FYSIKUM

Digital System Construction - 1

Lecture 5: Designing for FPGA

Frequently-asked questions/ Good design practice

Timing

Types and records in VHDL

Lab 4 intro: digital stopwatch

Frequently asked questions

- Do functions/processes “remember” the values of variables?
  - Yes!
    - For example, a "counter" variable remembers the last count.
  - Different from software, where function variables can be re-initialized each time the function is called
- Signal/port names:
  - Can a signal in an architecture can have the same name as a port of one of the components?
  - Yes!
    - `port map (select => select, ....);`
  - Good practice to propagate common signals throughout the design:
    - `reset, clock, ...`

Frequently asked questions

- How do you use (read from) the value of an output port within the architecture?
  - You can't read outputs directly.
  - If you need to, create a signal for internal use, and then assign that signal to the output port:
    - ```
      int_buf <= (int_buf and enable);
      output <= int_buf;
      ```
  - Another method is to declare a port as bi-directional (`inout` instead of `out`)
    - Not recommended unless a bi-directional port is really needed for your design
      ```

Signals versus variables

- Example: counting with a signal:

```
architecture behav of clk_div is
   signal count: std_logic_vector (3 downto 0);
   signal slowck : std_logic := '0';

 begin
  cproc : process(clk)
   begin
    if rising_edge(clk) then
       if (count = "1111") then
         count <= "0000";
         slowck <= not slowck;
       else
         count <= count + 1;
       end if;
    end if;
  end process;
end behav;
```

count is only updated once in the process

Signals versus variables

- Counting with a <u>variable</u>:

```
architecture behav of clk_div is
  signal slowck : std_logic := '0';

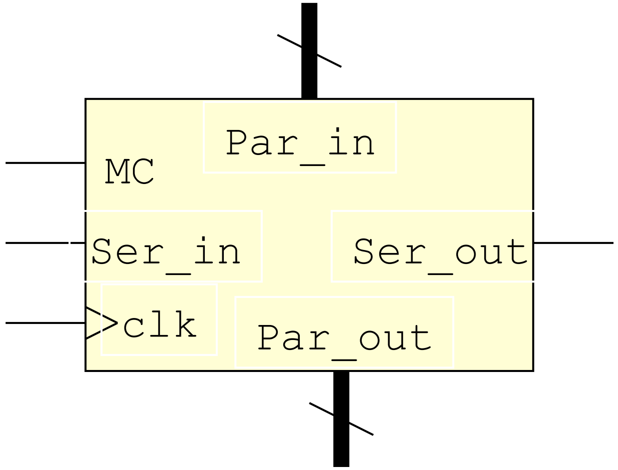
 begin
  cproc : process(clk)
    variable count : integer := 0;
   begin
    if rising_edge(clk) then
       count := count + 1;
       if (count > 15) then
         count := 0;
         slowck <= not slowck;
       end if;
    end if;
  end process;
end behav;
```

Only one signal needed outside the process

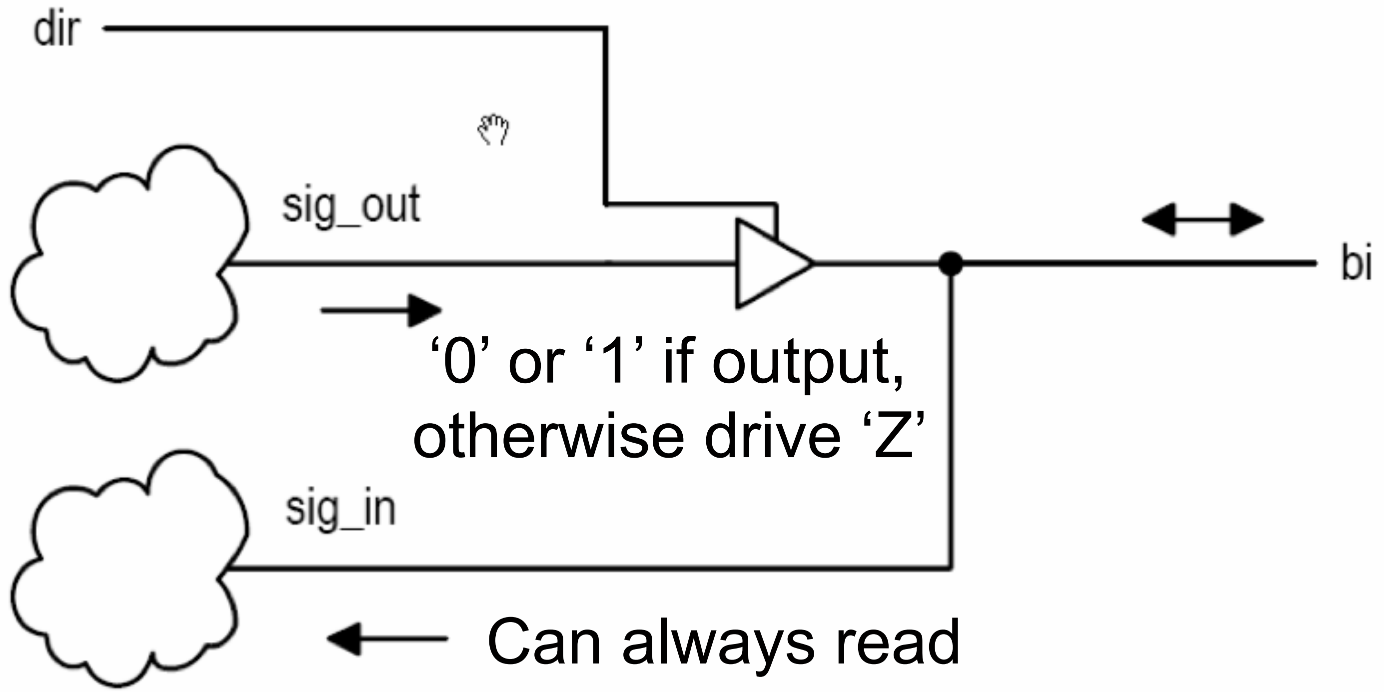
count can be changed more than once with variable assignment (:=)

Component port maps

- Good to connect <u>named signals</u> to ports
  - Even if they are not used elsewhere, you can follow them while simulating
  - Helps avoid accidental connections
- Inputs must <u>always</u> be connected
- Outputs don't need to be connected
  - But unused logic is trimmed anyway
  - Can also use: Par_out => open
- Connect outputs to <u>unique</u> signals
  - Avoid multiple drivers
  - <u>Exception</u>: Tri-state ('Z') connection to a bus



Bi-directional signals (with tri-state)



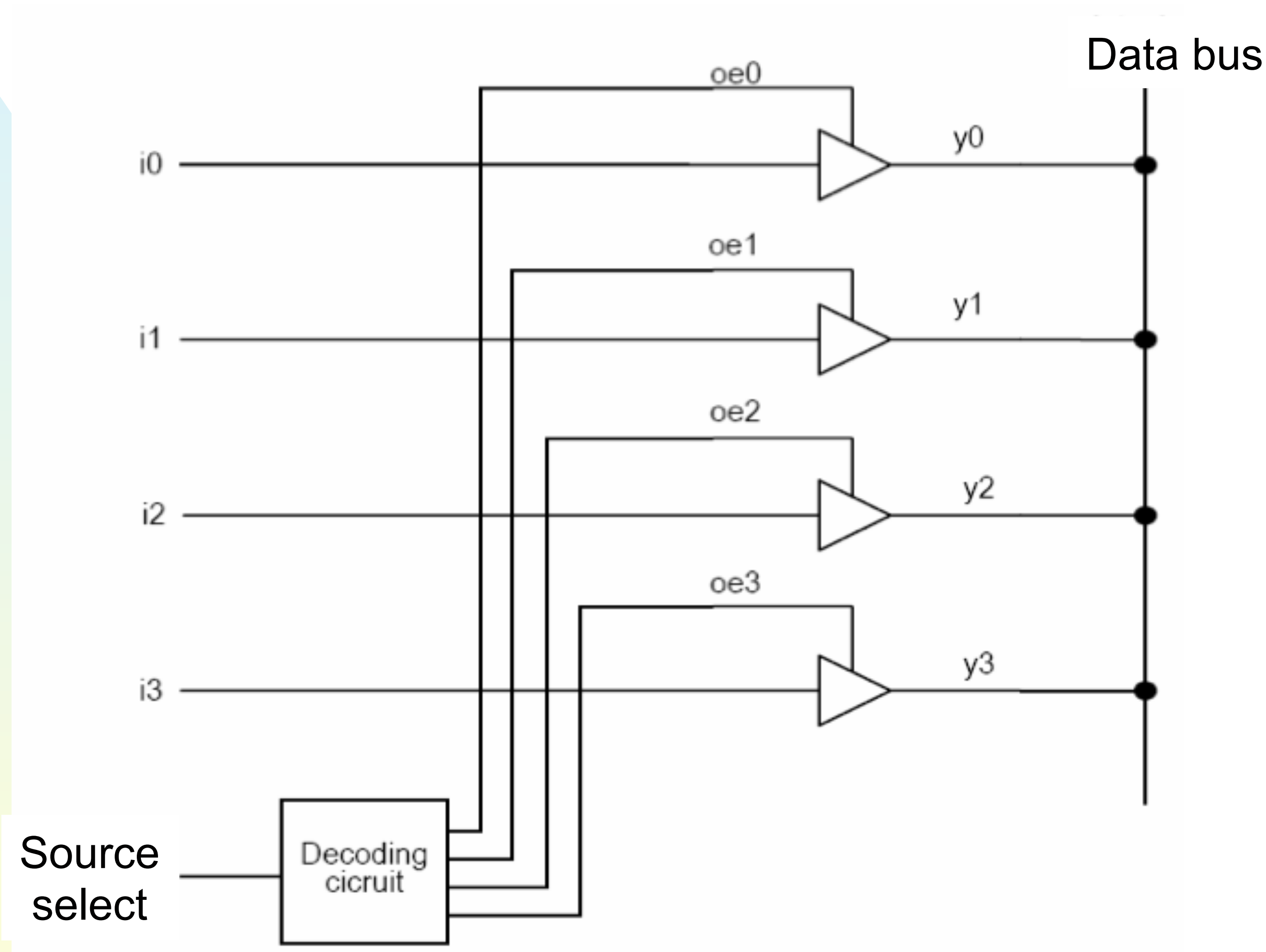
Implementing in VHDL

```
entity bus_IO is
        Port(Din:  in    std_logic;
                Dout: out        std_logic;
                IO_pin: inout std_logic;
                wr_en: in        std_logic );
end bus_IO

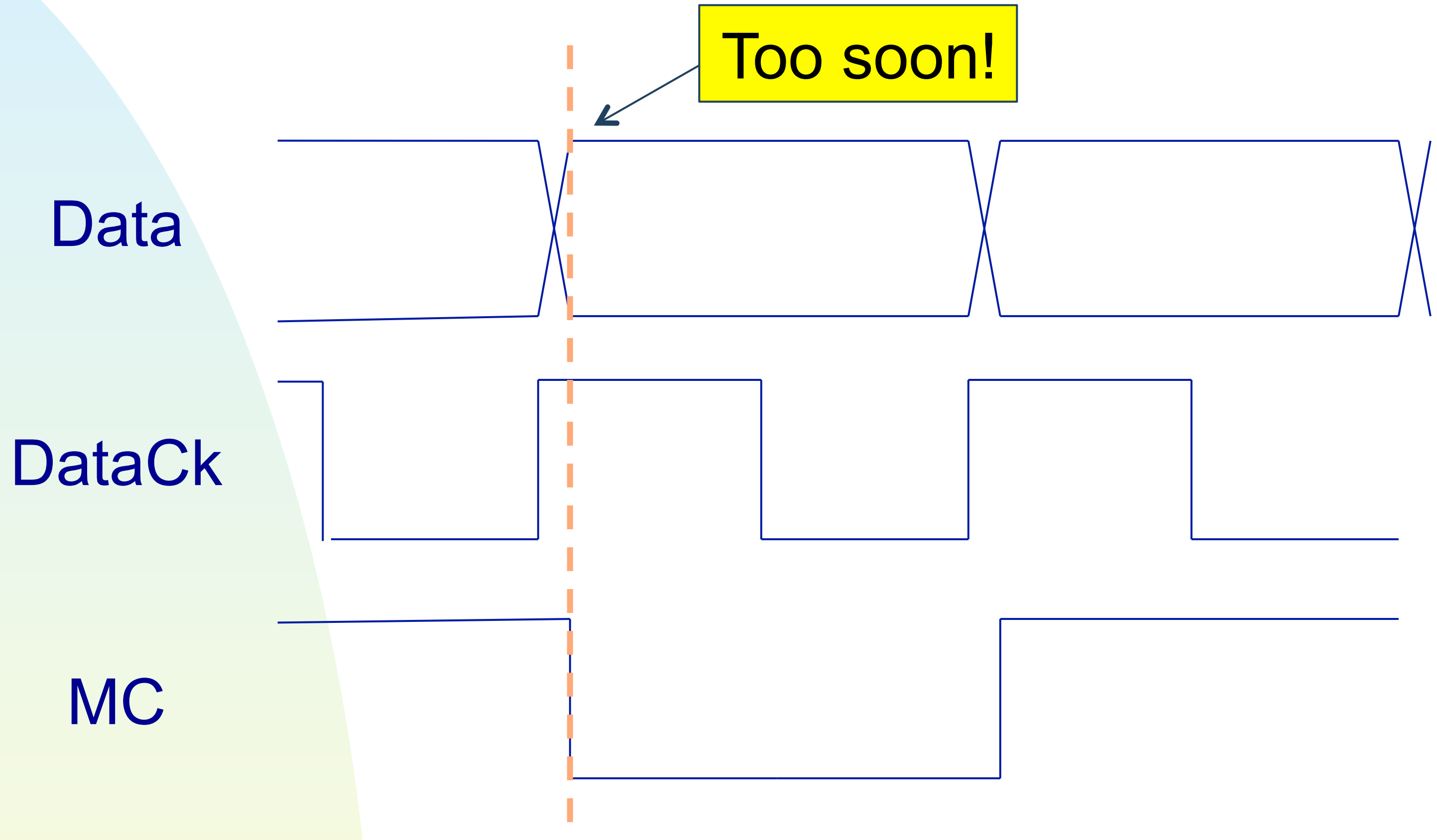
......

buffer_rw : process (Din, IOpin, wr_en) -- can also be clocked
        begin
            if (wr_en) then
                IO_pin <= 'Z';   -- high impedence if not driving
            else
                IO_pin <= Dout;
            end if;
            Din <= IO_pin;       -- Always can read
end process;
```

Tri-state bus example

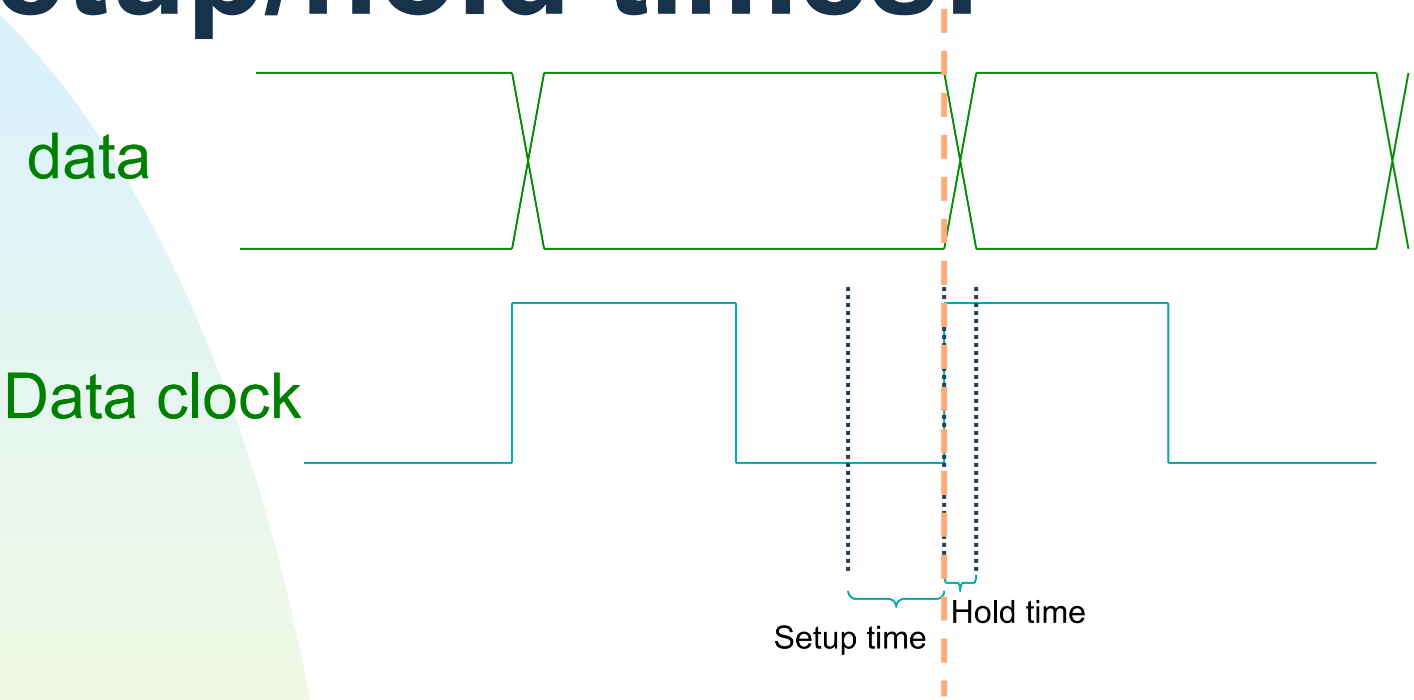


Be careful with timing…




```
if falling_edge(MC) then newdata <= data;
```

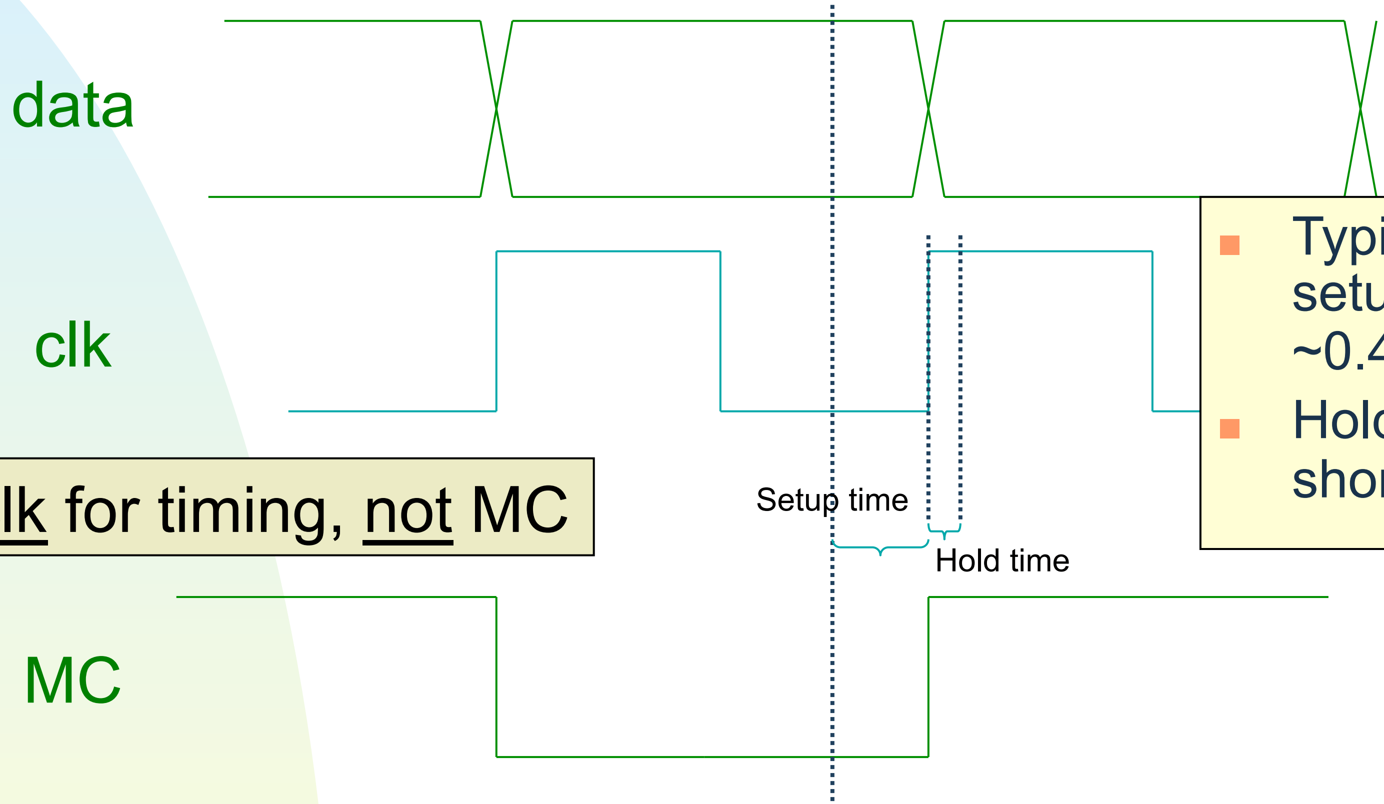
Flip-flops have minimum setup/hold times:



Setup time: Data must be stable before clock edge

Hold time: Time for flip-flop output to stabilize (short)

A better approach:



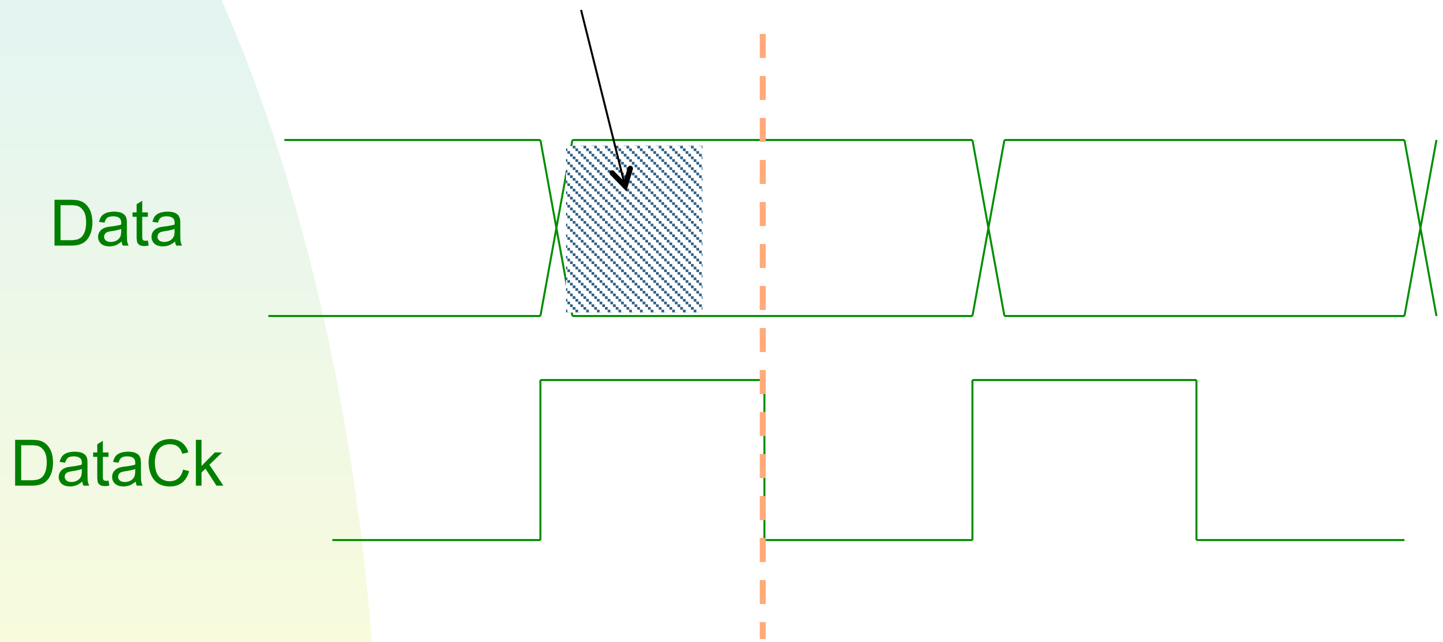
- Typical setup time: ~0.4 to 2 ns
- Hold time: short

Use clk for timing, not MC

```
if rising_edge(clock)and MC='0' then
     newdata <= data;
```

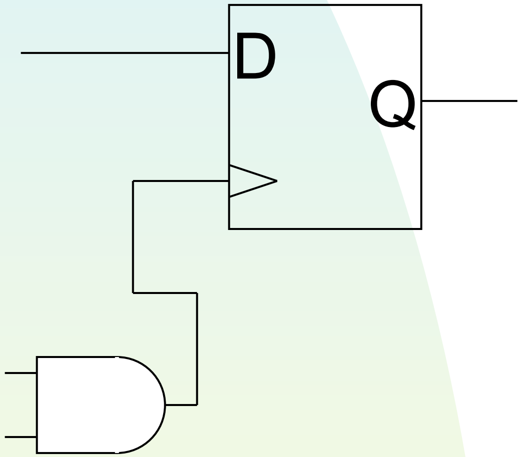
Even this can be risky:

New data can take some time to become valid (propagation delays, setup/hold times, etc)




```
if falling_edge(DataCk) then newdata <= data;
```

"Gated Clocks"



- Flip-flops driven by combinatorial logic
- Disadvantages:
  - Timing differences between FFs driven by the same source (propagation)
  - Unpredictable: hard to get good synthesis performance
  - Gated clock edge degraded as more FFs load the same driver
- If you need to use a logic-derived signal to drive flip-flops, you can:
  - Pass the logic output through a flip-flop (driven by another clock in your design), or
  - Distribute the new timing signal through a global (or regional) FPGA clock buffer

Global clock buffers

- Distribute selected timing signals synchronously
- Inputs can be external timing signals, internally synthesized clocks, gated signals, etc
- Can use Clocking Wizard in IP Catalog
- 7-series global buffer in pure VHDL:

```
Library UNISIM
use UNISIM.vcomponents.all

component bufg  -- (or ibufg)
     port (
          I: in std_logic;
          O: out std_logic);
end component;
```

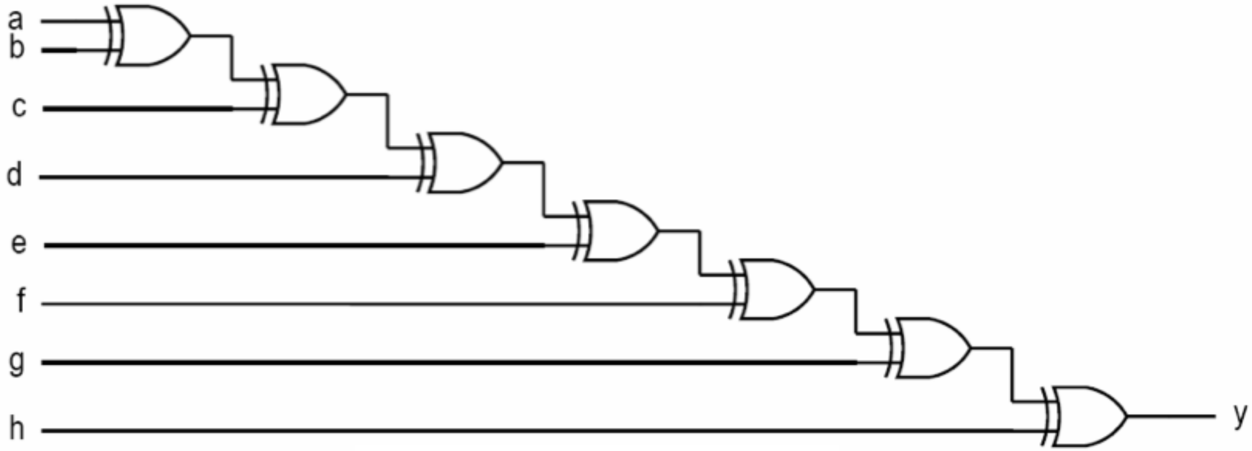
Good clocking practice

- Drive flip-flops with global clocks
  - Use gated logic to “enable” clock
- Bring high-quality clocks to the FPGA
  - Crystal oscillators have low jitter
  - “Jitter cleaner” circuits can improve performance of non-crystal clocks
- Use designated clock pins, if available
  - Designed for optimal input to global buffers
- Distribute internally-generated clocks via global or regional clock buffers as well

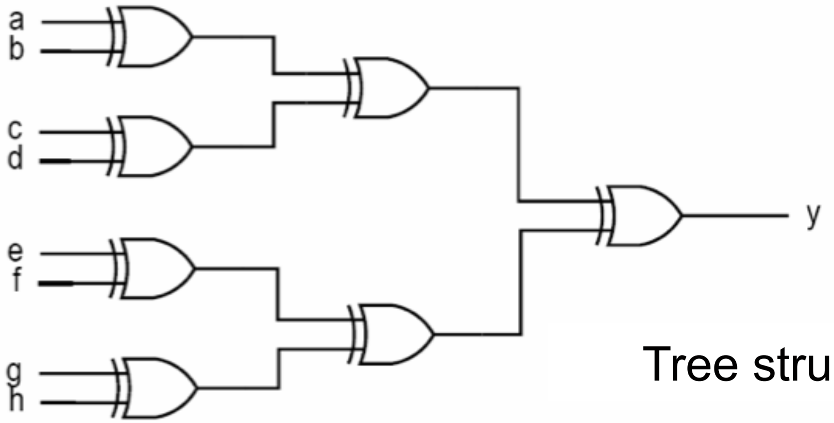
Don't blindly trust the synthesis!

- Modern tools very powerful, but not perfect
  - Some HDL descriptions can lead to non-optimal results
    - Synthesis software doesn't know exactly what you want
    - Optimal solution can't be derived, optimization process is an iterative search in a large "space" of possible solutions
    - Good HDL code gives a good starting point for that search
- Pay attention to errors and warnings
  - Even buggy designs can synthesize, but not work right!
  - Check timing/mapping reports
    - Constraints met? Reasonable resource use?
- Check RTL schematics in ISE
  - Structure look like you expect?

Example: XOR cascading chain vs. tree structure

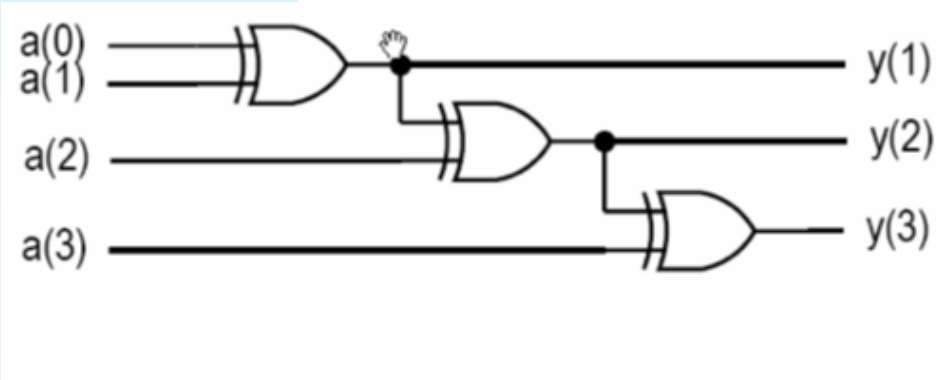


Cascading chain

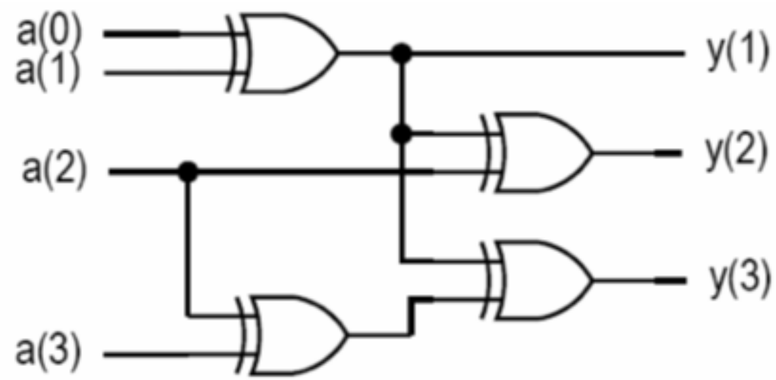


Tree structure

Can choose to optimize for speed / area / both



Less logic
(area)



Smaller delay
(speed)

Types in VHDL

Constants and variables

```
constant number_of_bytes : integer := 4;
constant number_of_bits : integer := 8*number_of_bytes;
constant e : real := 2.718281828;
constant prop_delay : time := 3 ns;

variable index: integer := 0;
variable start, finish : time := 0 ns;
```

- Constants can be declared globally (in architecture) or within sequential code (process, procedure, functions)
  - Generally declared with a value
- Variables are declared in a piece of sequential code; not normally visible outside that process, etc.

Scalar types

- A scalar type has discrete values
  - No composite elements
- All values are ordered
  - Each value has an implicit position number
  - Predefined relational operators work
- Scalar types include:
  - Numeric (integer, floating point)
  - Physical (for example, time)
  - Enumerated types
- `integer` is a scalar type representing all whole numbers representable on the host computer

Scalar types

Example of declaration and use in a package

```
type apples is range 0 to 100;

package int_types is
      type small_int is range 0 to 255;
end package int_types;

use work.int_types.all;
entity smaller_adder is
      port (a, b  : in small_int;
            s :    out small_int);
end entity small_adder;
```

Floating-point types

Not usually used for synthesis

```
type input_level is range -10.0 to 10.0;
type probability is range 0.0 to 1.0;

variable input_A: input_level;
```

Assignments use variables or constants of same type

```
type day_of_month is range 0 to 31;
type year is range 0 to 2100;

variable today: day_of_month := 19;
variable start_year: year := 2005;
```

Not legal to make this assignment:

```
start_year := today;
```

Integer/vector conversion in std_logic_unsigned

```
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.conv_std_logic_vector;

signal vector : std_logic_vector (v_width downto 0);
integer int_var;

--std_logic to integer:

int_var <= conv_integer(vector);

--integer to std_logic:

vector <= conv_std_logic_vector(int_var, vector'length);
```

data types have <u>attributes</u> → vector'length

Some useful VHDL attributes

- `type'pos(value) -- integer position of value in the type`
- `type'val(i) -- value of type at integer position I`

- `array'length – number of elements in an array/vector`
- `array'range -- range of an array/vector`
- `array'low -- lowest subscript of an array/vector`
- `Array'high -- highest subscript of an array/vector`

- `signal'event -- true if signal changes value`

(Not an exhaustive list…)

Enumerated types

```
type alu_function is (disable, pass, add, subtract,
                      multiply, divide);
type octal_digit is ('0','1','2','3','4','5','6','7');

variable alu_op: alu_function;
variable last_digit: octal_digit := '1';

alu_op       :=    subtract;
last_digit   :=    '7';
```

Enumerated type attributes

```
type alu_function is (disable, pass, add, subtract,
                      multiply, divide);
```

- Each enumeration literal has an integer "position number" (0, 1, 2, 3, etc.)
- The first (left-most) enumeration literal has position 0, the next has position 1, etc.

  1
  - `int_variable <= alu_function'pos(pass);`
  - `alu_option <= alu_function'val(3);`

  subtract

Data types: Record

- Useful for bundling groups of signals
  - Especially different signal types
- Example: CPU memory bus:
  - `address(), data()` -- vectors
  - `read_en, write_en, chip_sel` -- bits
  - `clock, reset` – timing

Declaring and using records

```
type memory_bus is record
        address, data          : std_logic_vector(15 downto 0);
        rd_en, wr_en, c_sel : std_logic;
        clk, reset             : std_logic;
end record;

signal address1      :      std_logic_vector(15 downto 0);
signal bus1, bus2    : memory_bus;

bus1.address <= address1; -- Assign signal to record element

bus1.wr_en <= '1';  -- Assign value to record element

bus2.data <= bus1.data; -- Copy part of a record

bus2 <= bus1;  -- Copy entire record
```

Record example (ATLAS): L1Calo trigger topology processor (L1Topo)

Package: L1TopoDataTypes.vhd

```
library ieee;
use ieee.std_logic_1164.all;

Package L1TopoDataTypes is
        type ClusterTOB is record
             Et : std_logic_vector (7 downto 0);
             isol:  std_logic_vector (4 downto 0);
             eta: std_logic_vector (5 downto 0);
             phi: std_logic_vector (5 downto 0);
        end record;
        type JetTOB is record
             Et1 : std_logic_vector (8 downto 0);
             Et2:  std_logic_vector (9 downto 0);
             eta: std_logic_vector (4 downto 0);
             phi: std_logic_vector (4 downto 0);
        end record;
        type ClusterArray is array (natural range <>) of ClusterTOB;
        type JetArray is array (natural range <>) of JetTOB;

        -- After selection and sorting, use generic type for all TOBs.
        type GenericTOB is record
             Et : std_logic_vector (9 downto 0); -- Pad unused bits with zeros
             eta: std_logic_vector (5 downto 0);
             phi: std_logic_vector (5 downto 0);
        end record;
        type TOBArray is array (natural range <>) of GenericTOB;
end;
```

Can declare with any width

Using in an algorithm:

```
library ieee;
use ieee.std_logic_1164.all;
use work.L1TopoDataTypes.all;

entity DeltaPhiIncl1 is
        generic(InputWidth    : integer := 8);
        port(
                Tob          : in  TOBArray(InputWidth - 1 downto 0);
                Parameters : in  ParameterArray;
                ClockBus    : in  std_logic_vector(2 downto 0);
                Results     : out std_logic_vector(NResultBits - 1 downto 0));
end DeltaPhiIncl1;

. . . . . .

 deltaPhi_calc1 : for i in 0 to (maxCount - 2) generate
        deltaPhi_calc2 : for j in (i + 1) to (maxCount - 1) generate
                dphiCalc_inst : entity work.DeltaPhiCalc
                        port map(
                           phi1In      => Tob(i).phi,
                           phi2In      => Tob(j).phi,
                           deltaPhiOut => deltaPhi(i)(j)
                        );

        end generate;
end generate;
```

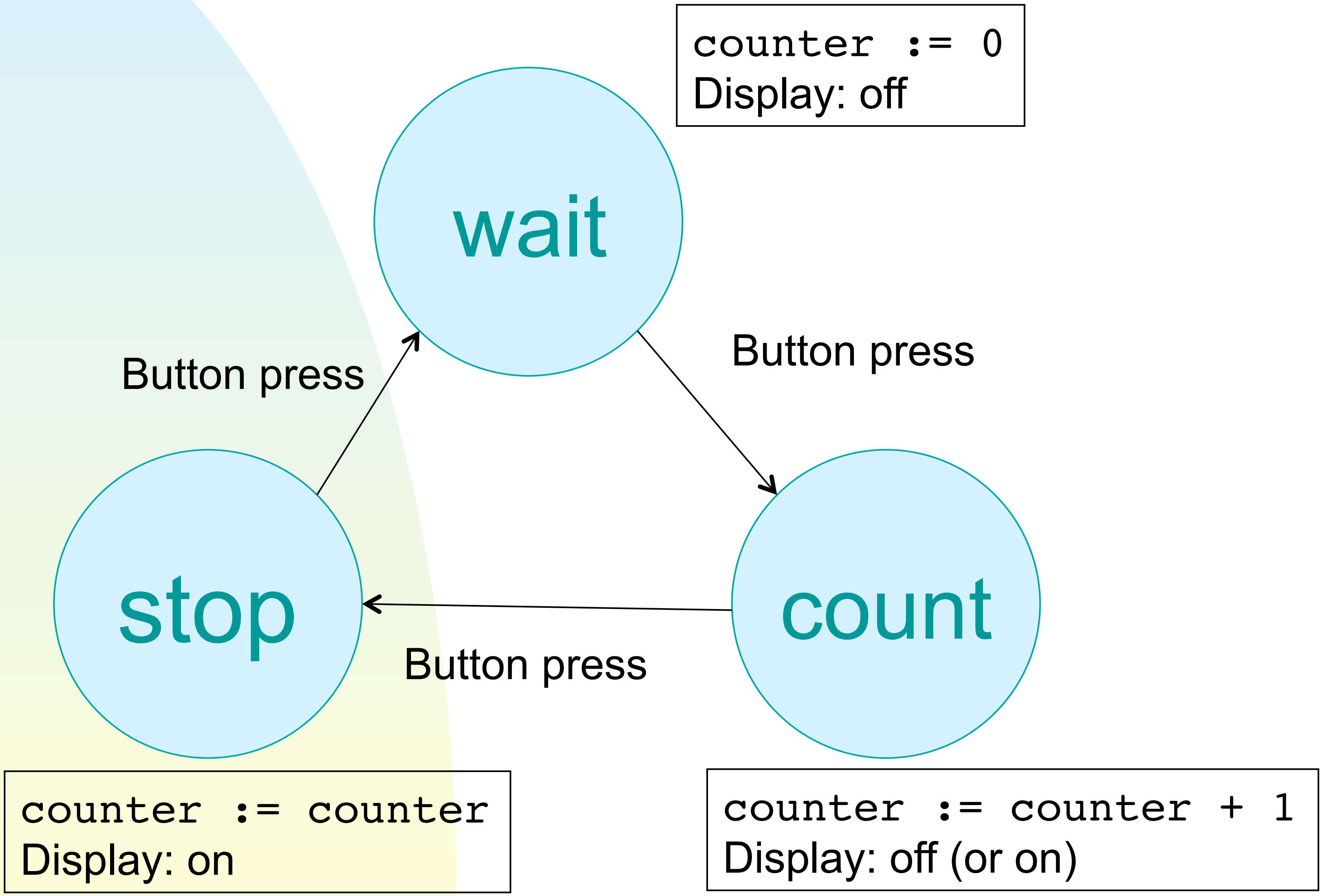
Compare all TOB pairs

Lab 4: Digital counter (stopwatch)

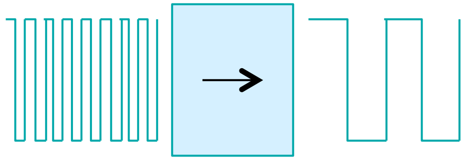
Basic stopwatch function

- Input: single pushbutton
- Output: 7-segment display
  - (time to 1/100 second)
- Behavior:
  - Push once: start timer
  - Push again: stop, display time
  - Push again: clear timer

State diagram (simplified)



Clock divider



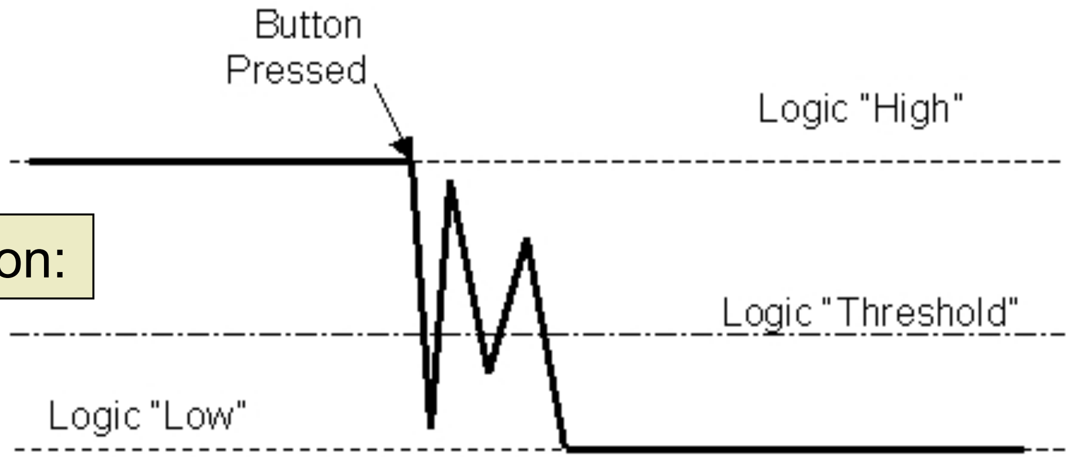
- <u>Problem</u>:
  - FPGA has a <u>fast</u> clock (e.g. 100 MHz)
  - We need a <u>slow</u> clock (100 Hz)
- <u>Solution</u>: a clock divider
  - Internally a <u>counter</u>, clocked at 100 MHz
  - Reset counter to zero every 0.01 s
    - Max count (100.000.000 / 100) = 1.000.000
- Good to have a <u>symmetric</u> output:
  - If count < (1.000.000 / 2) then output <= ‘1’
  - Otherwise output <= ‘0’;

‘Debouncing’ a button

(not needed for lab)

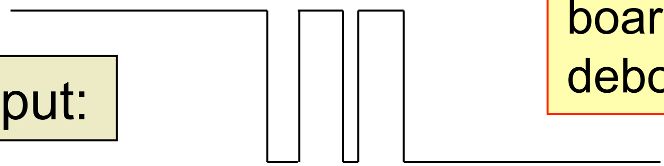
One approach:
use a counter

Electrical signal from button:

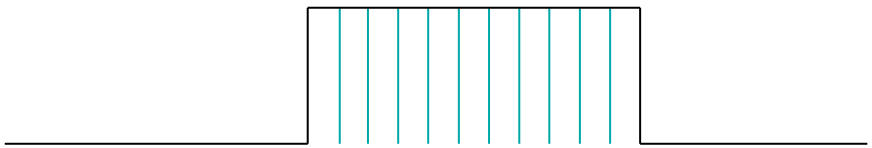


Buttons on your lab boards are passively debounced (capacitor)

Logical signal seen at FPGA input:



Debounce circuit waits a few clock cycles before the output is allowed to fall:



Stopwatch block diagram