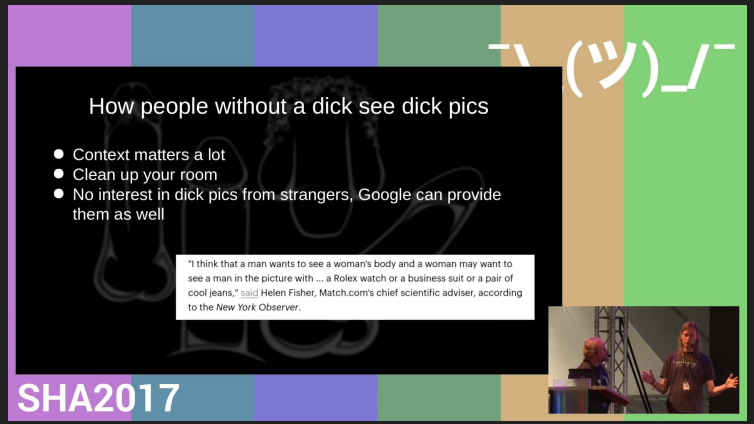
Basic FPGA development

Be able to create a simple FPGA project

https://github.com/ranzbak/fpga-workshop

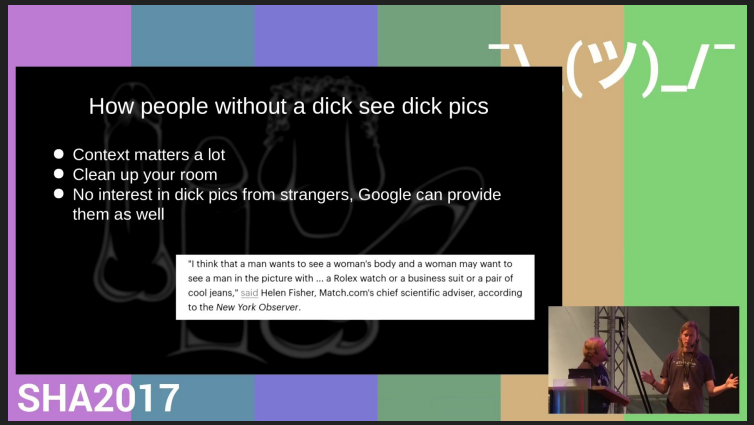
Who Am I

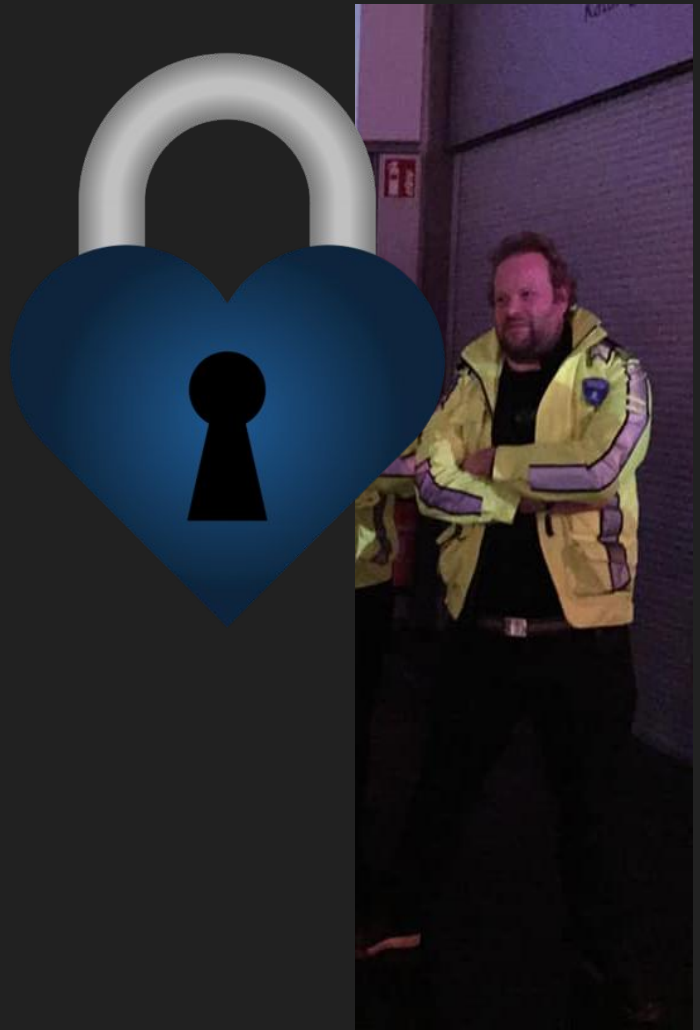
- Paul Honig
- Nerd
- HDL programming hobbyist
- Also does chemical projects ->



Who is my assistent

- Anne Jan Brouwer
- Nerd
- C / C++ opensource developer
- Member of Cyberonderzoeksraad



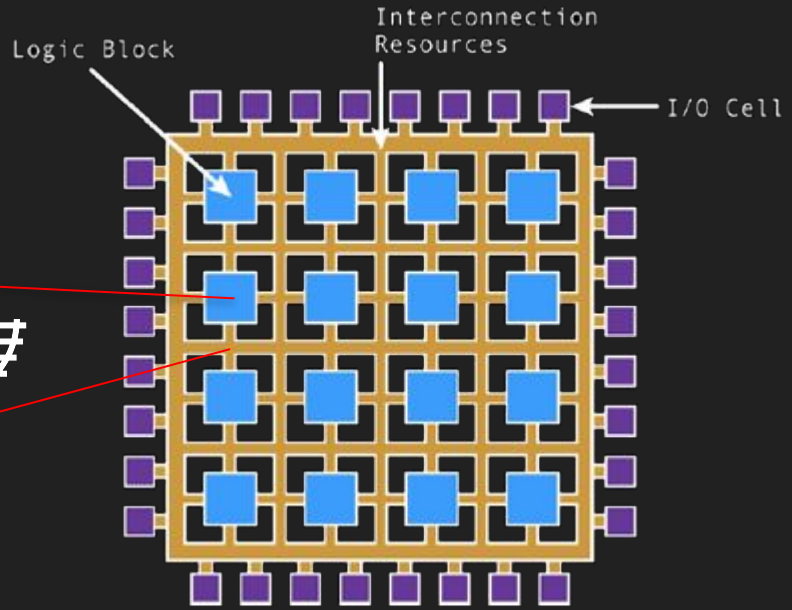


What is an FPGA

- A chip containing a bunch of gates that you can wire together (almost at will).
- A fun way to play with designing digital circuits
- A rapid prototype method
- A good way to hurt/train your brain
- Build higher frequency circuits
- Build low Jitter circuits

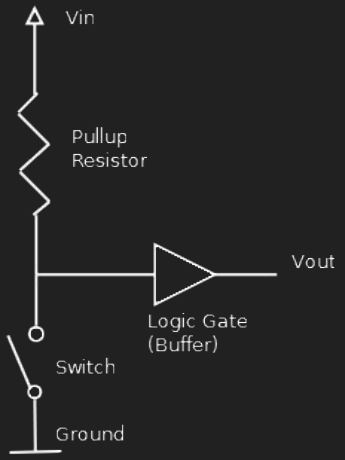
NAND truth table ->

Input	Input	Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

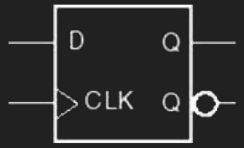


When you are here for a presentation about “Filipino Professional Golf Association” you might be in the wrong room.

Some digital electronics basics

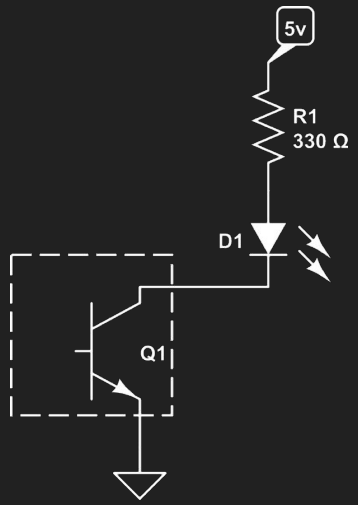


Pullup resistor



D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

Positive edge D-flip flop



Open collector
or
Pull low

When should/can FPGA’s be used?

In general

- Whenever bit banging is used with microcontrollers
- When protocol timing is very strict
- Implementing high speed interfaces/data throughput
- More interfaces are needed (i2c,rs232 ….)
- Custom/proprietary protocols are being used, and the chips are hard to get
- When you want to use hardware features like dual channel ram
- You have too much money and want to make an ASIC out of your design

  (The FPGA can be used as a prototype in this case)

When I used an FPGA

Some practical examples

- Handling PS/2 devices
- Generating VGA signals
- On the fly video signal filtering
- Convert 50->60Hz VGA signals and applying filters on the video
- Running softcore CPU’s
- Working on an implementation of a Commodore 64 in VHDL
    - Upgrading to Amiga ;-)

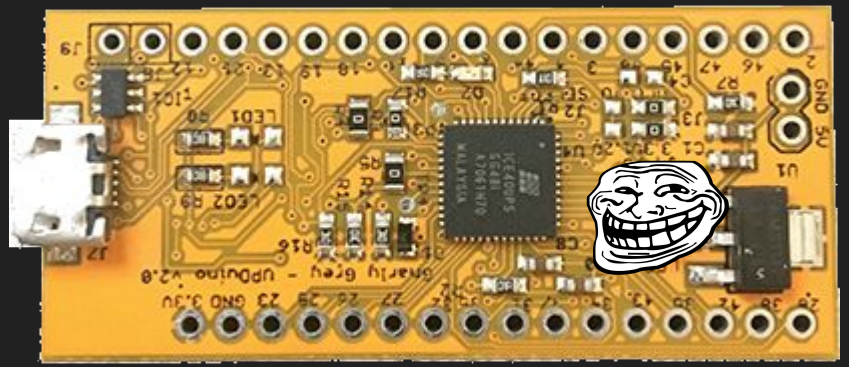
The HDL language Verilog

- A HDL (Hardware definition language)
- Looks like C but but works quite different
- Does the same job as VHDL, and like Emacs and Vim both are used

Test board for the workshop

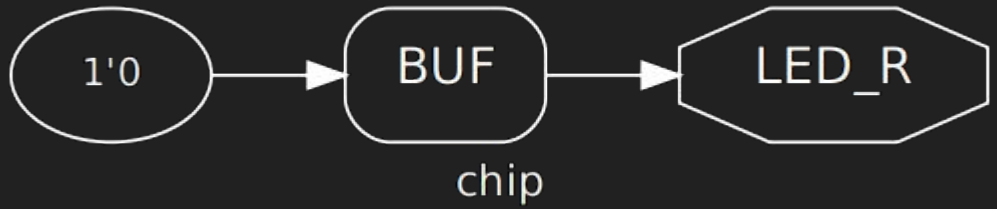
- UPDuino V2 (https://www.gnarlygrey.com/)
- Because it’s cheap (around 13 euros/ex pp)
- An open source toolchain is available (http://www.clifford.at/icestorm/)
- Easy to use
- Breadboard compatible
- Documentation available
- Nice bright LED to annoy your neighbours.

  (Learning how to dim it’s intensity is one of the goals for today)



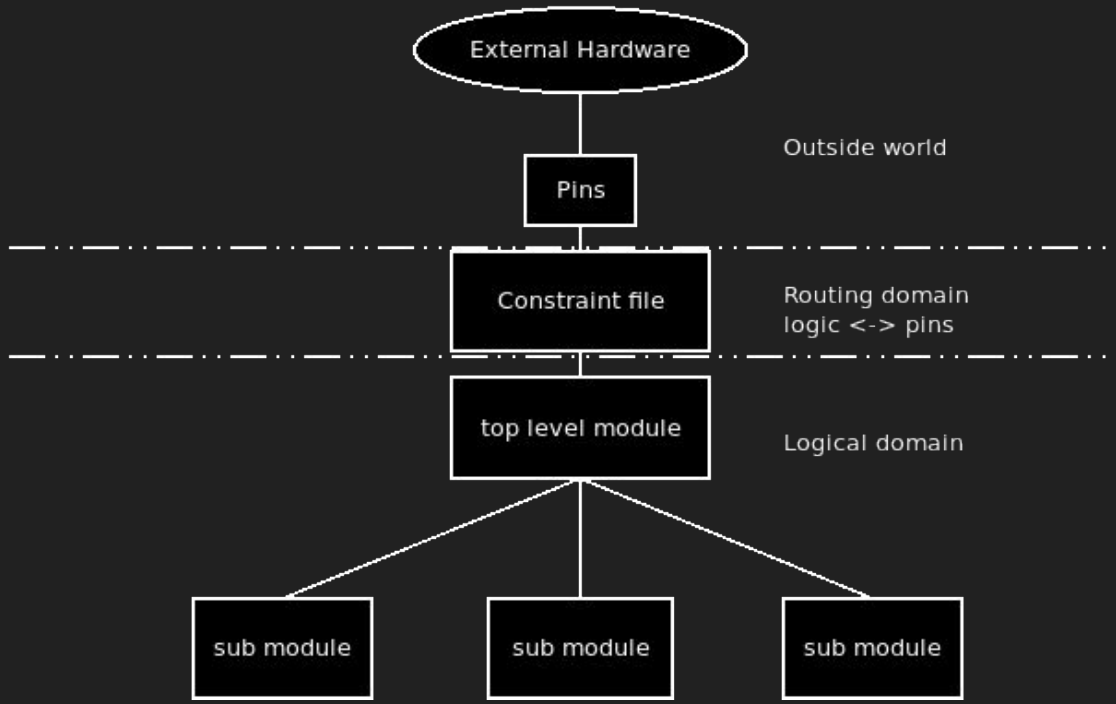
Hello world to the UPDuino

- Make sure the “Ice storm” project is installed on your laptop
- First get the project from: https://github.com/ranzbak/fpga-workshop
- Change directory into the 01-hello directory, type `make` and `make flash`
- After you hard work, enjoy the LEDs bright light



In the next few slides we are going to turn on green and blue as well.

Structural overview



Basic syntax Verilog

chip.v

```
// Example 01 Hello world
module chip (
  output  O_LED_R
  );

  // A wire
  wire  w_led_r;

  // Continuous assignment to a wire
  assign w_led_r = 1'b0;

  // Connecting the wire to the output
  assign O_LED_R = w_led_r;
endmodule
```

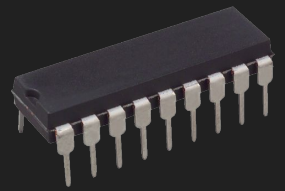
- Assign makes a permanent assignment to led_r
- 1’b0 , tells you that this is a 1 bit binary constant with value ‘0’ (low)
  (For literals, also ‘o’ Octal, ‘h’ Hex, ‘d’ Decimal)
- Endmodule ends the module block

First step modules

02-module/led.v

```
module led(output o_led_r);

  // Permanent assignments
  assign o_led_r = 1'b0;

endmodule
```

- Module defines a block with inputs and outputs, like a
- Projects are easier to develop when modular
- Modular designs promote code reuse



Basic syntax Verilog

Chip.v

```
/*
 * Top level module
 */
module chip (
  output  O_LED_R
  );

  // Module instantiation
  led my_led (
    // Connect the module output
    .o_led_r(O_LED_R)
  );
endmodule
```

- Wire: as the name suggests this is a wire that connects components in the design
- ‘led my_led’ instantiates the module led
  - The .o_led_r(O_LED_R) is the named binding of variables Variables can also be bound by order.
- Assign, like the previous example, but here it connects the wire to the output of the module

Constraint file

upduino_v2.pcf

- The IceStorm constraints file is simple
- Define what IO pin (number) is connected to what name (wire) in the design
- Input or output are determined by the modules input/output designation

```
# Constraints file
# <command> <name> <pin number>
set_io LED_R  41
~
```

Find the other pins in the schema included in the GIT repo:
./docs/UPDuino_v2_0_C_121217.pdf

Exercise

- Create the route from led.v to chip.v to upduino_v2.pcf to turn on all 3 colors

Let’s blink!

- Change directory to ‘03-blink’
- Type ‘make’ and ‘make flash’ to upload the project to the UPDuino
- After a few seconds put something over the LED
  - Or unplug the UPDuino until we deal with the chromatic overload

The clock

- The clock makes it possible for a circuit to change state over time
- Clocks synchronize changes (Like rowers on a classic ship)
- 10 kHz and 48MHz signals can be obtained from the internal oscillator
- Other frequencies can be generated with a PLL
- For higher precision an external crystal oscillator can be used (Not present on the UPDuino V2 board)

The Lattice ICE40 has two oscillators

SB_LFOSC – Low Frequency Oscillator (10 kHz)
SB_HFOSC – High Frequency Oscillator (48MHz)

The clock

In chip.v of the blink source we find

```
SB_HFOSC #(
  .CLKHF_DIV("0b01")
  ) u_hfosc (
        .CLKHFPU(1'b1),
        .CLKHFEN(1'b1),
        .CLKHF(clk)
    );
```

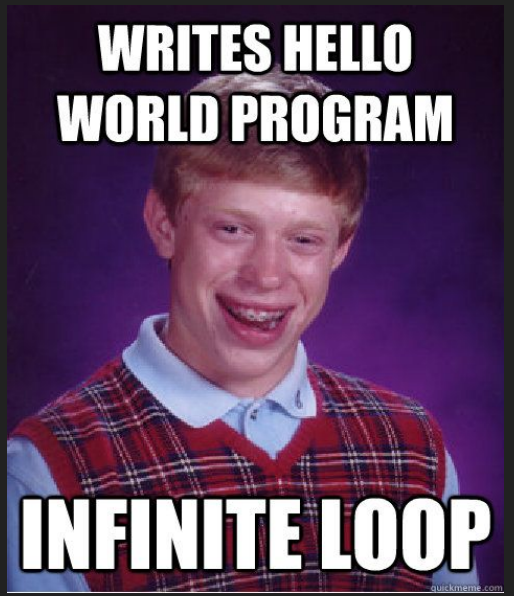
This piece of verilog calls the high frequency oscillator.
CLKHFPU powers up the oscillator CLKHF_DIV clock divider (48,24126MHz)
CLKHFEN Enables the output
CLKHF Clock output
This built in modules are called IP cores (**I**ntellectual **P**roperty cores)

* Even in an open source implementations they are called IP cores

The clock

In blink.v of the blink source we find

```
// always at clock pulse
always @(posedge i_clk)
begin
  if(i_rst)
  begin
    count <= 0;
  end
  else
  begin
    count <= count + 1;
  end
end
```



The always @(posedge clk) executes the code every positive edge of the clock

The code within the always block is evaluated top -> bottom

Visualizing wave forms

- Using GTKWave
- To see what happens on the inside
- Start GTKWave # make gtkwave

Why don’t you see the LED outputs change?

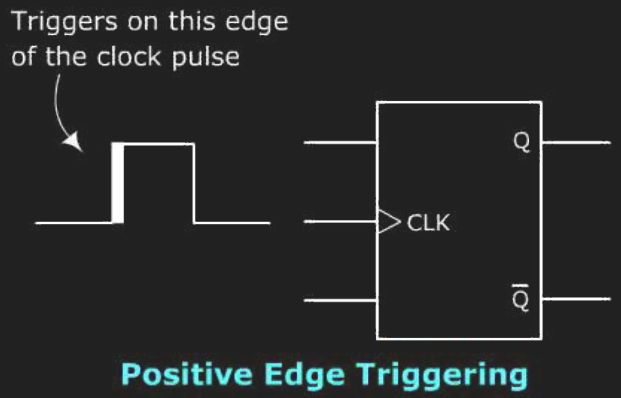
Always block

- To define a block of combinational logic (without clock)

```
always @ (input_1 or input_2)
begin
  and_gate = input_1 & input_2;
end
```

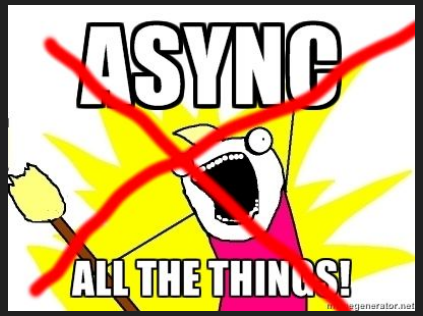
- To define a block of sequential logic (with clock)

```
always @ (posedge i_clock)
begin
  and_gate <= input_1 & input_2;
end
```



Modern designs try to use sequential logic as much as possible.
This to make debugging more convenient, and prevents hazards

Asynchronous reset



Asynchronous reset

- Only needed when the clock source is not reliable
- Can be connected to many modules, so makes your design slow
- Many examples on the internet use it, but avoid is not necessary

Synchronous

```
always @ ( posedge clock)
begin
if (reset)  //Do something
else     //Do something else
end
```

Asynchronous

```
always @ (posedge clk, negedge reset_n)
begin
if ( 'reset_n) //Then reset (active low).
else  // Do something else
end
```

Parameters

Change to example: 04-parameter

- Promotes reusability
- Makes modules more flexible
- Can be used to aid simulation timing

Change the blink example to work in simulation and the real FPGA without changing the module.

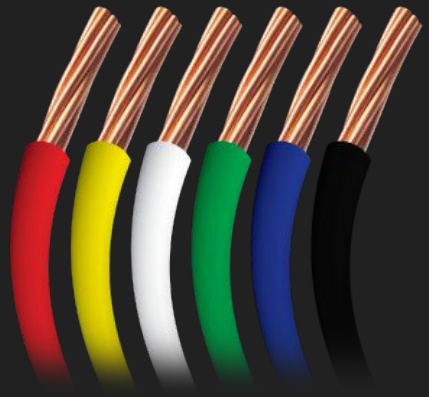
```
// Count bits
parameter r_bit=25;
parameter g_bit=24;
parameter b_bit=23;

// Counter register
reg signed [25:0] count;

// Permanent assignments
assign led_r = count[r_bit];
assign led_g = count[g_bit];
assign led_b = count[b_bit];
```

```
// Instanciate the module
blink #(
  .r_bit(4),
  .g_bit(5),
  .b_bit(6)
  ) uut (
  .clk(clk),
  .rst(rst),
  .led_r(led_r),
  .led_g(led_g),
  .led_b(led_b)
  );
```

Wires, Registers and Variables

Wires

- Connect components in the design
- Do not have state when not driven
- Can only be driven with the assign statement



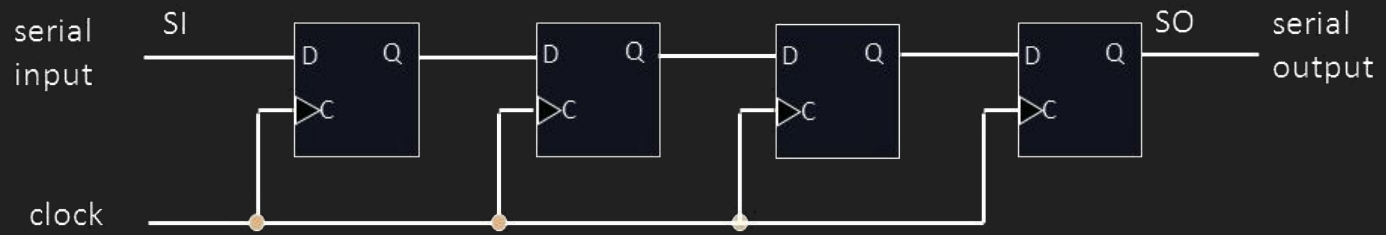
```
module chip (
  output  LED_R,
  output  LED_G,
  output  LED_B
  );

  wire clk, led_r, led_g, led_b;

  assign led_r = 0'b1;
  assign led_g = 0'b1;
  assign led_b = 0'b1;

  assign LED_R = led_r;
  assign LED_G = led_g;
  assign LED_B = led_b;
endmodule
```

Wires, Registers and Variables

Registers

```
wire A_in;
reg A_out, B_out, C_out;
always @( posedge clk )
begin
 A_out <= A_in;
 B_out <= A_out + 1;
 C_out <= B_out + 1;
end
```

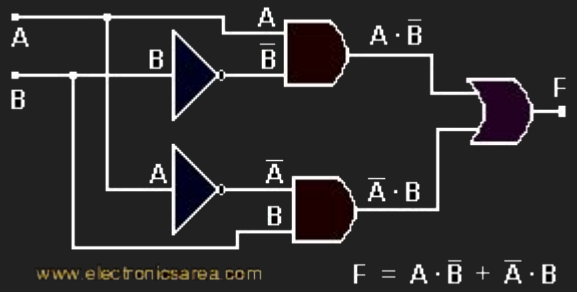
- Stores the state until the next assignment of the register
- Are assigned by a **non-blocking assignment** <=
- Gets its value at the end of the combinational logic
- Assigning one value first and later another value, means the last value is the only one going to be assigned.
- Values are assigned at the end of the block.



Wires, Registers and Variables

Variables

- Variables store state like registers
- Variable are assigned using the **blocking assignment =**
- Variables can only be used within always/initial blocks
- Where registers are created as flip-flops, variables are transformed into logic
- Gets its value assigned immediately

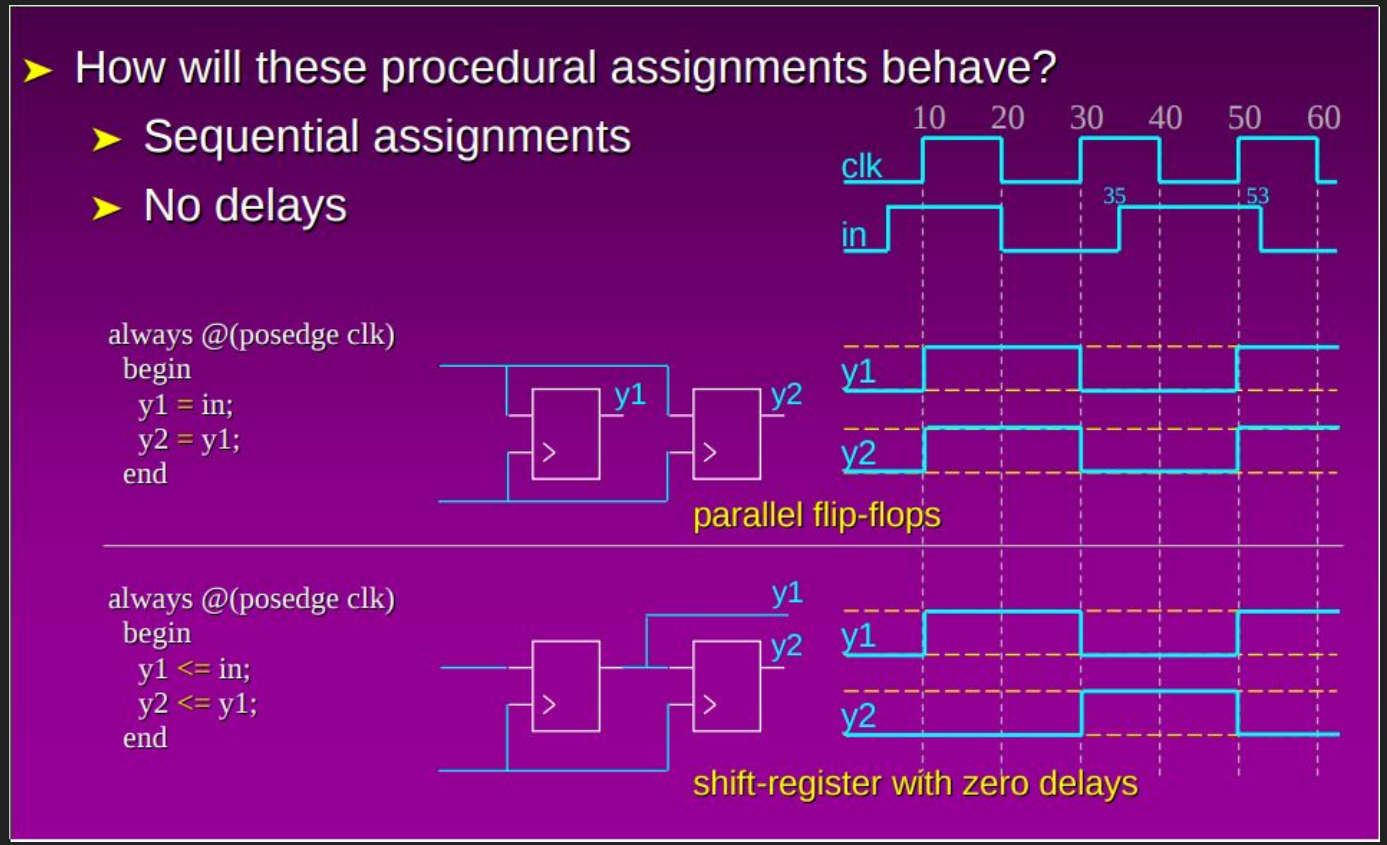


<- not the actual logic*

```
always @(posedge i_clk)
begin
  y1 = in;
  y2 = y1;
end
```

Wires, Registers, Variables and clock

- How will these procedural assignments behave?
  - Sequential assignments
  - No delays



Borrowed from:
Sutherland HDL

Wires, Registers and Variables

- Mixing variable and register assignments is not a good idea (#warnings)
- When using variables in multiple concurrent blocks the result is unpredictable
- When using registers in concurrent blocks, the results are predictable

Wires, Registers and Variables

Next change directory to 05-varreg, and flash it into the board.

```
// Set RED and GREEN to on (low is on)
led_r_reg <= 0'b1;
led_g_var =  0'b1;

// Copy the register and the variable to the output wires
led_r <= led_r_reg;
led_g <= led_g_var;

// Set RED and GREEN to off (higt is off)
led_r_reg <= 1'b1;
led_g_var = 1'b1;

// After programming for the first time uncomment the two lines below
//led_r <= led_r_reg;
//led_g <= led_g_var;
```

What happens if this code is flashed into the FPGA?

(hands for both, none, only red or only green LED’s on)

10 Minute break

Bit Arrays or numbers

Bit Arrays or numbers

In Verilog Registers can be grouped together in arrays

These arrays of bits can be used as numbers

```
reg [25:0] count;
```

```
parameter p_size=1000;
reg [p_size:0] r_buffer;
reg [$clog2(p_size)-1:0] r_buffer_index;
```

The array is always declared highest number first, indicating the most significant bit.

Arithmetics can be done on these arrays:

+ , - , * , / , %

(Be aware that multiplications and divisions can take up more components, adding to propagation delay)

```
// always at clock pulse
always @(posedge clk)
begin
  if(rst)
  begin
    count <= 0;
  end
  else
  begin
    count <= count + 1;
  end
end
```

Bit Arrays or numbers

Within arrays single bits can be picked out using

```
// Permanent assignments
assign led_r = count[25];
assign led_g = count[24];
assign led_b = count[23];
```

Also two or more arrays can be concatenated together

```
reg [15:0] join;
always @* begin
    join = {AS_1, AS_2};
end
```

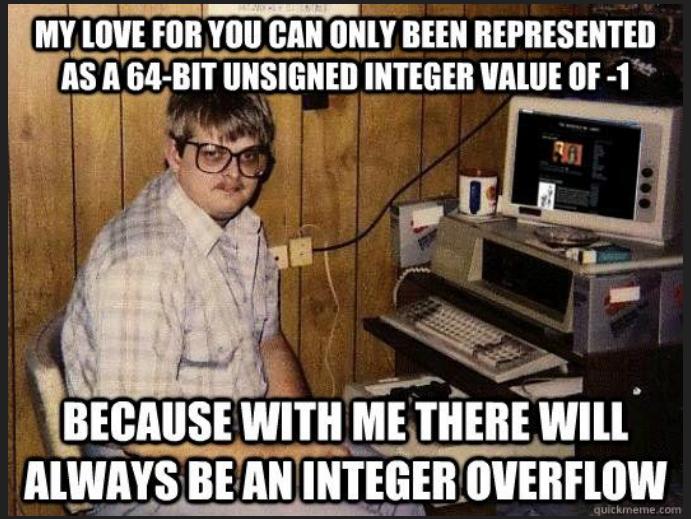
Bit Arrays or numbers

Numbers can be signed

```
reg signed [25:0] count;
```

Or unsigned, (The unsigned is implicit in Verilog)

```
reg [25:0] count;
```



Bit Arrays or numbers

Verilog, it seems, is strongly inclined towards unsigned numbers. Any of the following yield an unsigned value:

- Any operation on two operands, unless both operands are signed.
- Based numbers (e.g. 12′d10), unless the explicit “s” modifier is used) e.g. 12’sd10
- Bit-select results
- Part-select results
- Concatenations

Dim the bright light

Exercise:
Go back to the 04-parameter example, and implement the following

- Use the clock to dim the 3 LED’s to ⅛ of the intensity while still color cycling

Tasks

- Tasks are declared inside a module
- Tasks can have any number of inputs and outputs
- Tasks can call other tasks and functions
- Tasks can drive global variables external to the task
- Tasks can use non-blocking and blocking assignments

They are like functions in C :-)

```
task sum;
  input  [7:0] a, b;
  output [7:0] c;
  begin
    c = a + b;
  end
endtask

initial begin
  reg [7:0] x, y , z;
  sum (x, y, z);
endtask
```

Functions

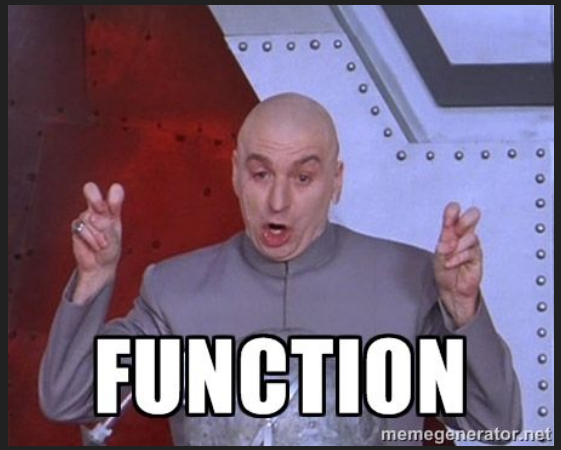
- Functions can have any number of inputs but only one output (one return value)
- The return type defaults to one bit unless defined otherwise
- Functions can call other functions but they cannot call tasks
- Non-blocking assignments in a function are illegal

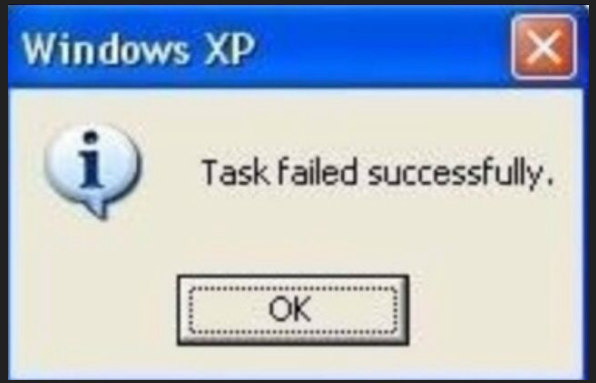
Like macros in C

```
module function_example ();
  reg r_Bit1=1'b1, r_Bit2=1'b0, r_Bit3=1'b1;
  wire w_Result;

  function do_math;
    input i_bit1, i_bit2, i_bit3;
    reg   v_Temp; // Local Variable
    begin
      v_Temp  = (i_bit1 & i_bit2);
      do_math = (v_Temp | i_bit3);
    end
  endfunction

  assign w_Result = do_math(r_Bit1, r_Bit2, r_Bit3);
endmodule
```

Tasks and functions

- Try to do the dimming of the previous exercise (at home)
  - With a task (example 7)
  - With a function (example 8)





If statements

- If statements are always used in a (always) block
- If multiple statements need to be placed inside an if block they need to be enclosed by ‘begin’ and ‘end’

```
if ([expression])
  Single statement

// Use "begin" and "end" blocks for more than 1 statements
if ([expression]) begin
  Multiple statements
end

// Use else to execute statements for which expression is false
if ([expression]) begin
  Multiple statements
end else begin
  Multiple statements
end
```


```
if (r_command == c_skip_rom); begin

end
```

For loops

For loops

- Loops work differently in HDL languages than in languages like C.
- For loops in synthesizable code are used to expand replicated logic.

The code below shows what the equivalent is of the ‘C’ for loop in Verilog.
As you can see it’s not quite the same.

```c
// C code
int main(void){
  int data[10];

  // Example Software Code:
  for (int i=0; i<10; i++)
    data[i] = data[i] + 1;
}
```

```
// Verilog code
Always @(posedge clock)
Begin
  if (index < 10)
  begin
    data[index] <= data[index] + 1;
    index       <= index + 1;
  end
End
```

For loops

The below examples are equivalent, and show what the for loop does in Verilog.

```
// Performs a shift left using a for loop
always @(posedge i_Clock)
begin
  for(ii=0; ii<3; ii=ii+1)
    r_Shift_With_For[ii+1] <= r_Shift_With_For[ii];
end
```

```
// Performs a shift left using regular statements
always @(posedge i_Clock)
begin
  r_Shift_Regular[1] <= r_Shift_Regular[0];
  r_Shift_Regular[2] <= r_Shift_Regular[1];
  r_Shift_Regular[3] <= r_Shift_Regular[2];
end
```

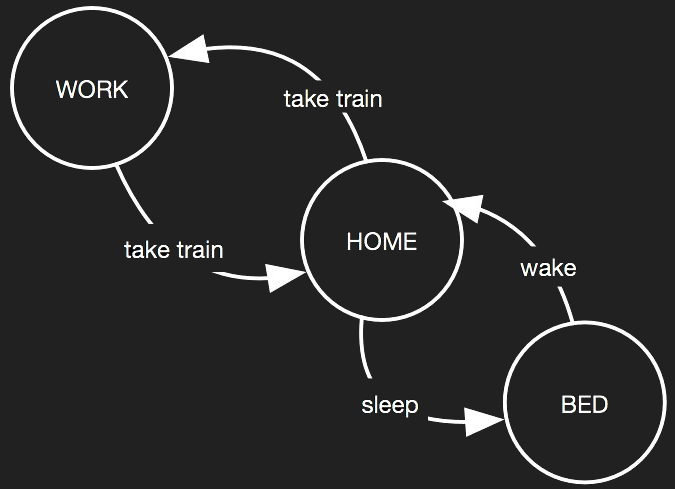
In this example makes evident that for loops just replicate code in Verilog.

Case statement

The case statement in Verilog is very important for creating state machines.

State machines are the basis of most digital synchronous systems.

```
// Here 'expression' should match one of the items (item1 or item2)
case (expression)
  item1 : [single statement]
  item2 : begin
            [multiple statements]
          end
  default : [statement]
endcase
```



Case statement

For readability Cases should be enumerated, But since Verilog doesn’t have enum, we use parameter

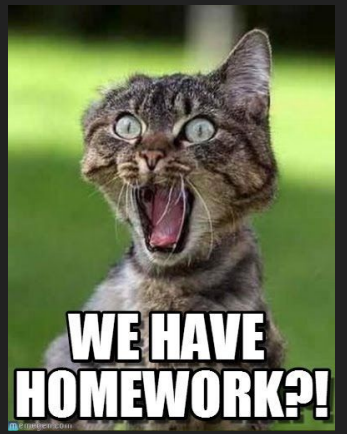
```
parameter
  start        = 0, // Start the sequence
  waitforinput = 1, // Wait for input
  decode       = 2, // Decode received character
  execute      = 3, // Implement command
  done         = 4; // Done, return to start
```

Exercises

- To play around with state machines, go to 08-case, and see if another message than SOS can be sent with the case statement.

After the workshop you can :

- Also try to use the dim function to dim the blinking light
- Cycle through the colors, for every message cycle



I/O pins in verilog

I/O pins in verilog

Words of caution:

- Do not connect the output pin directly to ground, or VCC doing so can damage the device.
- Also the max I/O voltage rating is 3.6 V

There are three kinds wires to the ‘outside’ world

- Output - drives pin
- Input - reads pin inputs
- Inout - drives pin, when output is set to ‘Z’ input can be read

I/O pins in verilog

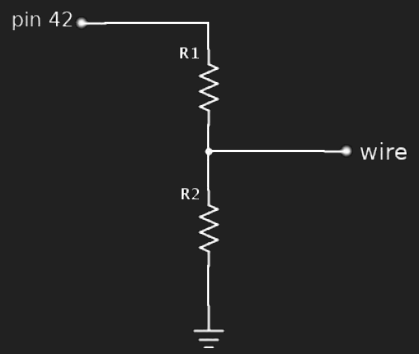
Use the Morse example, to ghetto test an Input to the FPGA

Add the following to the pcf file
set_io INPUT_1 42

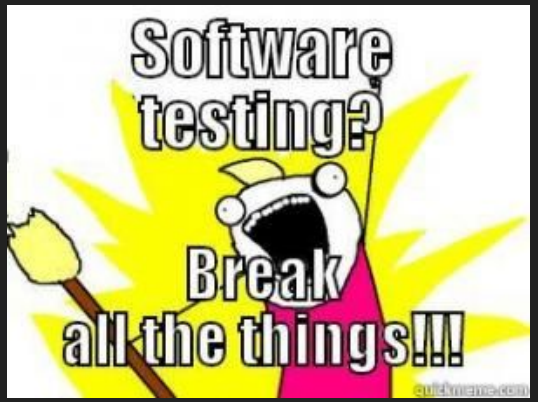
Define the INPUT_1 as an input in chip.v
Assign the pin to reset, and write code that resets the status and counter

Connect pin 42 using two 12k resistors like this →

Connect the wire to either gnd or + 3.3V for ‘0’ and ‘1’



Simulation



Simulation is ‘the unit test’ of the FPGA development.

It can be used to create rules that stop the build if broken.

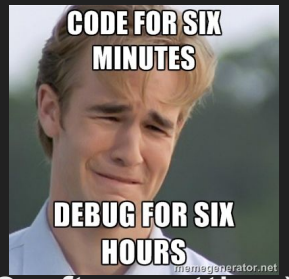
- A mock top level module is used (simulate.v)
- Simulation requires a testbench file (simulate_tb.v)
- Simulations can have code that test for the state of signals

run:
make sim

```
always @(posedge clk) begin
  if( uut.count_cur > 1023 ) begin
    $display("%0t: %d", $time, uut.count_cur);
    $stop;
  end
end
```

Debugging

Sometimes simulating is not enough, you really need to ‘see’ what is happening.
For this we can use the Wave viewer GTKWave. (example 11)



make gtkwave

Introduce a bug, in cycle.v
cut 3 lines at line 56

```
    // reverse counter duty cycle
    if (count_duty_next == 0 || count_duty_next == 255)
      duty_dir <= ~duty_dir;
  end
end
```

Move it to line 66 (62 after cutting)

```
// reverse counter duty cycle
if (count_duty_next == 0 || count_duty_next == 255)
  duty_dir <= ~duty_dir;
```

Watch the behaviour change, and see if you can relate that to the code change.

(Hint, watch duty_dir at 1953100 s)

Interacting with other hardware

- When an inout pin is either ‘0’ or ‘1’ it will drive the pin low or high
- When the pin is set to ‘Z’ it’s in ‘high impedance’ mode and input can be read
- Combining the three states ‘0’,’1’,’Z’ into one inout pin, is called a three state buffer

With Yosys at this time the line below doesn’t work!

```
assign bidir = oe ? a : 1'bZ ;
```

This because Yosys has limited tri-state support.

Interacting with other hardware

Because Yosys has limited support for tristate buffers, they have to be declared using the SB_IO structure.
This sets up the IP block explicitly.

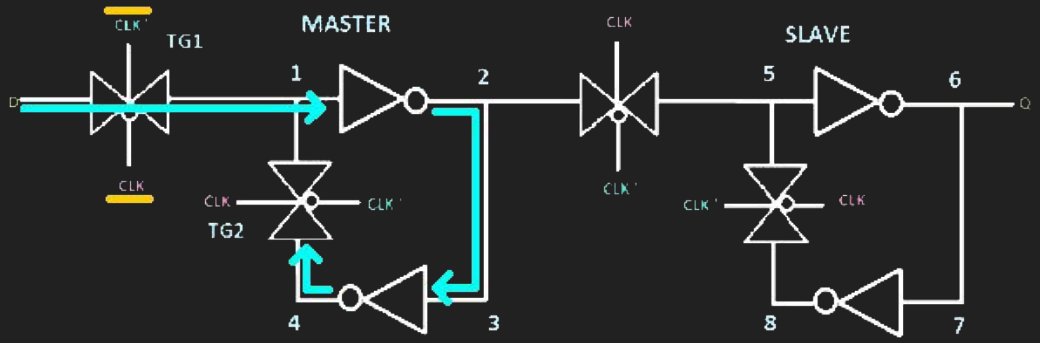
```
// Set the one-wire port to tri-state
SB_IO #(
  .PIN_TYPE(6'b 1010_01),
  .PULLUP(1'b 0)
) io_buf_one_wire (
  .PACKAGE_PIN(IO_ONE_WIRE),
  .OUTPUT_ENABLE(owr_out),
  .D_OUT_0(!owr_out),
  .D_IN_0(owr_in)
);
```

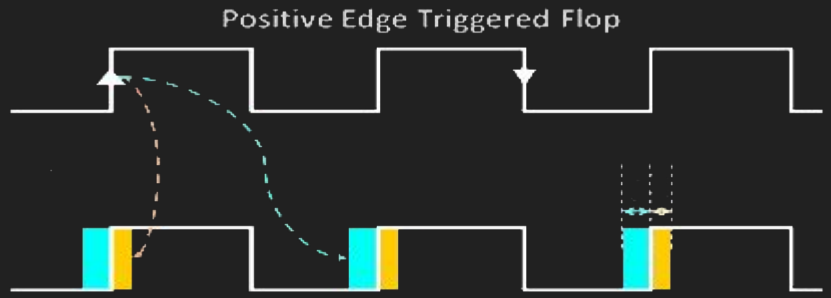
See example 12-detect/chip.v for a working example

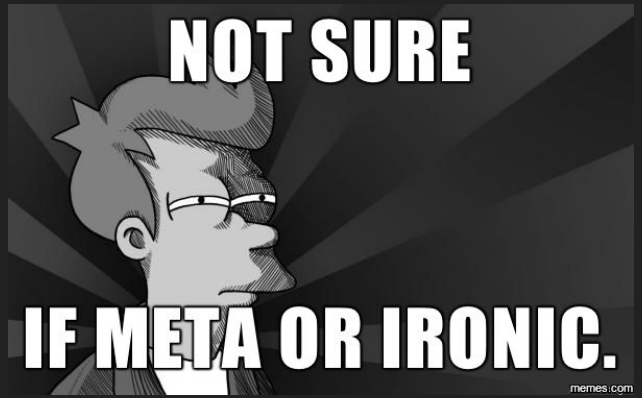
Some final thoughts

- Continue from here by trying projects of your own
- Buy a logical analyser, if you plan to do more serious projects
  - Build one with your freshly gained FPGA skills
- Some hardcore subjects, you should know about but don’t have to understand fully in the next four slides :-)

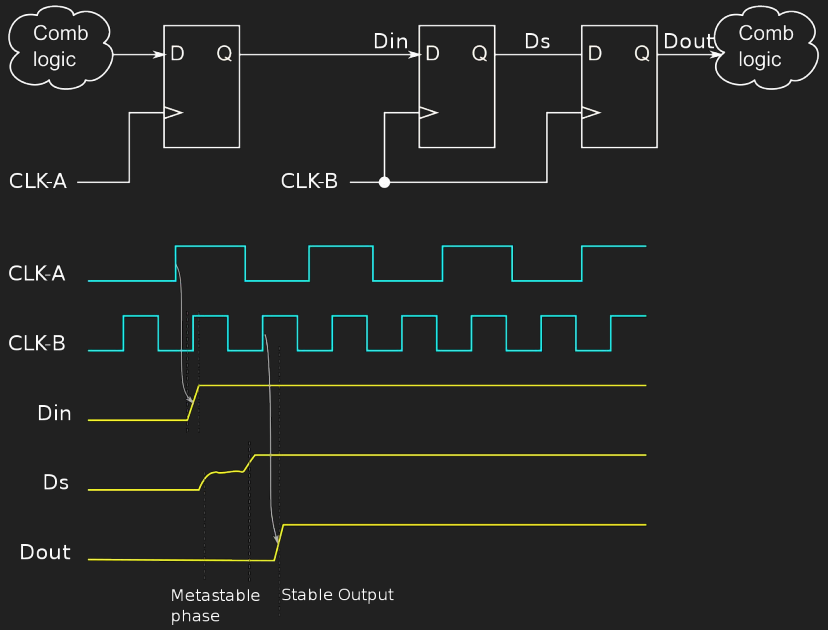
Meta stability





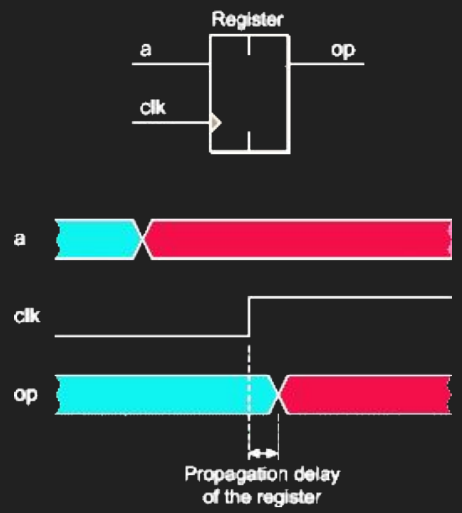


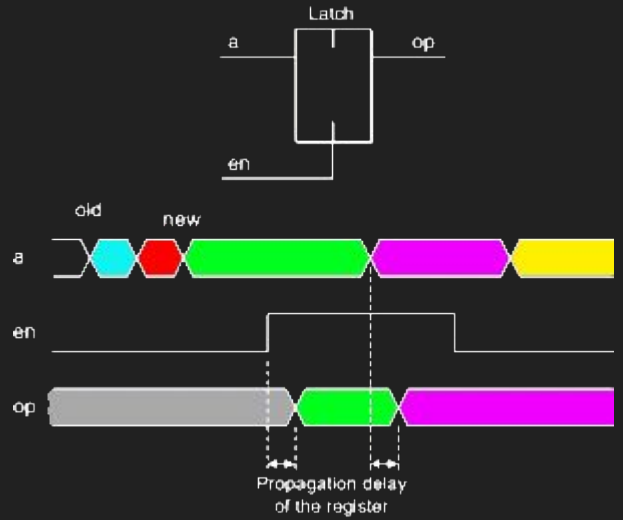
Meta stability



Avoiding Latches

- Latches change state during at any phase of the clock cycle
- The timing simulation software will not simulate latches correctly
- They can complicate designs and debugging





Avoiding Latches

All synchronised logic within a ‘always @(posedge clk)’ is safe.
So how do latches get created by accident?

Outside of synchronous logic, the following leads to latches

- Incomplete assignments
- Incomplete case statements

```
always @ (i_select)
  begin
    if (i_select == 2'b00)
      o_latch <= 4'b0101;
    else if (i_select == 2'b01)
      o_latch <= 4'b0111;
    else if (i_select == 2'b10)
      o_latch <= 4'b1111;
    // Missing one last ELSE statement!
  end
```

```
// BAD
always @ (sel, a)
  begin : latching_if
    if (sel == 1)
      f = a;
  end
end

// GOOD
always @ (sel or a or b)
begin : pure_if
  f = b;
  if (sel == 1)
    f = a;
  end
end
```

Suggestions

Please send your Suggestions to paul<at>printf.nl


That's all Folks!
VIA 9GAG.COM