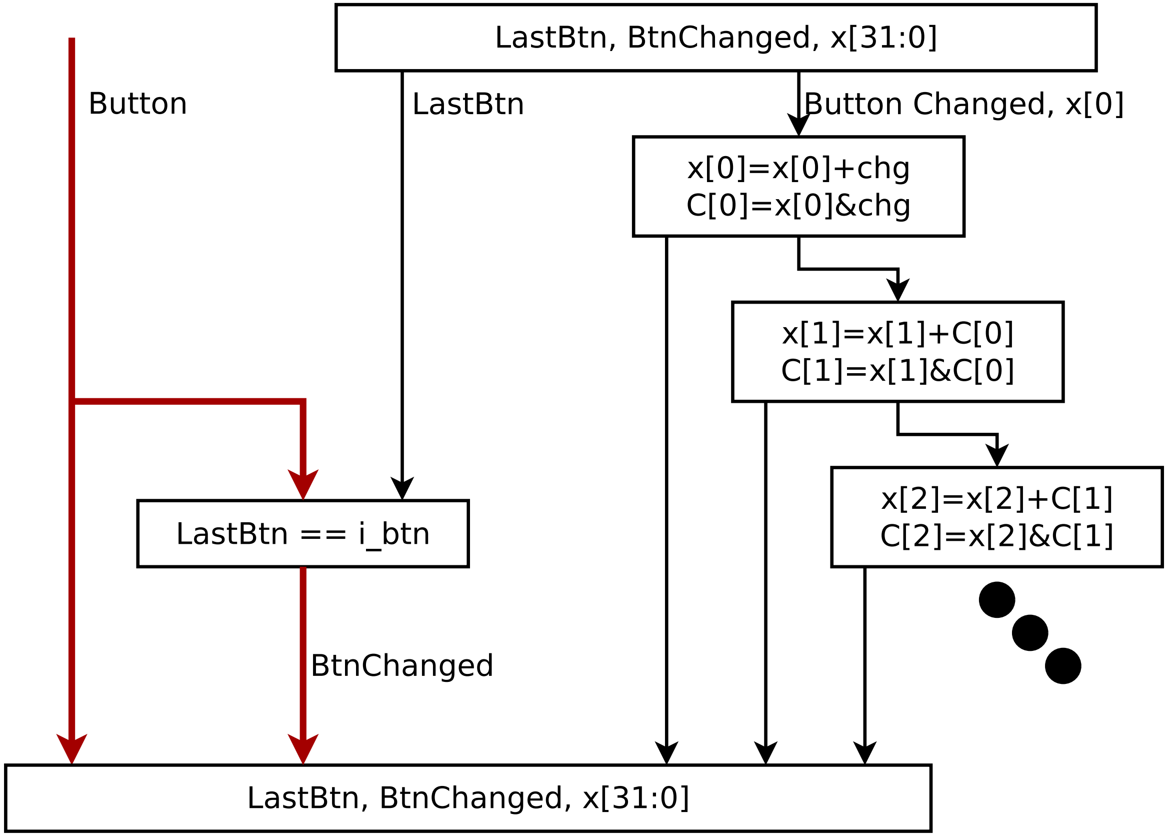
No margin

In our last design, . . .



We did *nothing* to guarantee the button press plus our logic would fit between two clock ticks with margin left over

More margin

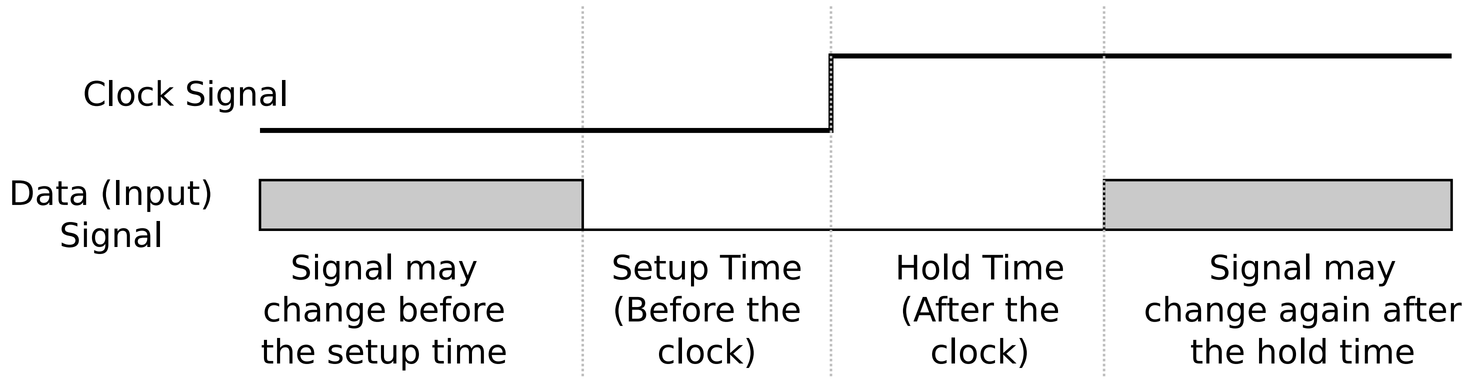


Eliminating almost all of the logic is better

- But still not good enough
- The button input must go directly into an FF

Asynchronous Input

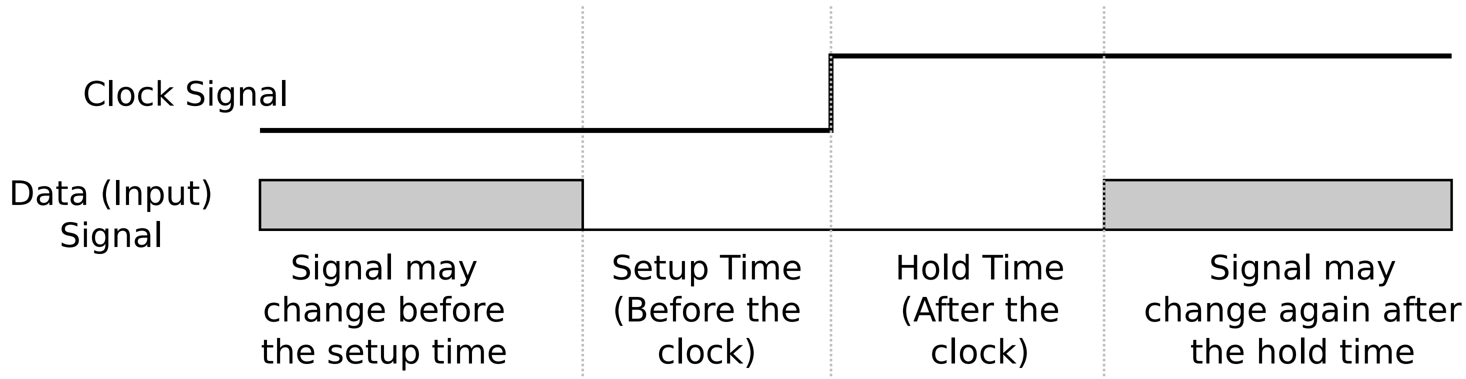
If we can't control when the button rises, . . .



How can we ensure the setup and hold times are met?

Asynchronous Input

If we can't control when the button rises, . . .

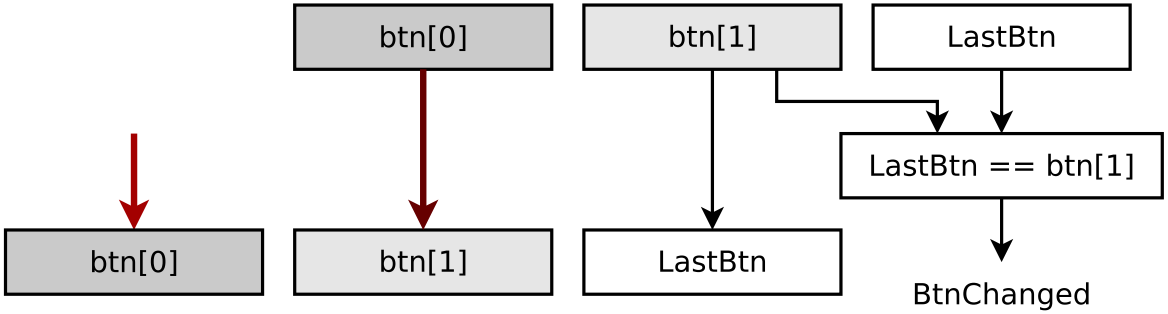


How can we ensure the setup and hold times are met?

- We can't

Asynchronous Input

Rule: All asynchronous inputs must go through a 2FF synchronizer

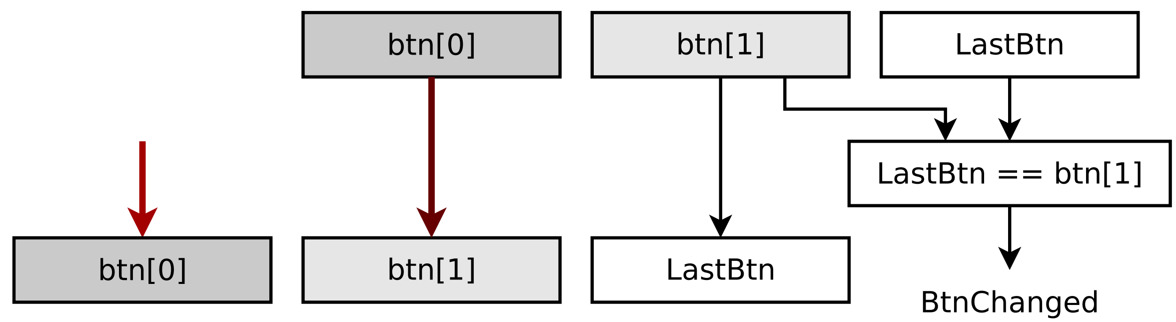


- Inputs must first go directly into a FF
  - No other logic is allowed
  - The output of this FF *may not (yet) be stable*

    Metastability is the name for when a logic value is neither zero or one. It is a rare result of not meeting setup and hold requirements

Asynchronous Input

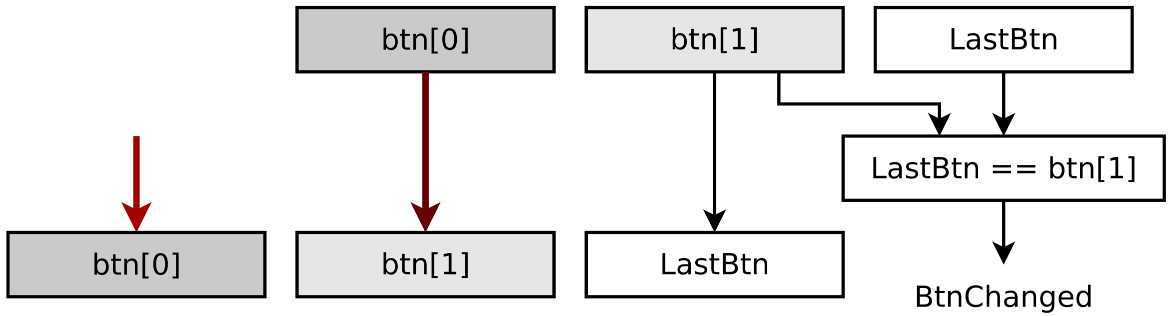
Rule: All asynchronous inputs must go through a 2FF synchronizer



- Inputs must first go directly into a FF
- To deal with the broken setup and hold times, we go directly into a *second flip-flop*
  - This reduces the likelihood of *metastability*

Asynchronous Input

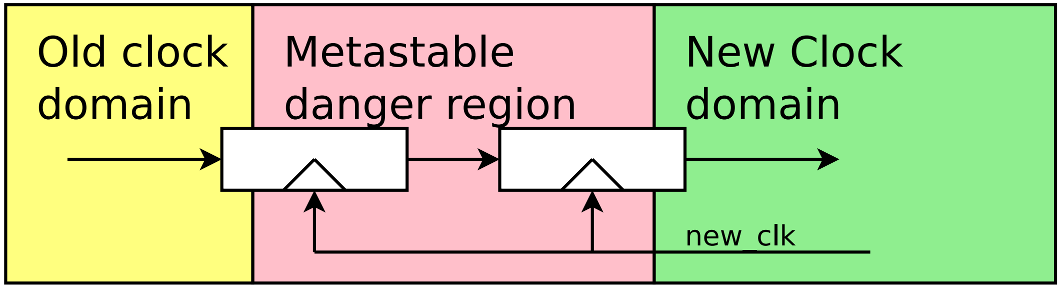
Rule: All asynchronous inputs must go through a 2FF synchronizer



Does this apply to other asynchronous inputs besides buttons?

- *Yes!* If it is not synchronized to your clock, it *must* go through a two flip-flop synchronizer
- Won't this slow signals down? Yes, it will.
  - This is why it is important to provide a clock together with any data signal(s) in low-latency applications

This is a 2 Flip-Flop (2FF) synchronizer



Synchronizing our button input would look like

```
reg      r_btn, r_aux;

initial { r_btn, r_aux } = 2'b00;
always @(posedge i_clk)
        { r_btn, r_aux } <= { r_aux, i_btn };
```

Bouncing

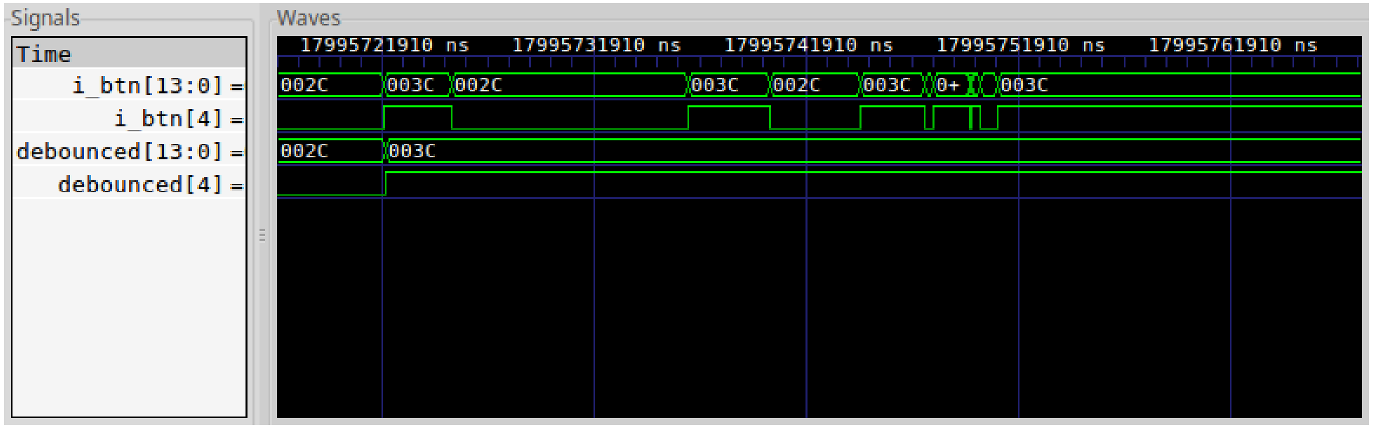
This will fix everything but the double-counts

- Often, pressing a button caused the counter to count *twice*
- The counter wouldn't skip, but one button press generated two counts

This is due to button *bouncing*

Bouncing

A trace from within our design might look like this

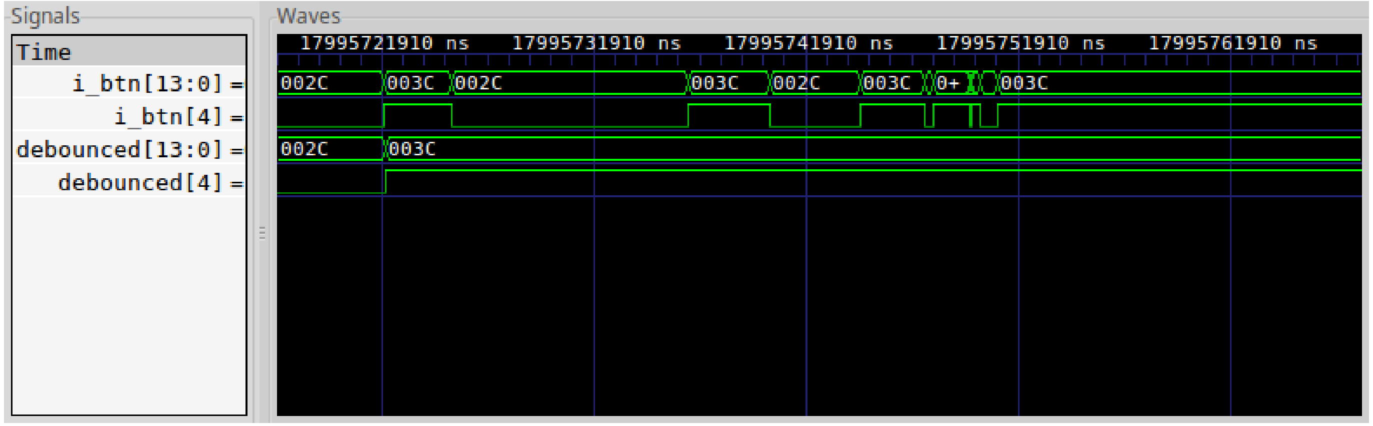


Look at the trace for `i_btn`[4]

- Notice how the button toggles, or “bounces” before it settles
- This is common
- It is caused by
  - Increased capacitance as the contacts come closer
  - A voltage slowly crossing through the threshold region

Bouncing

A trace from within our design might look like this

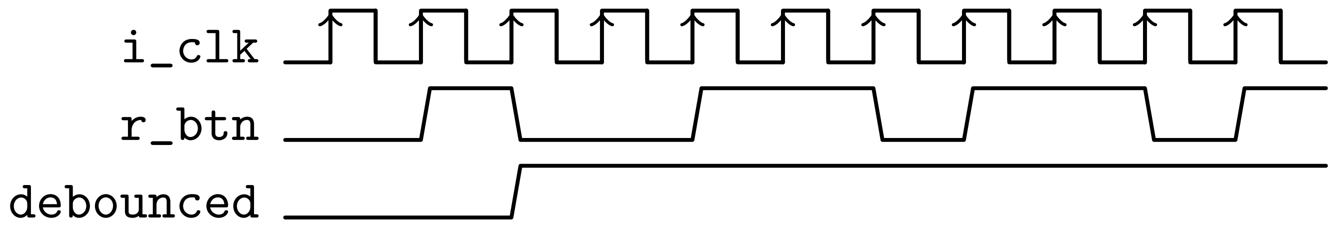


We'll need to simplify this "bouncing" trace

- This is called *debouncing*
- Our goal will be to produce a trace like `debounced[4]` above

Debouncing

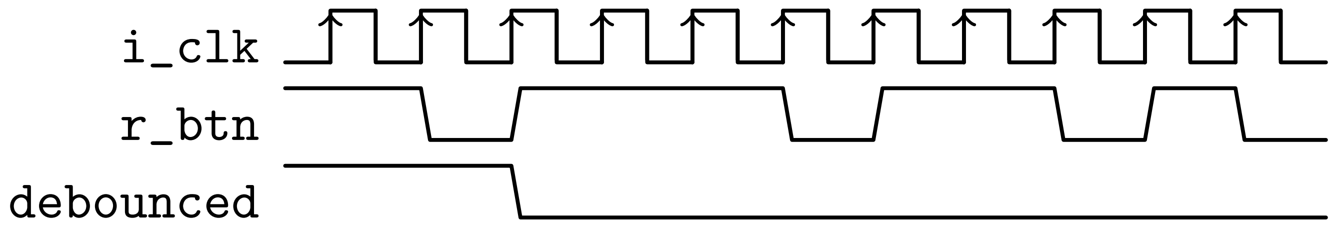
Our goal:



- Create an output that changes when the button changes
- Not when the button bounces

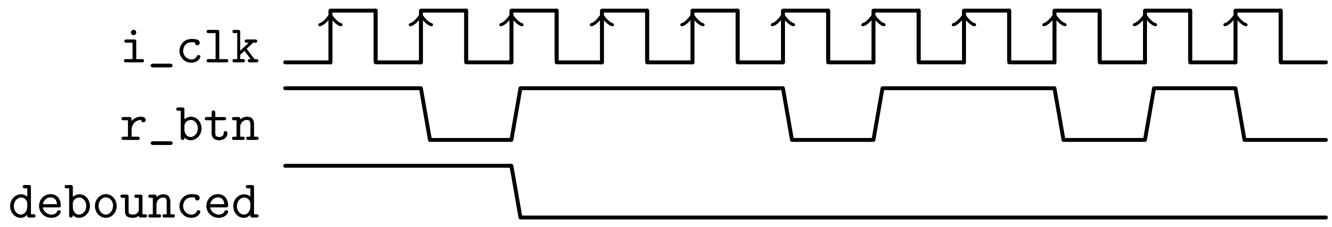
Debouncing

Our goal:



This applies both to the button press as well as to its release

Our goal:

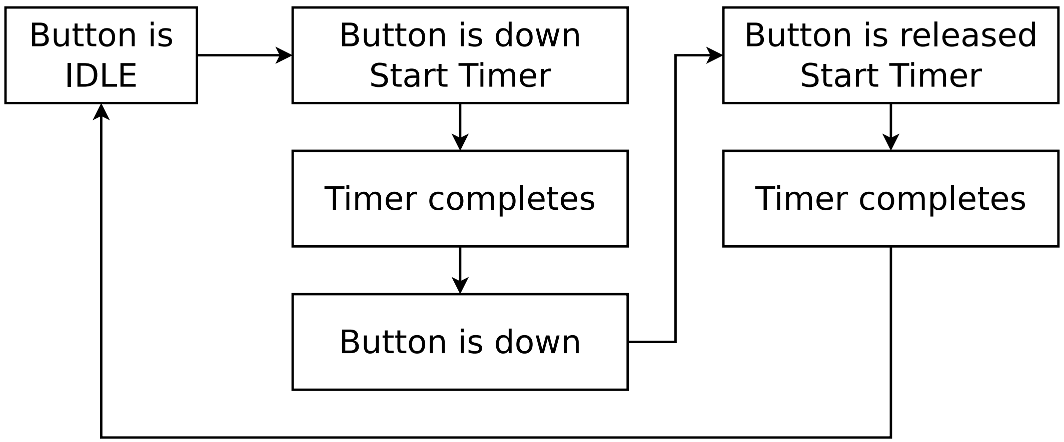


This applies both to the button press as well as to its release

A state diagram might make more sense of what we need to do

Debouncing FSM

Debouncing requires a timer



We'll respond to the button any time the timer is idle

- This should be starting to look familiar

Timer

A button debouncer has three basic parts

1. The 2FF synchronizer

```
initial { r_btn, r_aux } = 0;
always @(posedge i_clk)
        { r_btn, r_aux } <= { r_aux, i_btn };
```

Timer

A button debouncer has three basic parts

1. The 2FF synchronizer
2. The count-down timer

```
initial timer = 0;
always @(posedge i_clk)
if (timer != 0)
        timer <= timer - 1;
else if (r_btn != o_debounced)
        timer <= TIME_PERIOD-1;
```

Timer

A button debouncer has three basic parts

1. The 2FF synchronizer
2. The count-down timer
3. The output

```
always @(posedge i_clk)
if (timer == 0)
        o_debounced <= r_btn;
```

This looks simple enough. Now, how to verify it?

Simulation

The problem is that our *simulated* button never bounced

- If we can simulate a button bouncing, we'll can gain some confidence that our debouncer will work
- Perhaps if we toggled the button input randomly for some period of time, both
  - Following a button press, and
  - Following the button's release
- The simulated button would then stop toggling
  - Remaining in its pressed or released state

Making sure our simulation matches our hardware is an important and critical part of design!

Co-Simulation

A button co-simulator should . . .

- Be able to be pressed

```
class BUTTONSIM {
        // ...
        void press(void);
```

- Be able to be released

```
        void release(void);
```

- Bounce following any press or release

```
        int operator()(void);
}
```

Let's build out these methods

Co-Simulation

Our button class will have two state variables and a constant

```
#define TIME_PERIOD  50000 // 1/2 ms at 10ns
class   BUTTONSIM {
        int     m_state, m_timeout;
public:
        BUTTONSIM(void) {
                // Start with the button up
                m_state = 0; // Not pressed
                //
                // And begin stable, i.e.
                m_timeout=0;
        } // ...
```

- `m_state` is the current state of the button
- `m_timeout` is a count-down timer. When it reaches zero, our button's value will be stable

Sim Press

When a button is pressed, we'll change the state and set a timer

```
class BUTTONSIM {
        // ...
        void    press(void) {
                m_state = 1; // i.e. down
                m_timeout = TIME_PERIOD;
        }
```

The timer will tell us when to stop bouncing

Sim Release

Button release is nearly identical

```
class BUTTONSIM {
        // ...
        void    release(void) {
                m_state = 0; // i.e. released
                m_timeout = TIME_PERIOD;
        }
```

Sim Release

We can also support a test to see if the button is pressed

```
class BUTTONSIM {
        // ...
        bool    pressed(void) {
                return m_state;
        }
```

While this wasn't part of our initial design outline,

- We are going to need this method below

Co-Simulation

Now, let's make our button bounce

```
int     BUTTONSIM::operator()(void) {
        if (m_timeout > 0) // Always count down
                m_timeout--;
        if (m_timeout == TIME_PERIOD-1) {
                // Return any new button
                // state accurately and
                // immediately
                return m_state;
        } else if (m_timeout > 0) {
                // Until we become stable
                // Bounce!
                return rand()&1;
        }
        // Else the button has settled
        return m_state;
}
```

Simulation

Adding this to our simulation requires

- Declaring our button

```
BUTTONSIM *btn;
```

Adding this to our simulation requires

- Declaring our button, and allocating a button object

```
BUTTONSIM      *btn;
// ...
btn = new BUTTONSIM();
```

Simulation

Adding this to our simulation requires

- Declaring our button, and allocating a button object
- Adjusting our button press scheme

```
do {
        int     chv;
         chv = getch();
        if (chv == 'r')
                btn->release();
        else if ((chv != ERR)
                  &&(!btn->pressed())) {
                keypesses ++;
                btn->press();
        }
        // ...
} while(!done);
```

Adding this to our simulation requires

- Declaring our button
- Adjusting our button press scheme
- Adding it to our list of co-sim calls

```
for(int k=0; k<1000; k++) {
        // Advance the Verilator logic
        tb->tick();
        // Serial-port Co-sim
        (*uart)(tb->m_core->o_uart_tx);
        // Button co-sim
        m_core->i_btn = (*btn)();
}
```

Exercise

Your turn!
Build and experiment with the simulation

- Create a trace showing the button bouncing
- Make your Verilog timeout longer than the C++ `TIME_PERIOD`.

Exercise

Now build this on your hardware. Does it work?

- Do you ever get multiple counts for a single press?
- Does the counter ever jump?

Formal Methods

We haven't discussed formal methods this lesson

- Our debouncing circuit can still be verified
  - Although there's not much there
  - You should have an idea of how to do this from our last lessons
- What formal properties might you include to verify this design?

Conclusion

What did we learn this lesson?

- *Always* send asynchronous inputs through a 2FF synchronizer before using them
  - Failing to do this can result in some inexplicable behavior
  - Simulation and implementation might not match
    - Bugs of this kind can be very hard to find and fix
- Buttons *bounce*!
  - A basic debouncing circuit is another FSM
  - This time with a counter within it