

Introduction to digital design with Migen and Litex

V1.0

Franck Jullien,

- ▶ ARM, Staff HW bringup engineer
- ▶ Hardware / Firmware / FPGA,
- ▶ Hardware security research for fun,
- ▶ Open sources enthusiast.



@fjullien06



<https://github.com/fjullien>



What are we going to talk about ?

- ▶ Description of FPGAs
- ▶ Digital design challenges
- ▶ Migen: introduction and workshops
- ▶ LiteX: introduction and workshops

Digital Design – Base elements

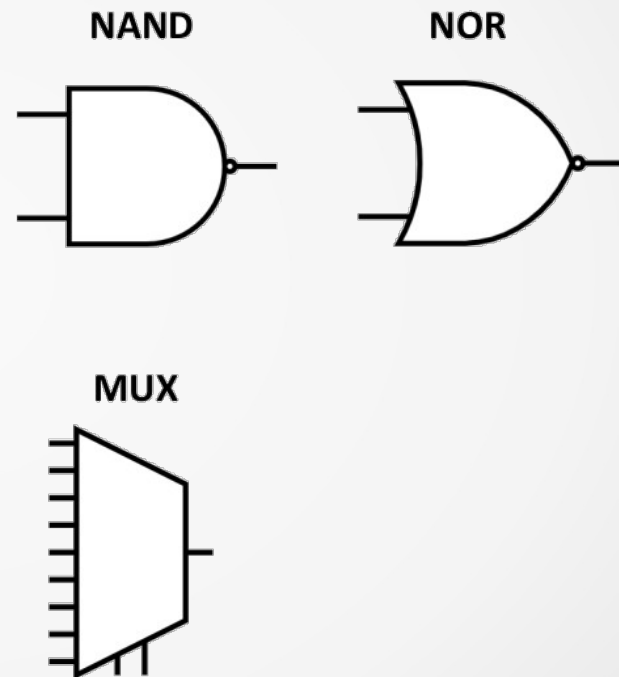
Hardware consist of a few simple building blocks

1. Combinatorial

- ▶ “Instant” state changes, e.g.:

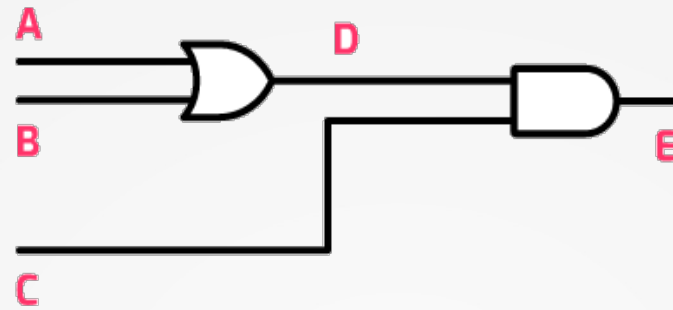
Classical gates (especially NAND & NOR)

Multiplexer (MUX)



A Scientist's Guide to FPGAs – Alexander Ruede – ICSC 2019

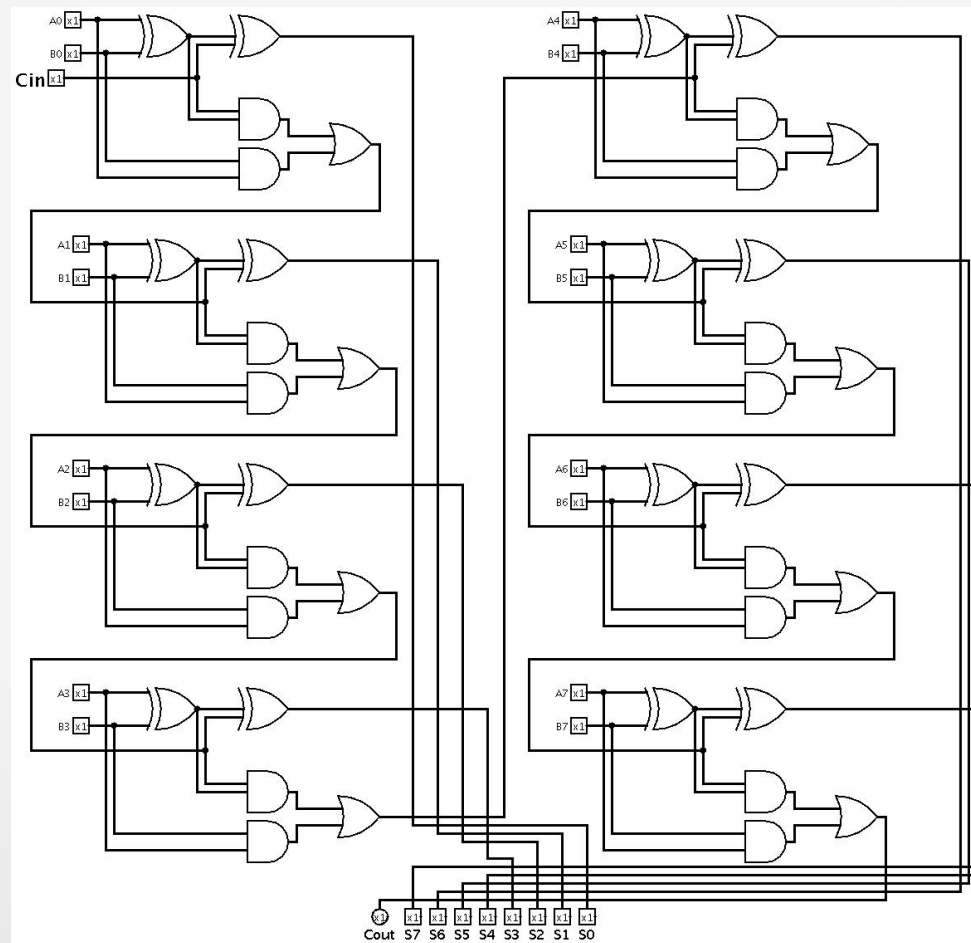
Digital Design – Truth table



A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

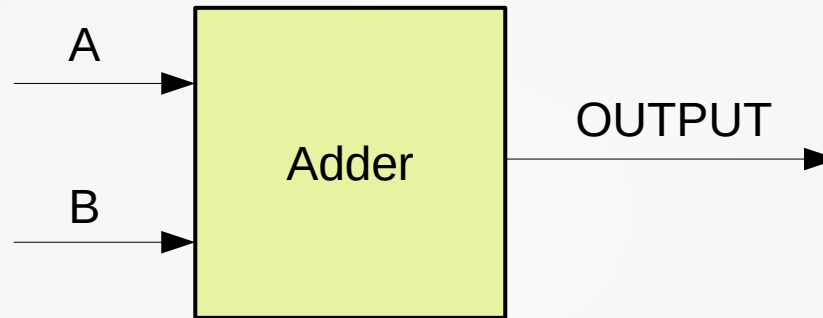
Digital Design – 8 bits adder

Design of an 8 bits adder



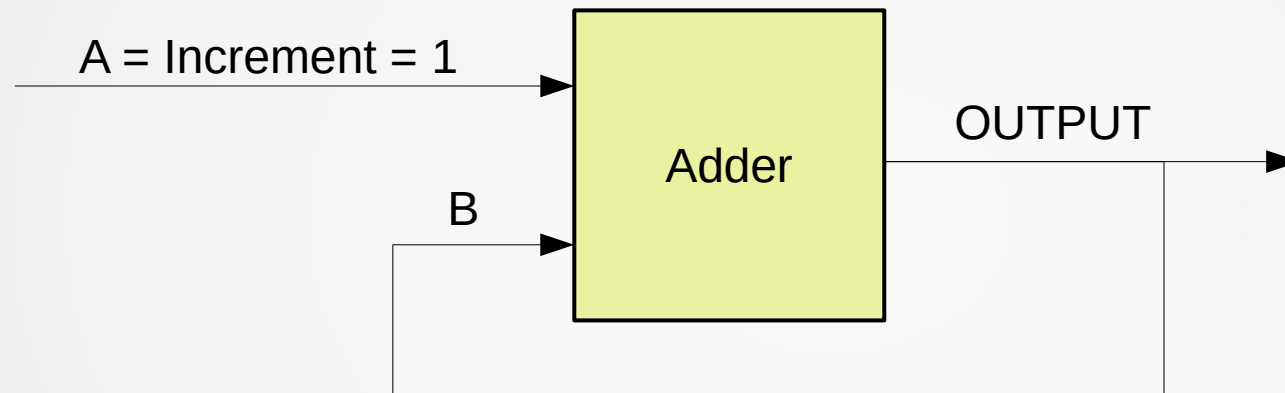
Digital Design – 8 bits counter

Design of an 8 bits counter



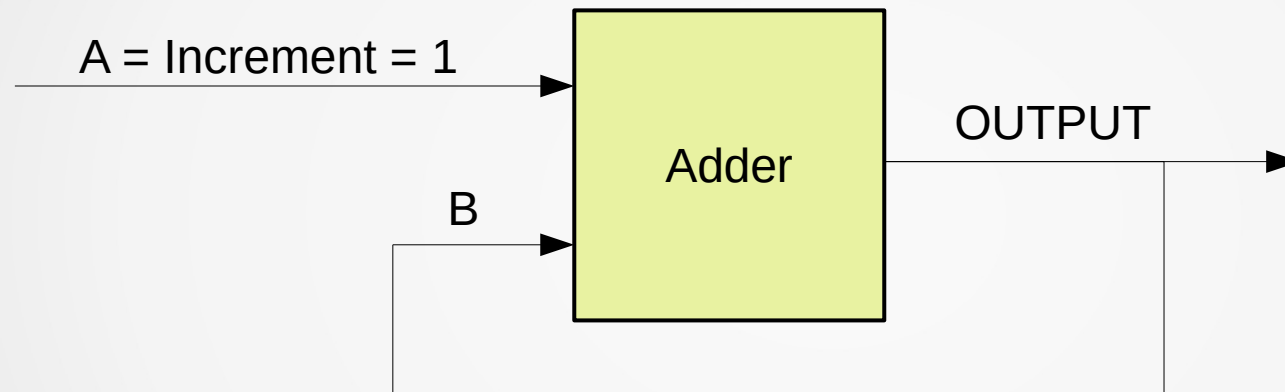
Digital Design – 8 bits counter

Design of an 8 bits counter



Digital Design – 8 bits counter

Design of an 8 bits counter

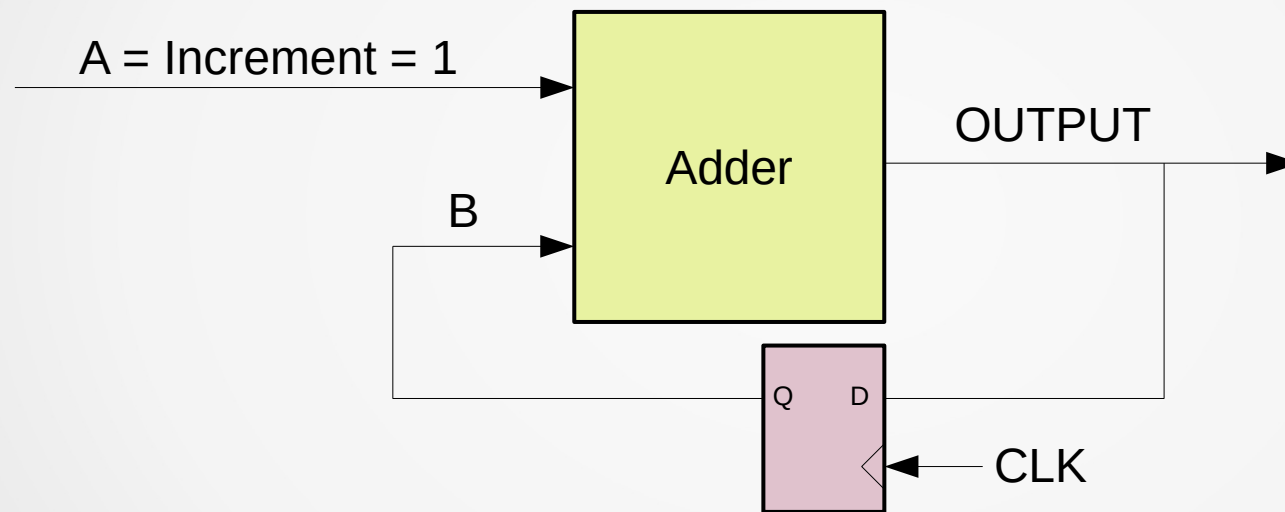


We have a infinite loop (combinatorial loop) !!

We need a way to save the previous result and slow down the counter.

Digital Design – 8 bits counter

Design of an 8 bits counter



We need a way to save the previous result (a **register**) and slow down the counter (a **clock**).

Digital Design – Base elements

Hardware consist of a few simple building blocks

1. Combinatorial

- ▶ “Instant” state changes, e.g.:

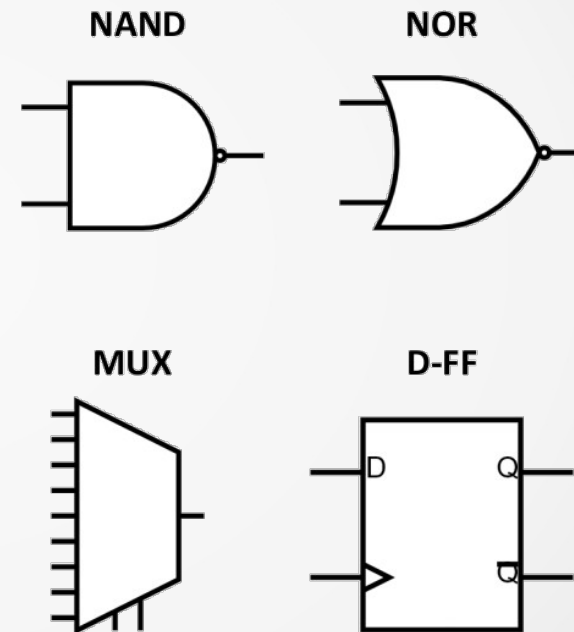
Classical gates (especially NAND & NOR)

Multiplexer (MUX)

2. Synchronous

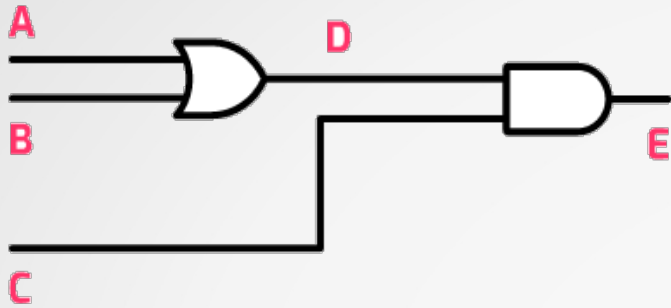
- ▶ “Clocked” state changes, e.g.:

Flip Flop (e.g. D-FF, register)



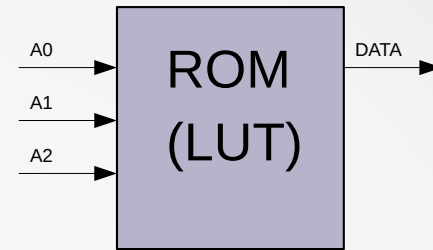
A Scientist's Guide to FPGAs – Alexander Ruede – ICSC 2019

Digital Design – LUT



A	B	C	E
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

=



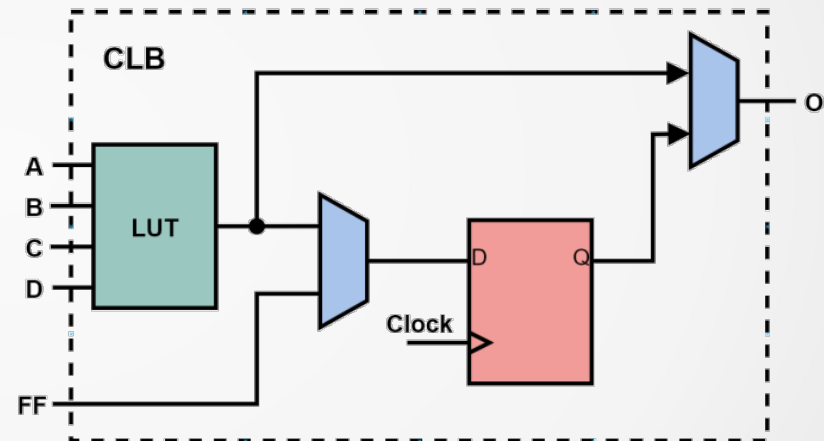
ADDR	DATA
000	0
001	0
010	0
011	1
100	0
101	1
110	0
111	1

Anatomy of FPGAs - CLB

FPGA are made of Configurable Logic Block (CLB)

(Also “logic cell” or “logic element”)

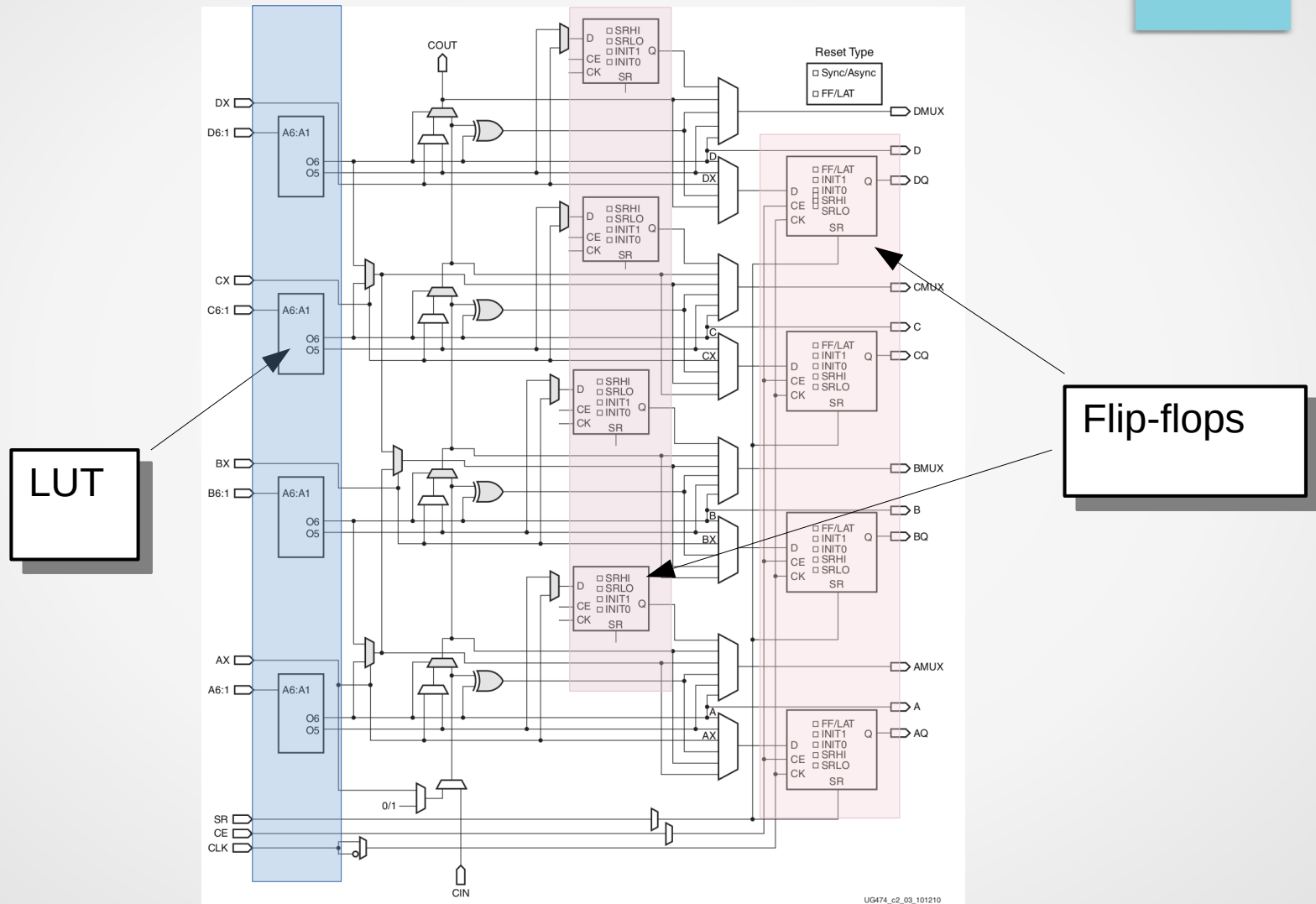
- ▶ LUT configuration is flexible
- ▶ D-type flip-flops configuration is flexible
- ▶ Flip-flops can take input from outside the CLB or from the LUT



Simplified example CLB with one 4-input LUT and one flip-flop

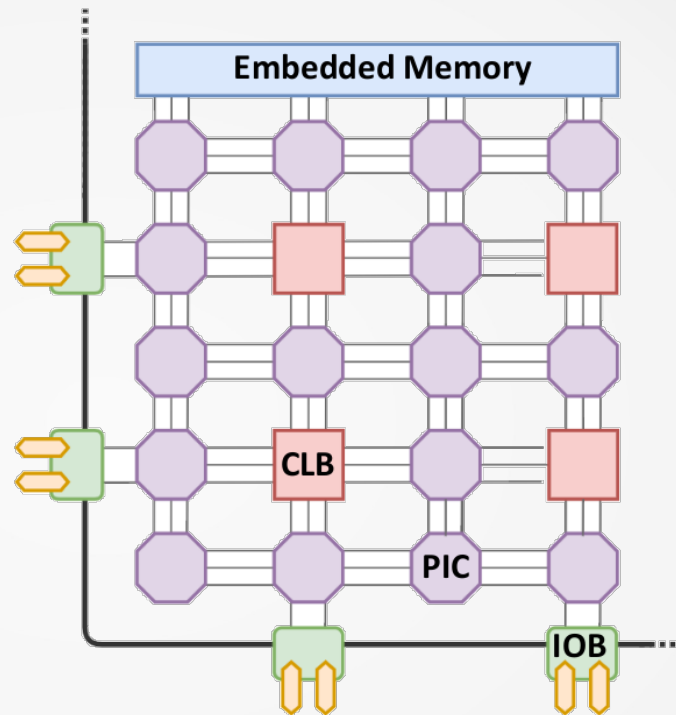
A Scientist's Guide to FPGAs – Alexander Ruede – ICSC 2019

Anatomy of FPGAs - SLICE example (Xilinx)



Anatomy of FPGAs - Matrix

- ▶ CLB: Configurable Logic Block
- ▶ PIC: Programmable Interconnect
- ▶ IOB: Input-Output Block [1]
- ▶ Clock Management [2]
- ▶ Hardened Cores



A Scientist's Guide to FPGAs – Alexander Ruede – iCSC 2019

- ▶ Programming a FPGA is configuring its interconnection matrix and basic blocks (IOB, CLB,...)

[1] [ug471_7series_SelectIO.pdf](#)

[2] [ug472_7series_Clocking.pdf](#)

Anatomy of FPGAs - Hardened cores

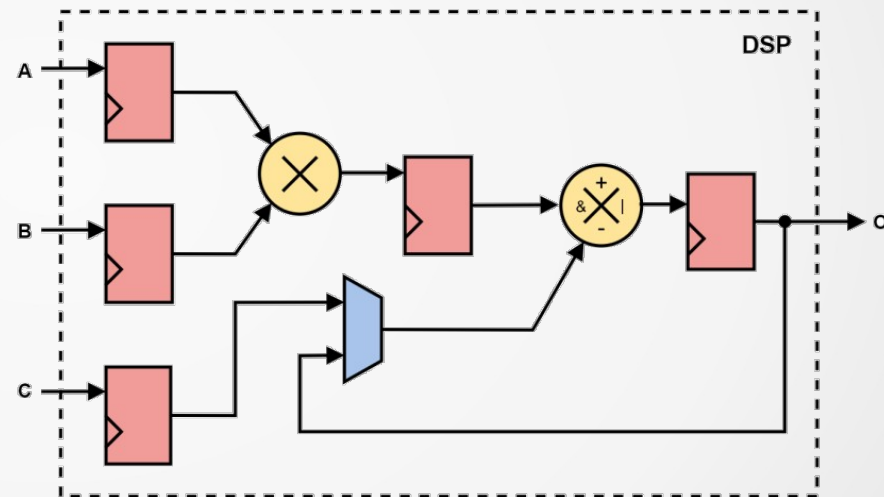
Hardened Cores (Also called “IP cores”)

Specialized tasks (e.g. multiplication) take up a lot of logic cells

Hardened cores in silicon for more effective use of resources

Typical cores found in modern FPGAs:

- ▶ Memory (Block RAM [1])
- ▶ DSP blocks
- ▶ Clocking (Programmable PLL)
- ▶ Communication interfaces (e.g. PCIe)
- ▶ Serializer/Deserializer (SerDes)
- ▶ CPU



Exemplary DSP block with multiplier, accumulator and pipeline stages

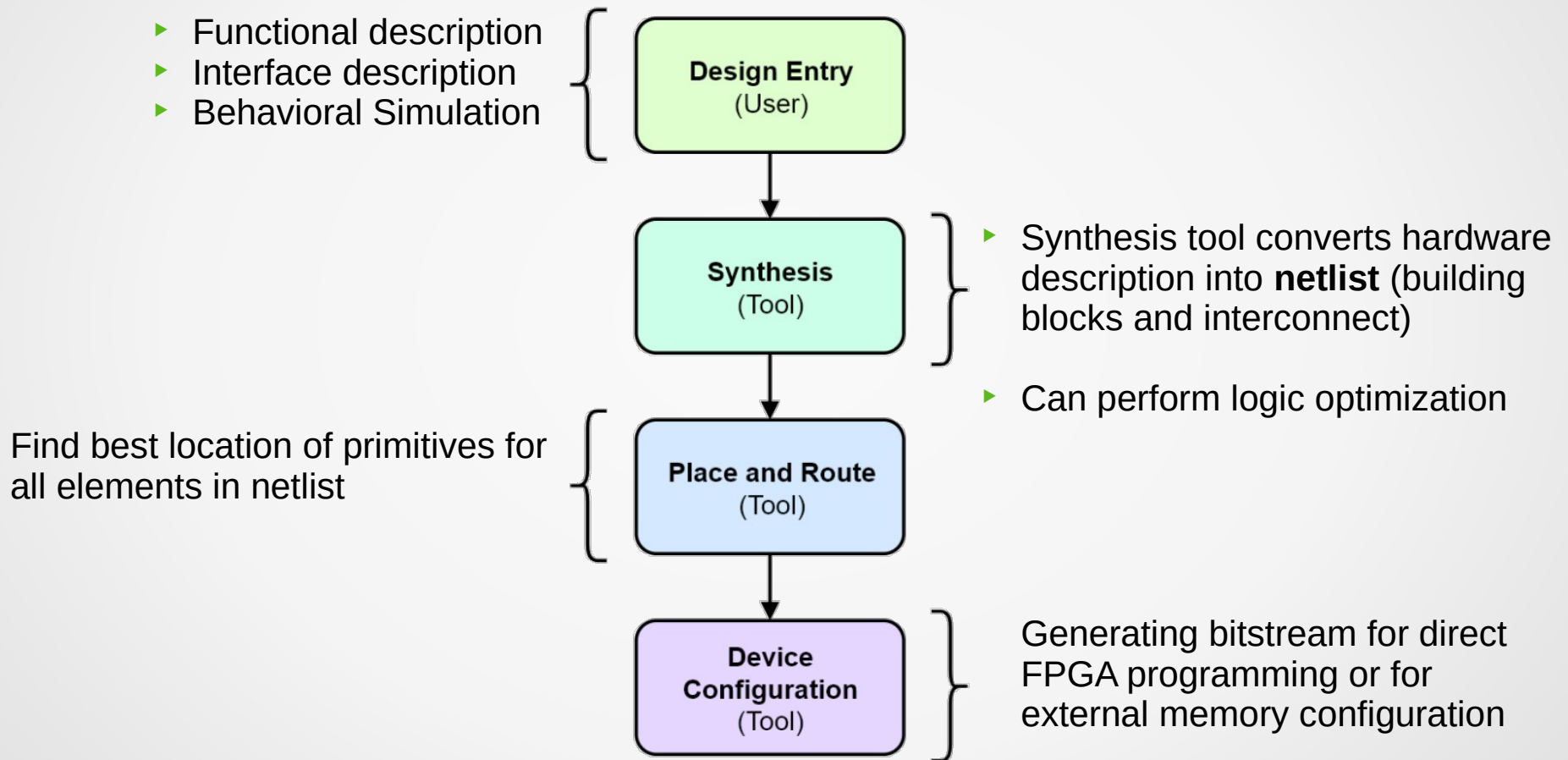
A Scientist's Guide to FPGAs – Alexander Ruede – ICSC 2019

[1] [ug473_7Series_Memory_Resources.pdf](#)

Anatomy of FPGAs - Classical Design Flow

- ▶ Create an FPGA design is:
 - Describing the interface between the FPGA and the electronic board
→ Configuration of **IOB**
 - Describing the modules (Adder, Multiplier, CPU, FFT, etc...) and how to connect them together.
 - Transforming this description (**RTL**) in a machine description called **bitstream** (LUT's configuration, Interconnection Matrix's configuration, etc...)
→ Configuration of **LUTs, PIC**

Anatomy of FPGAs - Classical Design Flow



A Scientist's Guide to FPGAs – Alexander Ruede – iCSC 2019

Anatomy of FPGAs - Classical Design Flow

```
def do_finalize(self, fragment):
    GowinPlatform.do_finalize(self, fragment)
    self.add_period_constraint(self.lookup_request("sys clk")

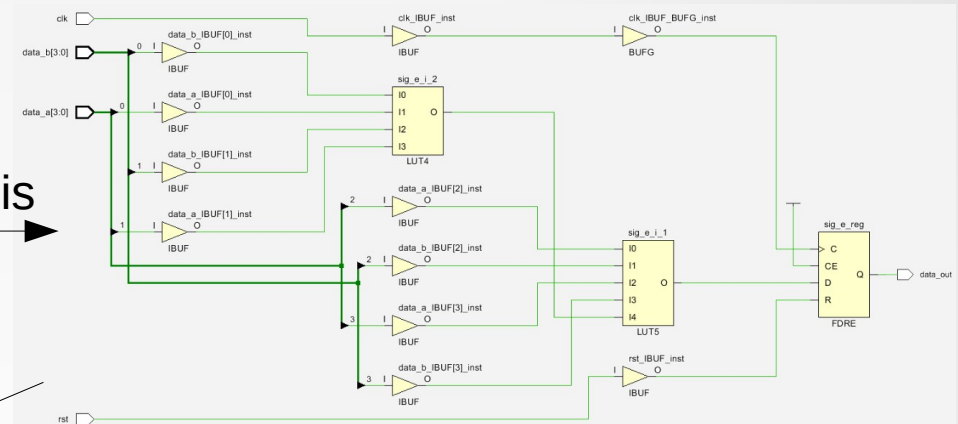
# Design -----

platform = Platform()
led = platform.request("user led", 1)
rst = platform.request("user btn", 0)
clk = platform.request("sys clk")

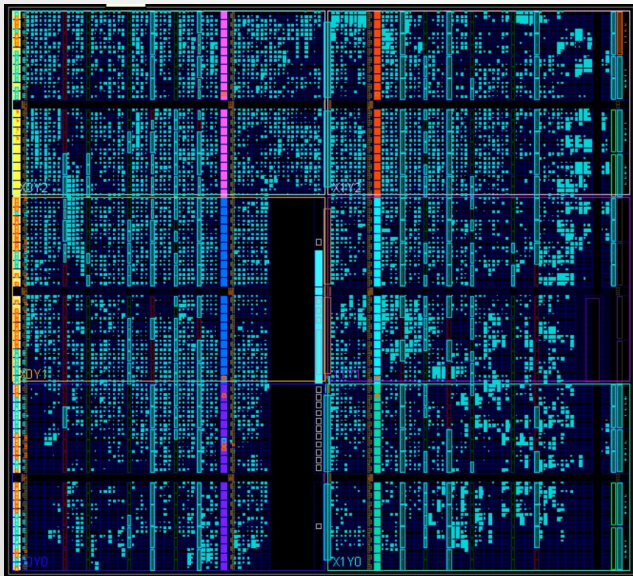
# Create our module (fpga description)
module = Module()
module.clock_domains.cd_sys = ClockDomain("cd sys")

module.comb += module.cd_sys.clk.eq(clk)
#module.comb += module.cd_sys.rst.eq(rst)
```

synthesis



P&R



Bitstream

Anatomy of FPGAs - What we've learnt

- ▶ FPGA are made of configurable logic blocks and dedicated blocks surrounded by I/Os, interconnected by a switch matrix
 - ▶ Programming the FPGA is basically writing values into LUTs and configuring the interconnection matrix
 - ▶ The hardware description is translated into a netlist by the synthesizer
 - ▶ The P&R finds the best locations for the primitives and interconnects the components
- ▶ Software / CPU specify a **sequence** of instructions
 - ▶ HDL / FGPA **describe** structure and behavior of digital components

Anatomy of FPGAs - FPGA vs CPLD

FPGA

- ▶ Configuration is volatile. Bitstream is stored in an external memory (SPI flash) and loaded.
→ Delay of several milliseconds at power ON.
- ▶ Variety of on-die dedicated hardware such as Block RAM, DSP blocks, PLL, DCMs, Memory Controllers, Multi-Gigabit Transceivers
- ▶ PCB cost much higher (BGA, multiple voltage rails, external SPI flash)

CPLD

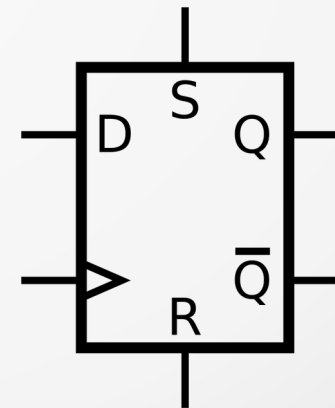
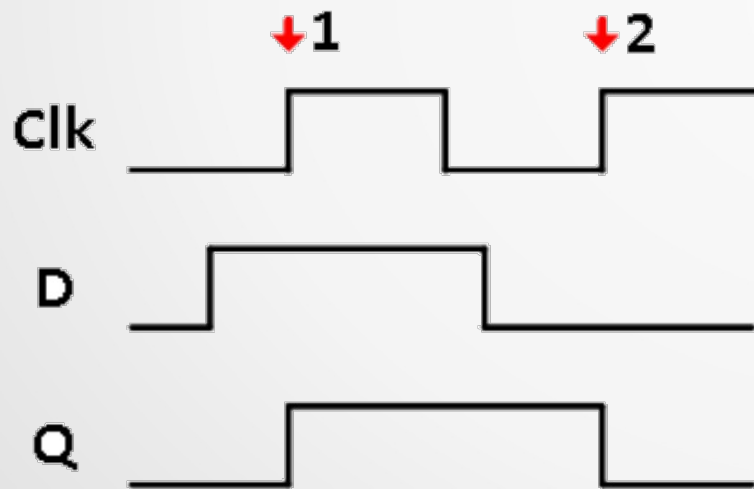
- ▶ Bitstream is stored in flash memory.
→ Instant ON
- ▶ Very small amount of logic resources
- ▶ No on-die hard IPs available (RAM, PLL,...)
- ▶ Only one voltage rail
- ▶ Available in TQFP package

Agenda

- ▶ Description of FPGAs
- ▶ **Digital design challenges**
- ▶ Migen: introduction and workshops
- ▶ LiteX: introduction and workshops
- ▶ LiteX: advanced topics

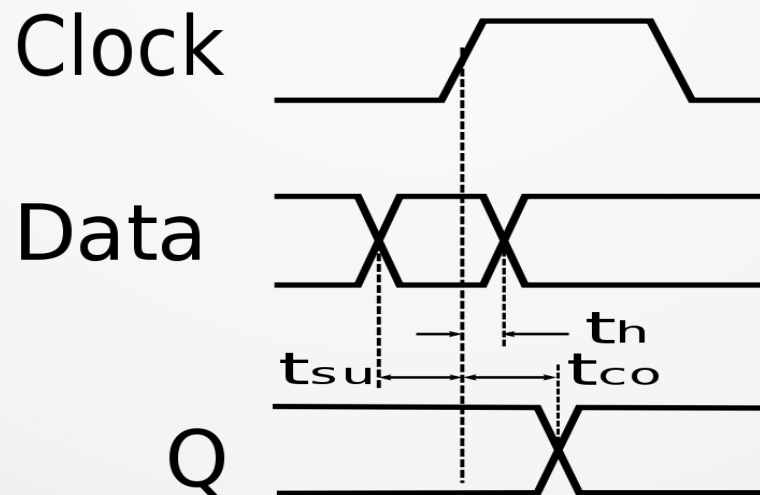
Flip-Flops - Description

- ▶ There are a few different types of flip-flops (JK, T, D) but the one that is used most frequently is the D Flip-Flop.
- ▶ Sequential logic operates on the transitions of a clock. When a Flip-Flop sees a rising edge of the clock, it **registers** (copy and hold) the data from the Input D to the Output Q.
- ▶ Flip-flops are the main components in an FPGA that are used to keep the state inside of the chip.



Flip-Flops – Timing considerations

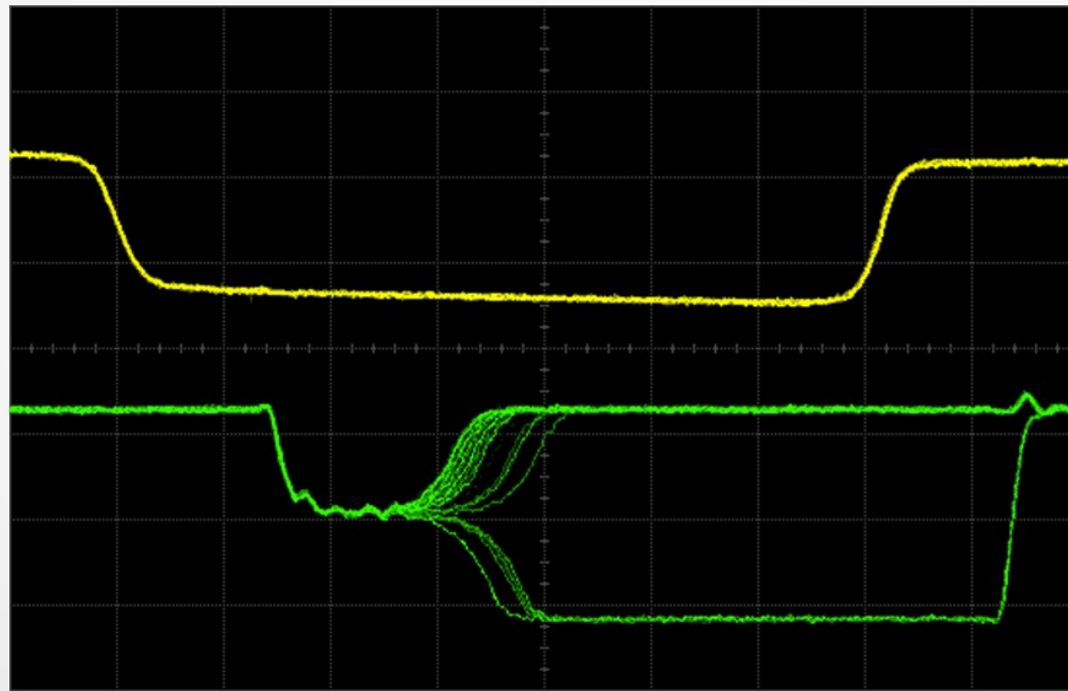
- ▶ Because of the construction of a flip-flop [1], the input must be held steady in a period around the rising edge of the clock.
- ▶ **Setup time** is the minimum amount of time the data input should be held steady before the clock event, so that the data is reliably sampled by the clock.
- ▶ **Hold time** is the minimum amount of time the data input should be held steady after the clock event, so that the data is reliably sampled by the clock.



[1] <https://www.edn.com/understanding-the-basics-of-setup-and-hold-time/>

Flip-Flops – Metastability

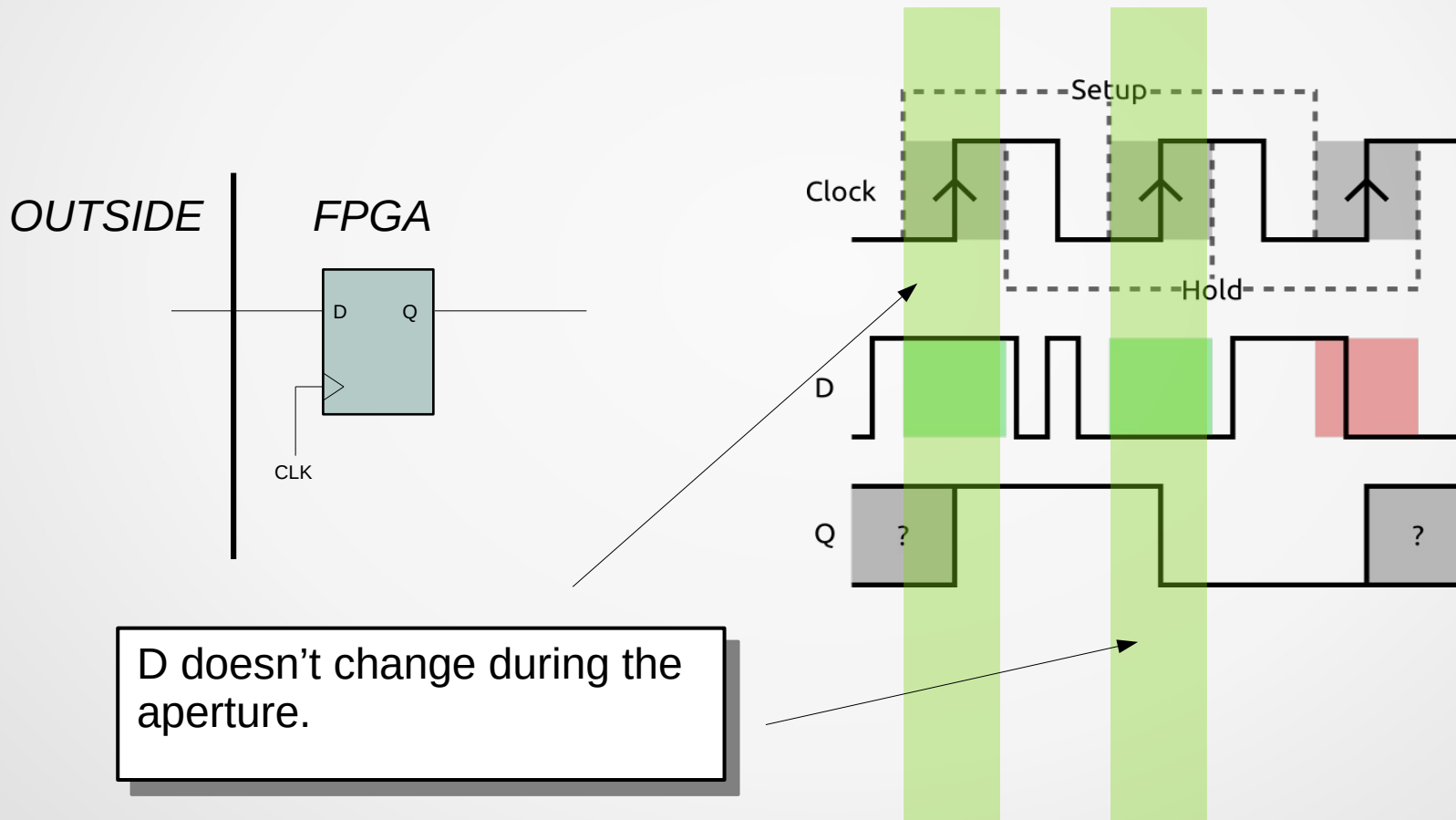
- ▶ If setup and hold time are not respected, flip-flops are subject to a problem called **metastability**
- ▶ The result is that the output may behave unpredictably, taking many times longer than normal to settle to one state or the other, or even oscillating several times before settling.



<https://youtu.be/5PRuPVIjEcs>

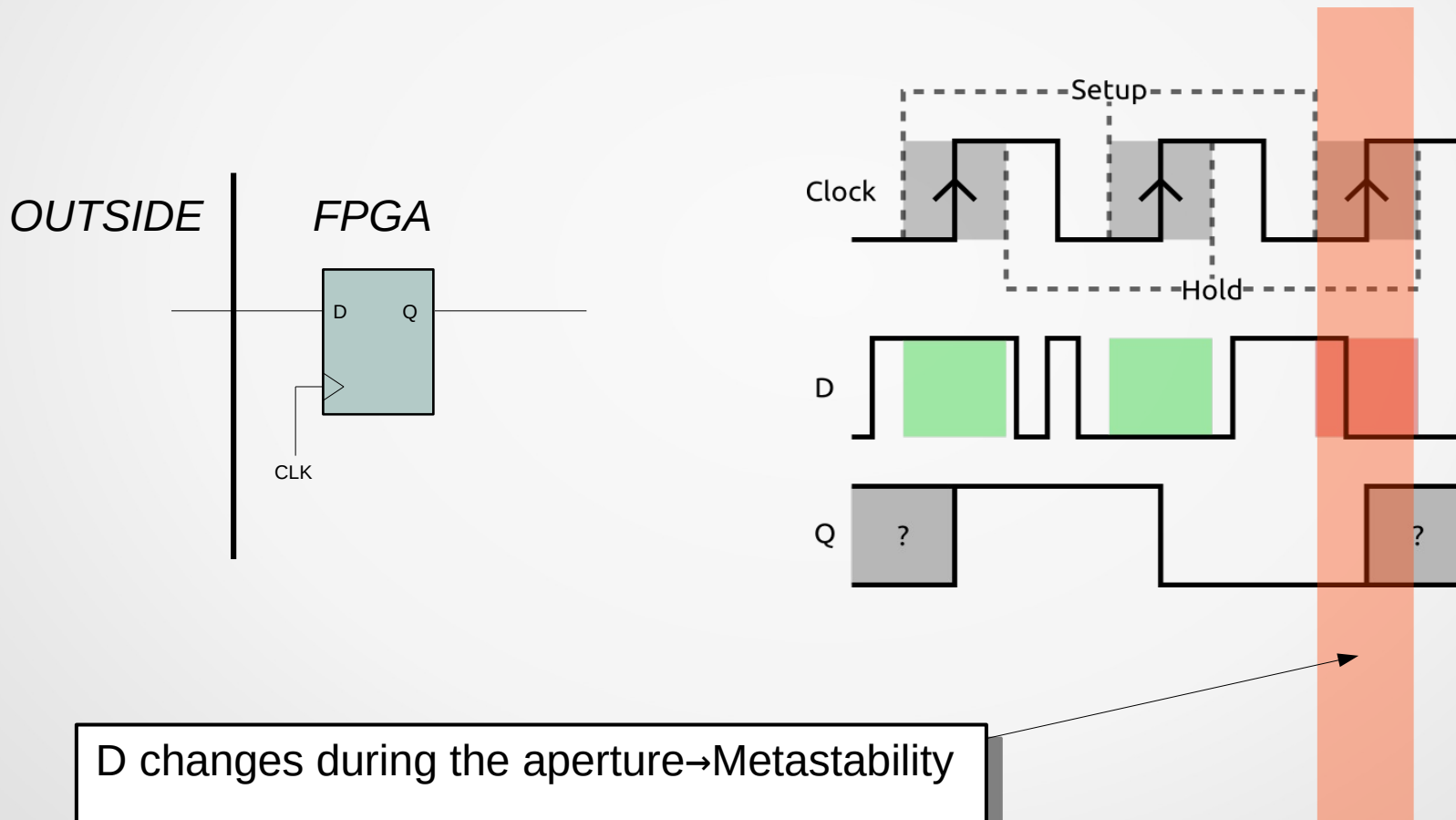
Three main reasons for metastability problem (1/3)

- An external signal (user input) is read inside the FPGA:



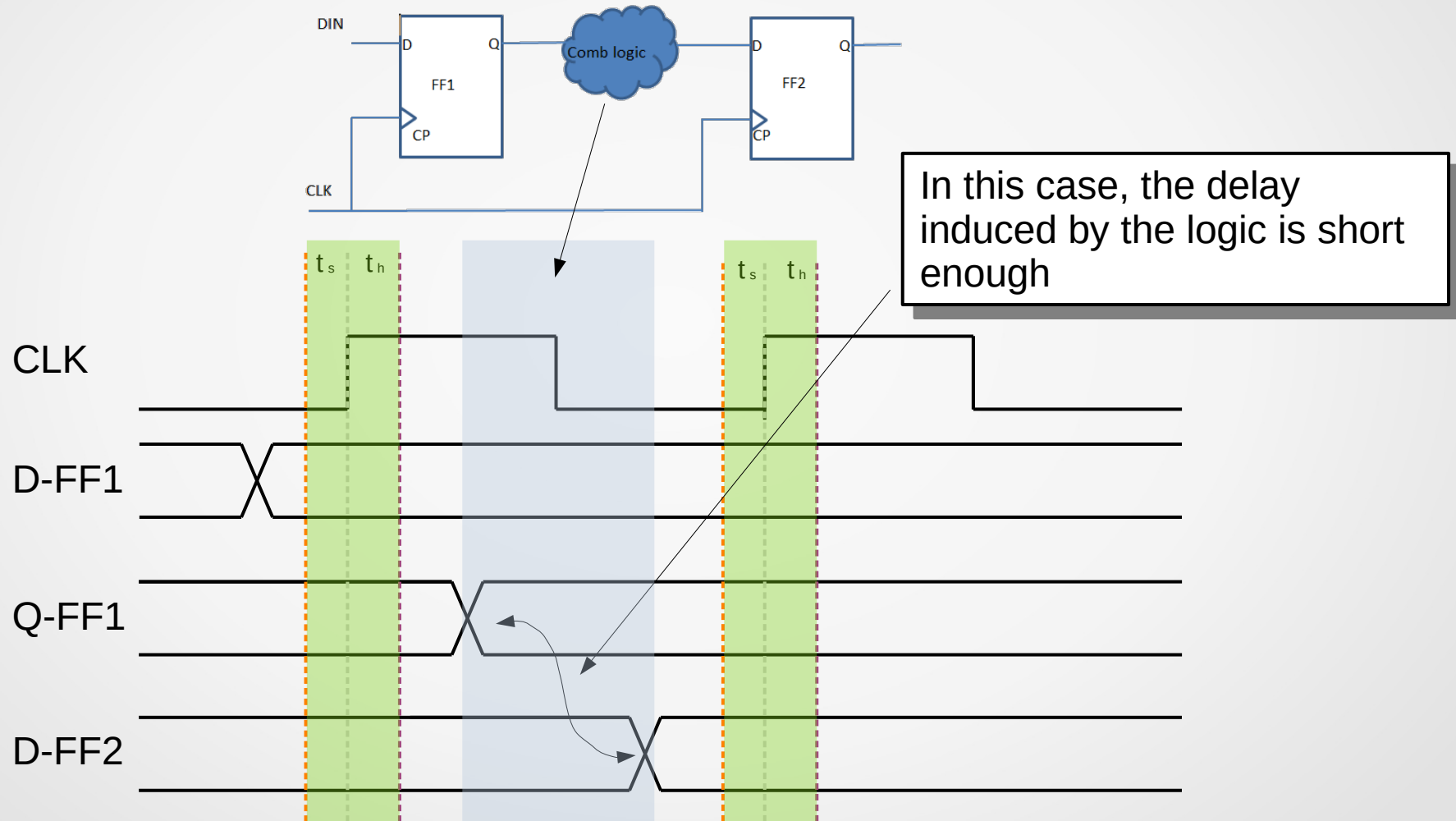
Three main reasons for metastability problem (1/3)

- An external signal (user input) is read inside the FPGA:



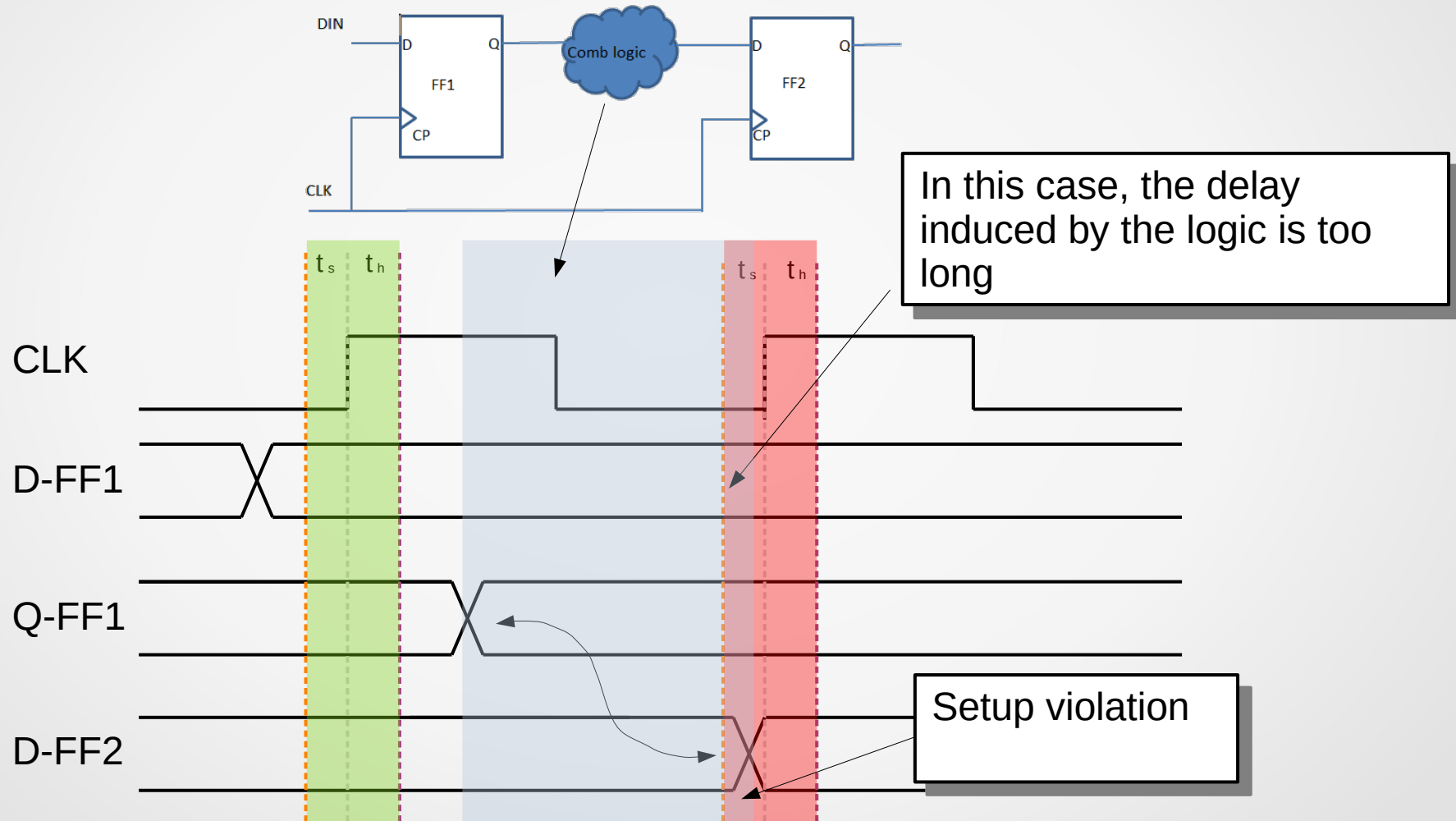
Three main reasons for metastability problem (2/3)

- ▶ Too much logic (delay) between flip-flops (setup violation):



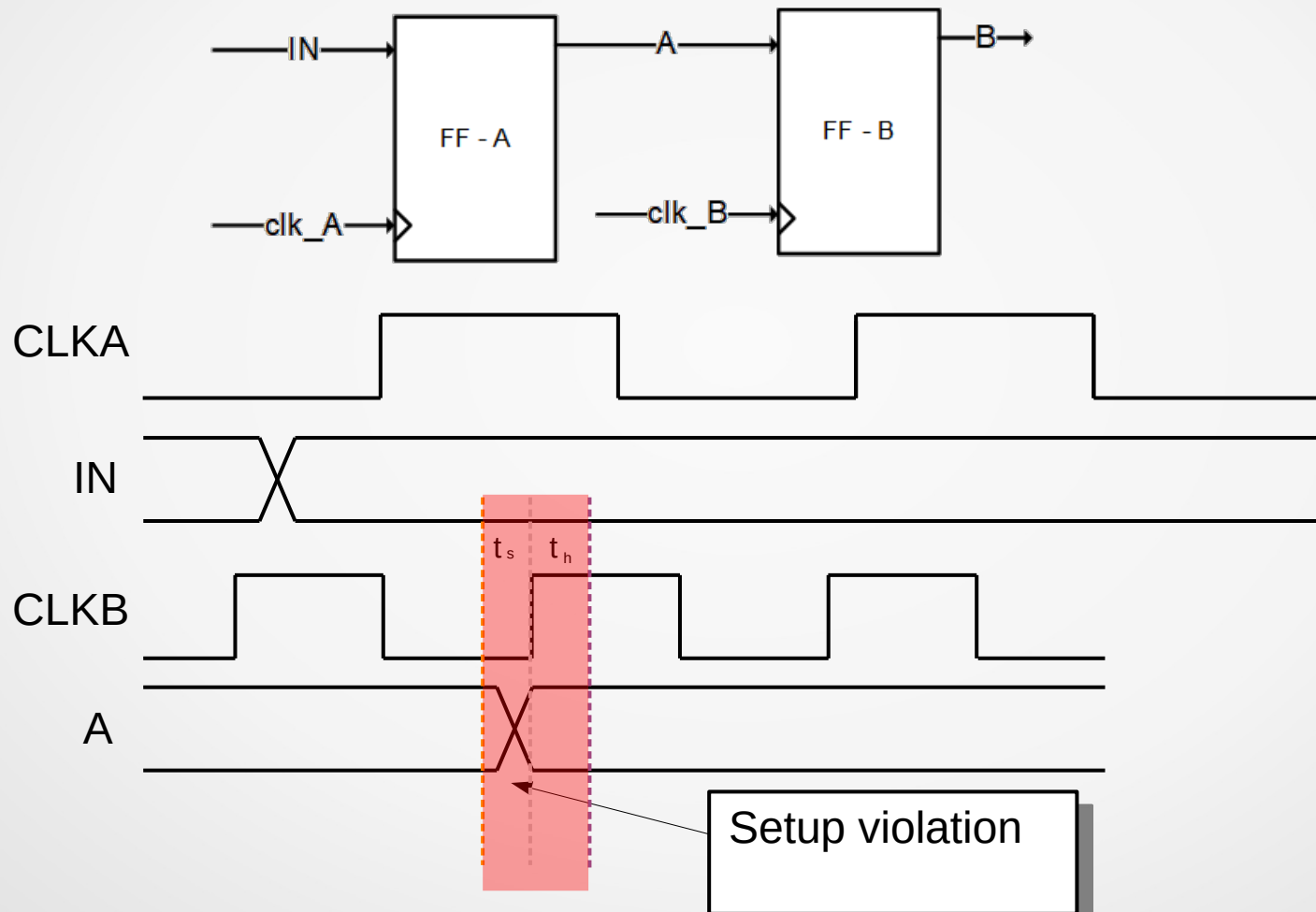
Three main reasons for metastability problem (2/3)

- Too much logic (delay) between flip-flops (setup violation):



Three main reasons for metastability problem (3/3)

- Multiple clock domains



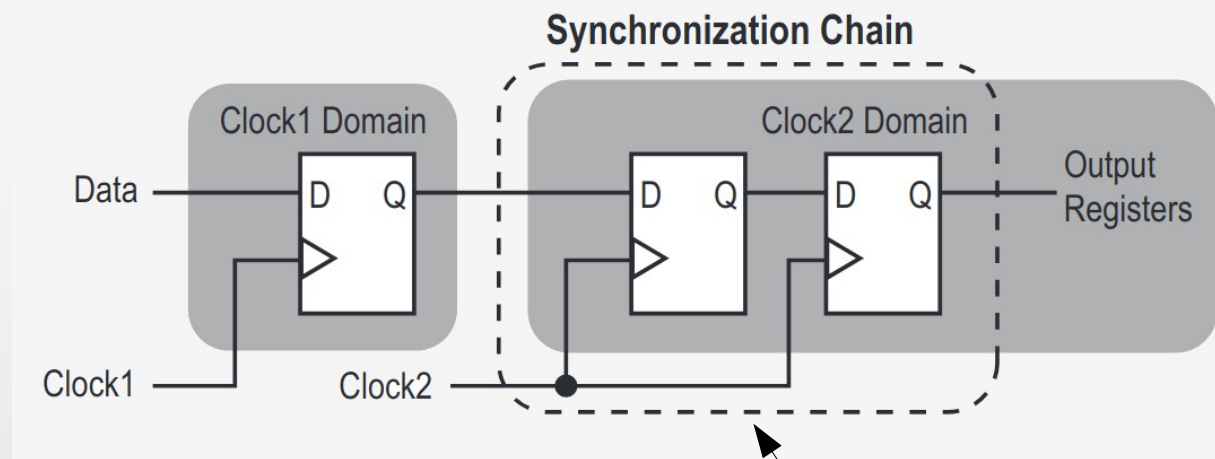
Because `clk_A` and `clk_B` are asynchronous, A can change anytime with regards to `clk_B` rising edge.

Who is responsible ?

- ▶ You are responsible for this. Designers must prevent **timing** problems:
 - External asynchronous signals must be handled properly with **synchronizers**,
 - when using multiple clock domains, use proper **clock domain crossing** (CDC) circuits,
 - look at **static timing analysis** report from your synthesis tool and take care (at least evaluate) of every (most) warnings.

Synchronizer

- Use a sequence of registers in the destination clock domain to resynchronize the signal to the new clock domain.
- Allows additional time for a potentially metastable signal to resolve to a known value before the signal is used in the rest of the design.



https://trilobyte.com/pdf/golson_snug14.pdf

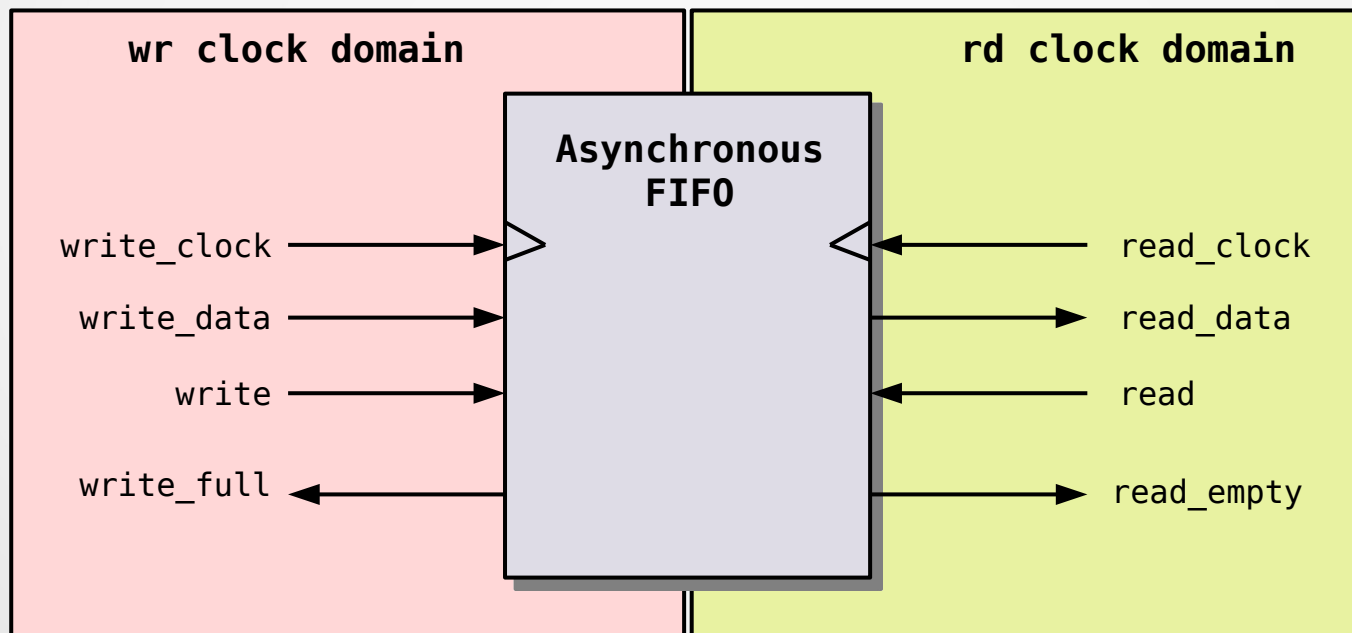
Must be kept close each other

Clock Domain Crossing

- ▶ Used when transferring data (single signals or busses) across clock domain boundaries.
- Use register based synchronizers
- FIFO based data synchronizers

FIFO based data synchronizers

- Data is pushed into the FIFO with transmitter clock and pulled out from FIFO with receiver clock.



Static Timing Analysis

- ▶ Performed by the implementation tool
- ▶ Needs constraints (SDC files)
- ▶ Verify every path and detect potential failures at every corners
- ▶ Gives Fmax

```
# Constrain clock port clk with a 10-ns requirement
create clock -period 10 [get ports clk]

# Set a false-path between two unrelated clocks
set false path -from [get clocks clk] -to [get clocks clkA]
```

Static Timing Analysis

- ▶ Performed by the implementation tool
- ▶ Needs constraints (SDC files)
- ▶ Verify every path and detect potential failures at every corners
- ▶ Gives Fmax

Maximum possible analyzed clocks frequency

Clock Name	Period (ns)	Frequency (MHz)	Edge
pll0_clkout0	9.806	101.978	(R-R)

Geomean max period: 9.806

Launch Clock	Capture Clock	Constraint (ns)	Slack (ns)	Edge
pll0_clkout0	pll0_clkout0	10.000	0.194	(R-R)

Static Timing Analysis

Maximum possible analyzed clocks frequency

Clock Name	Period (ns)	Frequency (MHz)	Edge
axi_clk	15.987	62.551	(R-R)
mipi_pclk	7.077	141.293	(R-R)
px_clk	12.711	78.671	(R-R)

Geomean max period: 11.288

Setup (Max) Clock Relationship

Launch Clock	Capture Clock	Constraint (ns)	Slack (ns)	Edge
axi_clk	axi_clk	12.500	-3.487	(R-R)
mipi_pclk	mipi_pclk	20.000	12.923	(R-R)
px_clk	px_clk	13.468	0.757	(R-R)

Hold (Min) Clock Relationship

Launch Clock	Capture Clock	Constraint (ns)	Slack (ns)	Edge
axi_clk	axi_clk	0.000	0.086	(R-R)
mipi_pclk	mipi_pclk	0.000	0.184	(R-R)
px_clk	px_clk	0.000	0.307	(R-R)

Static Timing Analysis

```
++++ Path 1 ++++++
Path Begin      : main_videocapture_gain_output_x[2]~FF|CLK
Path End       : main_videocapture_rawtorgb_r_bayer_11_01_0P_0[6]~FF|D
Launch Clock    : axi_clk (RISE)
Capture Clock   : axi_clk (RISE)
Slack           : -3.487 (required time - arrival time)
Delay           : 15.468

Logic Level : 8
```

Agenda

- ▶ Description of FPGAs
- ▶ Digital design challenges
- ▶ **Migen: introduction and workshops**
- ▶ LiteX: introduction and workshops

Agenda

- ▶ Migen: introduction and workshops
 - Concepts, Modules and signals
 - Blinker example
 - Attributes of Module()
 - Example of verilog output
 - Operators (If/Else and FSM)
 - Minimum project requirement (Migen/LiteX)
 - Workshop 1/2
 - Records
 - Simulation
 - Workshop 2/2

What is Migen

- ▶ An alternative HDL based on Python
- ▶ Generates a unique Verilog file from a Python description
- ▶ Migen has an **integrated simulator** that allows test benches to be written in Python
- ▶ Other alternative HDL exist:
Amaranth (nMigen), SpinalHDL, Chisel, PipelineC, Silice,...

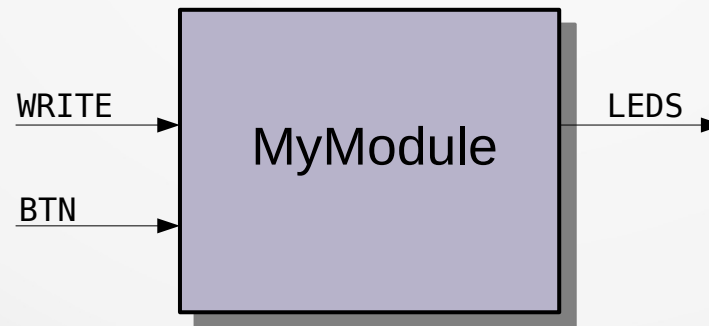
<https://m-labs.hk/gateway/migen>

<http://www.fabienm.eu/flf/hdl/>



Migen – Concepts

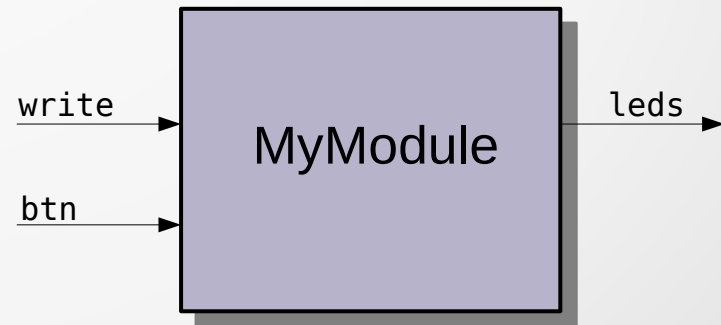
- ▶ Migen uses Python classes
- ▶ The most important class is **Module**
- ▶ A module has input / output signals and parameters
- ▶ The direction of signals in the interface is not explicit



Migen – Concepts

- ▶ Interfaces of modules are defined by attributes
- ▶ All **attributes** with the type **Signal()** are considered interfaces of the module
- ▶ In our case:

```
class MyModule(Module):  
    def __init__(self):  
        self.leds = Signal()  
        self.btn = Signal()  
        self.write = Signal()
```



Migen – Concepts

- ▶ Every signal assignment is either:
 - **combinatorial** (continuous assignments)
 - **synchronous** (at the edge of the clock signal)
- ▶ Module() has a ***sync*** and a ***comb*** attributes (lists)
- ▶ Assignment are added to the chosen type using the in-place addition operation (***+=***)

Migen – Concepts

```
class MyModule(Module):
    def __init__(self):
        self.out0 = Signal()
        self.out1 = Signal()
        self.out2 = Signal()
        self.write = Signal()

    ###

    self.comb += self.out0.eq(0)
    self.comb += [
        self.out1.eq(1),
        self.out2.eq(~self.write)
    ]
```

Migen – Signal

- ▶ Signal object represents a value that is expected to change in a circuit. It does exactly what Verilog's "wire" and "reg" and VHDL's "signal" do.
- ▶ They are assigned using the **eq()** method

```
a = Signal()  
b = Signal(2)  
c = Signal(max=23)  
d = Signal(reset=1)  
e = Signal(4, reset_less=True)  
f = Signal.like(c)  
g = f.nbits
```

```
self.sync += s1.eq(1)
```

Agenda

- ▶ Migen: introduction and workshops
 - Concepts, Modules and signals
 - **Blinker example**
 - Attributes of Module()
 - Example of verilog output
 - Operators (If/Else and FSM)
 - Minimum project requirement (Migen/LiteX)
 - Workshop 1/2
 - Records
 - Simulation
 - Workshop 2/2

Migen – Blinker module

- ▶ Functional block with input and outputs
- ▶ Signals of the interface are attributes of the class
- ▶ A module has important attributes (comb, sync,...)
- ▶ As any other Python class, parameters can be passed to modules



```
class Blink(Module):  
    def __init__(self, bit):  
        self.led = Signal()  
        self.write = Signal()  
        self.btn = Signal(5)  
  
        ###  
  
        counter = Signal(25)  
        self.comb += self.led.eq(counter[bit])  
        self.sync += counter.eq(counter + 1)
```

Migen – Blinker module

- ▶ Functional block with input and outputs
- ▶ Signals of the interface are attributes of the class
- ▶ A module has important attributes (comb, sync,...)
- ▶ As any other Python class, parameters can be passed to modules



```
class Blink(Module):  
    def __init__(self, bit):  
        self.led = Signal()  
        self.write = Signal()  
        self.btn = Signal(5)  
  
        ###  
  
        counter = Signal(25)  
        self.comb += self.led.eq(counter[bit])  
        self.sync += counter.eq(counter + 1)
```


Migen – Blinker module

- ▶ Functional block with input and outputs
- ▶ Signals of the interface are attributes of the class
- ▶ A module has important attributes (comb, sync,...)
- ▶ As any other Python class, parameters can be passed to modules



```
class Blink(Module):  
    def __init__(self, bit):  
        self.led = Signal()  
        self.write = Signal()  
        self.btn = Signal(5)  
  
        ###  
  
        counter = Signal(25)  
        self.comb += self.led.eq(counter[bit])  
        self.sync += counter.eq(counter + 1)
```

Migen – Blinker module

- ▶ Functional block with input and outputs
- ▶ Signals of the interface are attributes of the class
- ▶ A module has important attributes (comb, sync,...)
- ▶ As any other Python class, parameters can be passed to modules



```
class Blink(Module):  
    def __init__(self, bit):  
        self.led = Signal()  
        self.write = Signal()  
        self.btn = Signal(5)  
  
        ###  
  
        counter = Signal(25)  
        self.comb += self.led.eq(counter[bit])  
        self.sync += counter.eq(counter + 1)
```

Agenda

- ▶ Migen: introduction and workshops
 - Concepts, Modules and signals
 - Blinker example
 - Attributes of Module()
 - Example of verilog output
 - Operators (If/Else and FSM)
 - Minimum project requirement (Migen/LiteX)
 - Workshop 1/2
 - Records
 - Simulation
 - Workshop 2/2

Migen – Attributes of Modules

- ▶ ***comb*** → a list of combinatorial assignments
- ▶ ***sync*** → a list of synchronous assignments
- ▶ ***submodules*** → a list of modules used by this module
- ▶ ***specials*** → a list of Platform specific modules, Verilog instances, memories,...

- ▶ ***clock_domains*** → clock domains used by this module

Migen – Attributes of Modules: comb

- **comb** → a list of combinatorial assignments

```
class M1(Module):  
    def __init__(self):  
        # Interfaces  
        self.leds = Signal()  
        self.btn = Signal()  
        self.write = Signal()  
  
        ###  
  
        s = Signal()  
  
        # Functional description. This is what this  
        # module does.  
        self.comb += [  
            self.leds.eq(self.btn & self.write),  
            s.eq(~self.btn),  
        ]
```

Migen – Attributes of Modules: sync

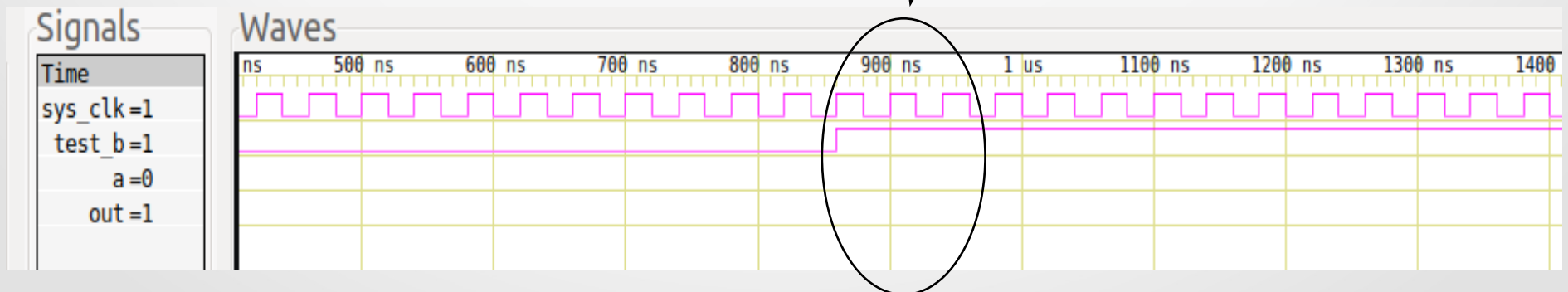
- **sync** → a list of synchronous assignments

```
class M1(Module):  
    def __init__(self):  
        self.test_b = test_b = Signal()  
        self.out      = out      = Signal()  
  
        a = Signal(reset=1)  
  
        self.sync += [  
            If(test_b,  
                a.eq(0)  
            ),  
            If(a == 0,  
                out.eq(1)  
            )  
        ]
```

Migen – Attributes of Modules: sync

- **sync** → a list of synchronous assignments

```
class M1(Module):  
    def __init__(self):  
        self.test_b = test_b = Signal()  
        self.out = out = Signal()  
  
        a = Signal(reset=1)  
  
        self.sync += [  
            If(test_b,  
                a.eq(0)  
            ),  
            If(a == 0,  
                out.eq(1)  
            )  
        ]
```



Migen – Attributes of Modules: sync

- **sync** → a list of synchronous assignments

```
class M1(Module):  
    def __init__(self):  
        self.test_b = test_b = Signal()  
        self.out = out = Signal()  
  
        a = Signal(reset=1)  
  
        self.sync += [  
            If(test_b,  
                a.eq(0)  
            ),  
            If(a == 0,  
                out.eq(1)  
            )  
        ]
```

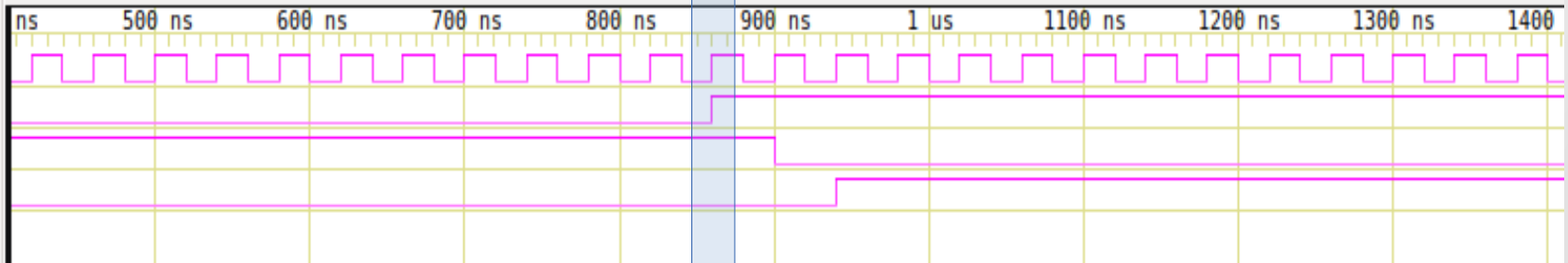
test_b == 0, set a to 0



Signals

Time
sys_clk=1
test_b=1
a=0
out=1

Waves



Migen – Attributes of Modules: sync

- **sync** → a list of synchronous assignments

```
class M1(Module):  
    def __init__(self):  
        self.test_b = test_b = Signal()  
        self.out = out = Signal()  
  
        a = Signal(reset=1)  
  
        self.sync += [  
            If(test_b,  
                a.eq(0)  
            ),  
            If(a == 0,  
                out.eq(1)  
            )  
        ]
```

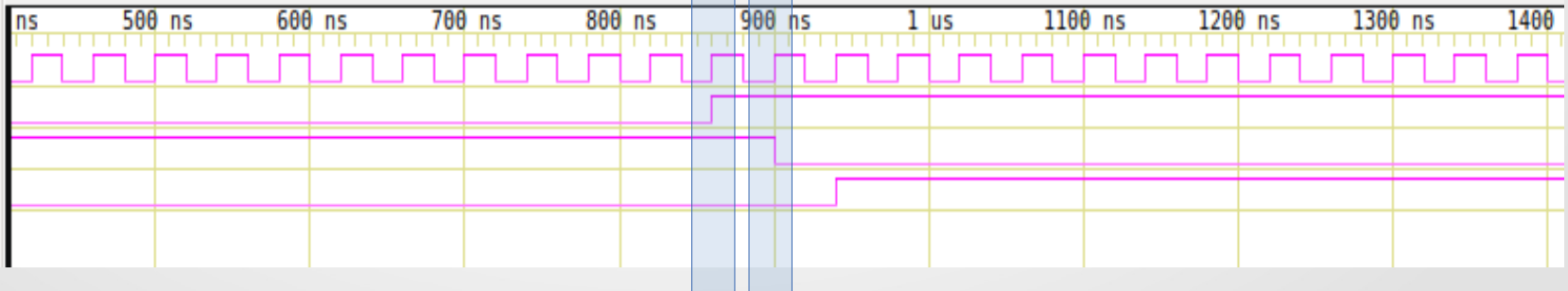
test_b == 0, set a to 0

a is now 0, a == 0 is true
so set out to 1

Signals

Time
sys_clk=1
test_b=1
a=0
out=1

Waves



Migen – Attributes of Modules: sync

- **sync** → a list of synchronous assignments

```
class M1(Module):  
    def __init__(self):  
        self.test_b = test_b = Signal()  
        self.out = out = Signal()  
  
        a = Signal(reset=1)  
  
        self.sync += [  
            If(test_b,  
                a.eq(0)  
            ),  
            If(a == 0,  
                out.eq(1)  
            )  
        ]
```

test_b == 0, set a to 0

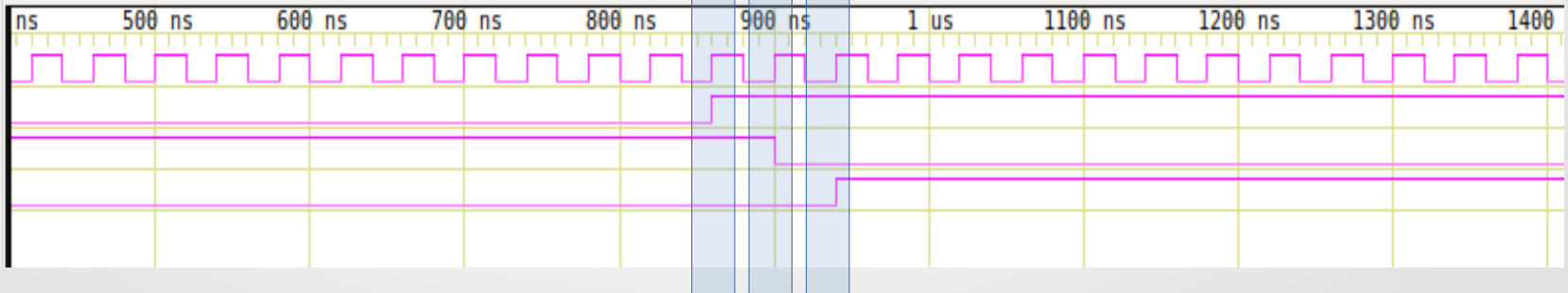
a is now 0, a == 0 is true
so set out to 1

Finally, out is now equal to 1

Signals

Time
sys_clk=1
test_b=1
a=0
out=1

Waves



Migen – Attributes of Modules: submodules

- **submodules** → a list of modules used by this module

```
class M1(Module):
    def __init__(self):
        # Interfaces
        self.leds = Signal()
        self.btn = Signal()
        self.write = Signal()

        ###

        btn_sync = Signal()

        # We want to use an instance of M2 in this module.
        m2 = M2()
        self.submodules += m2

        self.comb += [
            self.leds.eq(self.btn & self.write),
            # Here we use m2
            s.eq(m2.test_out),
        ]
```

We have access to the interface of m2

- Can be named (`self.submodules.m2 = m2`)

Migen – Attributes of Modules: special

- **special** → a list of Platform specific modules, Verilog instances, memories,...

```
class M1(Module):
    def __init__(self):
        self.leds = Signal()
        self.btn = Signal()
        self.write = Signal()

        ###

        btn_sync = Signal()

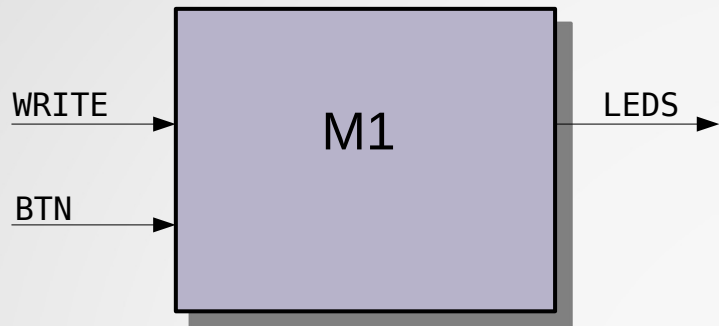
        # Multireg is a synchroniser that is defined for each
        # device/vendor
        self.specials += MultiReg(self.btn, btn_sync)

        self.sync += [
            self.leds.eq(btn_sync & self.write),
        ]
```

Agenda

- ▶ Migen: introduction and workshops
 - Concepts, Modules and signals
 - Blinker example
 - Attributes of Module()
 - Example of verilog output
 - Operators (If/Else and FSM)
 - Minimum project requirement (Migen/LiteX)
 - Workshop 1/2
 - Records
 - Simulation
 - Workshop 2/2

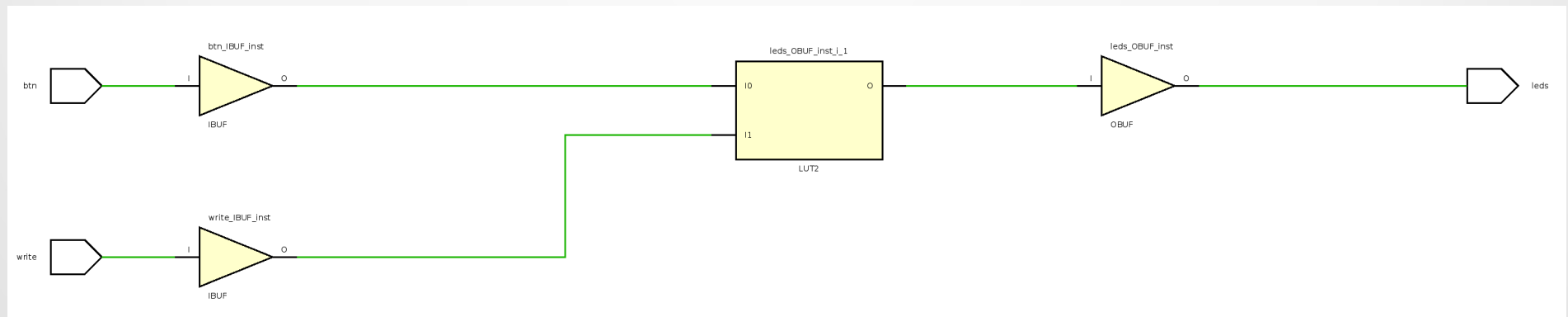
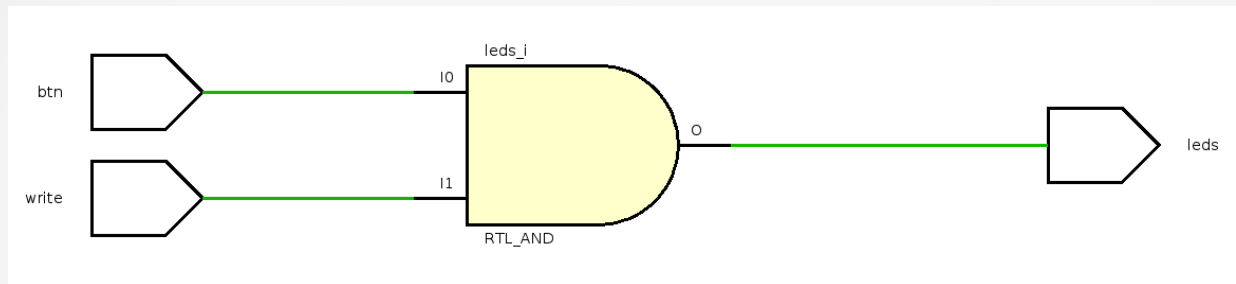
Migen – Combinatorial example



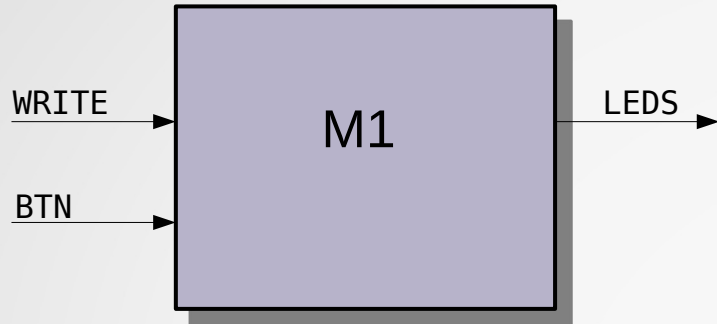
```
class M1(Module):  
    def __init__(self):  
        # Interfaces  
        self.leds = Signal()  
        self.btn = Signal()  
        self.write = Signal()  
  
        ###  
  
        # Functional description. This is what this  
        # module does.  
        self.comb += self.leds.eq(self.btn & self.write)
```

```
module top(  
    output leds,  
    input btn,  
    input write  
);  
  
assign leds = (btn & write);  
  
endmodule
```

Migen – Combinatorial example



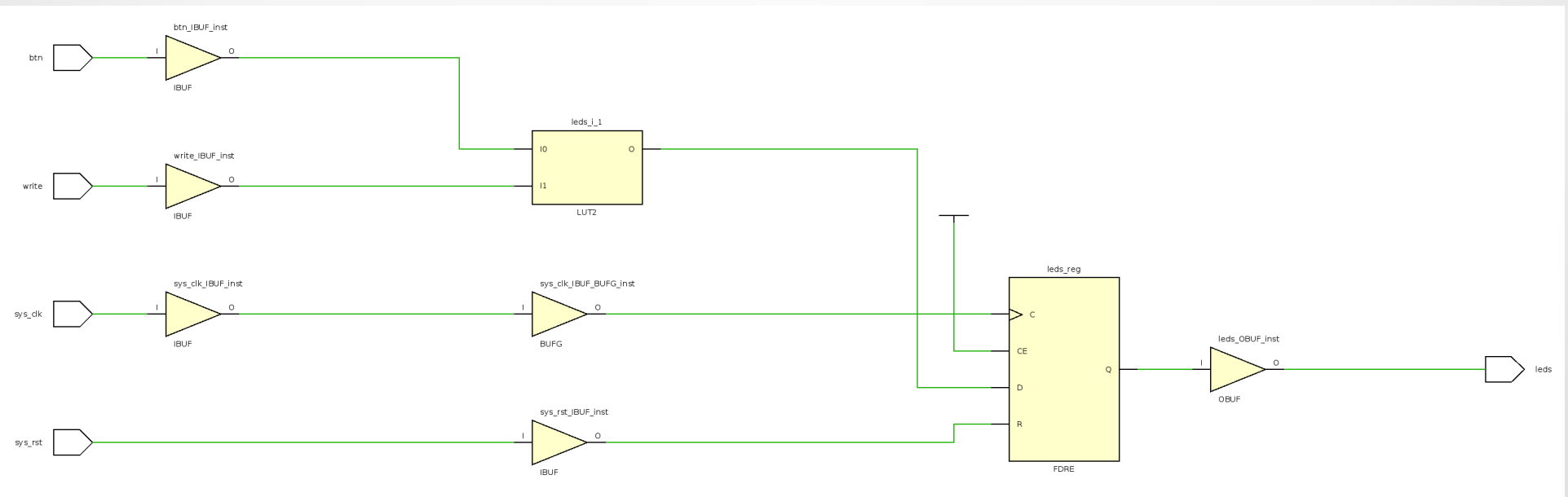
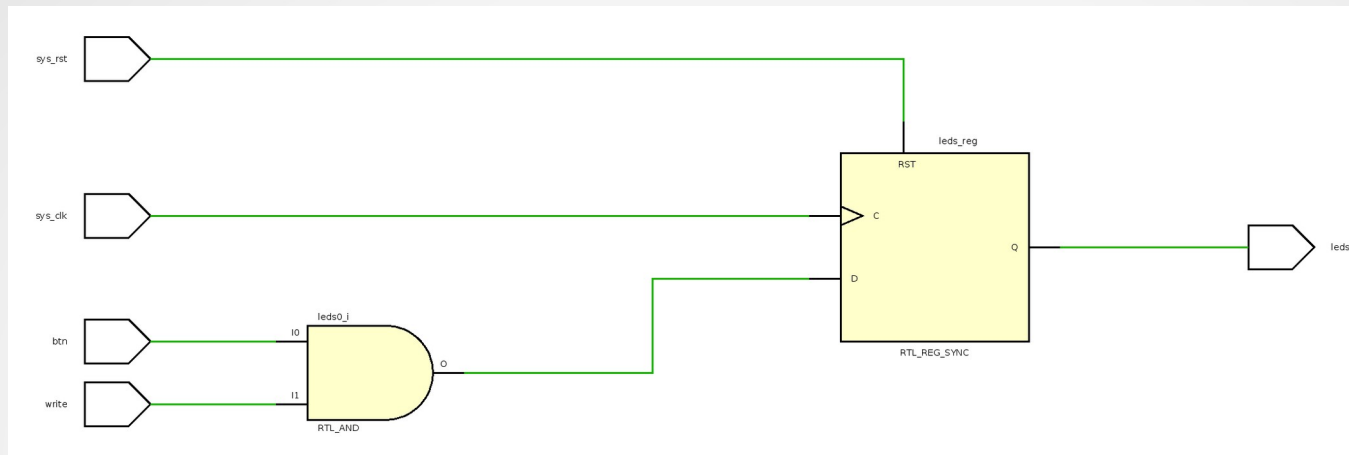
Migen – Synchronous example



```
module top(  
    output reg leds,  
    input btn,  
    input write,  
    input sys_clk,  
    input sys_rst  
);  
  
always @(posedge sys_clk) begin  
    leds <= (btn & write);  
    if (sys_rst) begin  
        leds <= 1'd0;  
    end  
end  
  
endmodule
```

```
class M1(Module):  
    def init (self):  
        # Interfaces  
        self.leds = Signal()  
        self.btn = Signal()  
        self.write = Signal()  
  
        ###  
  
        # Functional description. This is what this  
        # module does.  
        self.sync += self.leds.eq(self.btn & self.write)
```


Migen – Synchronous example



Agenda

- ▶ Migen: introduction and workshops
 - Concepts, Modules and signals
 - Blinker example
 - Attributes of Module()
 - Example of verilog output
 - Operators (If/Else and FSM)
 - Minimum project requirement (Migen/LiteX)
 - Workshop 1/2
 - Records
 - Simulation
 - Workshop 2/2

Migen – IF / ELSE

- ▶ Migen doesn't use Python's if/else.
- ▶ **If** is implemented as a Class. **Else** and **Elif** are methods.
- ▶ Assignments under If are separated by comas
- ▶ Can be used in **comb** or **sync** blocks

```
If(cond1,  
    asisgn1,  
    assign2,  
    ...  
) .Elif(cond2,  
    assign3,  
    assign4,  
    ...  
) .Else(  
    assign5,  
    assign6,  
    ...  
)
```

Migen – FSM

- ▶ Finite State Machine is a way to implement sequential execution
- ▶ **FSM()** is a module, it needs to be added to submodules
- ▶ States are defined with **fsm.act**
- ▶ Assignments are separated by a coma

```
self.submodules += FSM(reset_state="START")

fsm.act("START",
        # assignment1,
        # assignment2,
)
```

Migen – FSM

- ▶ **NextState()** is used to move to another state

```
self.submodules += FSM(reset_state="START")

fsm.act("START",
        # do something,
        NextState("WAIT")
)

fsm.act("WAIT",
        #...
)
```

Migen – FSM

- ▶ **NextValue(a, value)** is used make a synchronous assignment. It is equivalent to ***self.sync += a.eq(value)***
- ▶ The signal keep it's value outside the state it has been assigned

```
self.submodules += FSM(reset_state="START")

a = Signal(4, reset=3)

fsm.act("START",
        NextValue(a, 5),
        NextState("WAIT")
)

fsm.act("WAIT",
        If(a == 2,
            #...
        )
)
```

a will be equal to 5
on the next clock
cycle

Coming from "WAIT",
a is equal to 5

Migen – FSM

```
self.submodules += FSM(reset_state="START")  
  
a = Signal(4, reset=3)  
  
fsm.act("START",  
        NextValue(a, 5),  
        NextState("WAIT")  
)  
  
fsm.act("WAIT",  
        If(a == 2,  
           #...  
        )  
)
```

This is equivalent to
this (pseudo code)

```
self.sync += [  
    If(fsm == "START",  
       a.eq(5)  
    )  
]
```

Migen – FSM

- Direct assignment **.eq()** is used make a combinatorial assignment. It is equivalent to **self.comb += a.eq(value)**

```
self.submodules += FSM(reset_state="START")

a = Signal(4, reset=3)

fsm.act("START",
        a.eq(5)
        NextState("WAIT")
)

fsm.act("WAIT",
        # a == 0 here
)
```

a is equal to 5 as long as we are in this state

- a is equal to 5 when in “START” state and 0 in other states

Migen – FSM

```
self.submodules += FSM(reset_state="START")  
  
a = Signal(4, reset=3)  
  
fsm.act("START",  
        a.eq(5)  
        NextState("WAIT")  
)  
  
fsm.act("WAIT",  
        # a == 0 here  
)
```

This is equivalent to
this (pseudo code)

```
self.comb += [  
    If(fsm == "START",  
        a.eq(5)  
    )  
]
```

Migen - Libraries

Migen has a library (genlib) with most of the base elements required to digital logic:

- Records (group signals together with direction),
- FSM (Finite State Machine),
- Clock Domain Crossing,
- Memory,
- Instance (reuse Verilog/VHDL),
- FIFO,
- ...

Most of the useful functions are grouped in the Migen Cheatsheet

Agenda

- ▶ Migen: introduction and workshops
 - Concepts, Modules and signals
 - Blinker example
 - Attributes of Module()
 - Example of verilog output
 - Operators (If/Else and FSM)
 - Minimum project requirement (Migen/LiteX)
 - Workshop 1/2
 - Records
 - Simulation
 - Workshop 2/2

Migen/LiteX – Minimum project requirement

- ▶ Declare IO resources
- ▶ Choose a platform and gives it the IO list
- ▶ Request platform resources (IOs)
- ▶ Assign requested resources to Module's interface
- ▶ Add timing constraints
- ▶ Let the platform build system do its job (build the bitstream)

Migen/LiteX – IO resources

- Declare IO resources (as a python list of tuples)

```
io = [  
    ("sys_clk", 0, Pins("35"), IOStandard("LVCMOS33")),  
  
    ("user_btn", 0, Pins("15"), IOStandard("LVCMOS33")),  
    ("user_btn", 1, Pins("14"), IOStandard("LVCMOS33")),  
  
    ("user_led", 0, Pins("16"), IOStandard("LVCMOS33")),  
    ("user_led", 1, Pins("17"), IOStandard("LVCMOS33")),  
    ("user_led", 2, Pins("18"), IOStandard("LVCMOS33")),  
]
```

- Look all available options in the documentation

Migen/LiteX – IO resources

- ▶ Declare IO resources (as a python list of tuples)

```
io = [  
    ("sys_clk", 0, Pins("35"), IOStandard("LVCMOS33")),  
  
    ("user_btn", 0, Pins("15"), IOStandard("LVCMOS33")),  
    ("user_btn", 1, Pins("14"), IOStandard("LVCMOS33")),  
  
    ("user_led", 0, Pins("16"), IOStandard("LVCMOS33")),  
    ("user_led", 1, Pins("17"), IOStandard("LVCMOS33")),  
    ("user_led", 2, Pins("18"), IOStandard("LVCMOS33")),  
]
```

- ~~▶ Look all available options in the documentation~~

No documentation (for now)

Migen/LiteX – Platform

- Choose a platform and pass it the IO list

```
class Platform(GowinPlatform):  
    def __init__(self):  
        GowinPlatform.__init__(self, "GW1N-LV1QN48C6/I5", _io, [], toolchain="gowin", devicename="GW1N-1")  
        self.toolchain.options["use_done_as_gpio"] = 1  
        self.toolchain.options["use_reconfig_as_gpio"] = 1
```

- Litex provides infrastructure for:

- altera,
- efinix,
- gowin,
- lattice,
- microsemi,
- quicklogic,
- xilinx

Migen/LiteX – Request resources

- Request platform resources (IOs)

```
class Tuto(Module):  
    def __init__(self, platform):  
  
        # Get pin from ressources  
        rst = platform.request("user_btn", 0)  
        btn = platform.request("user_btn", 1)  
        clk = platform.request("sys_clk")
```

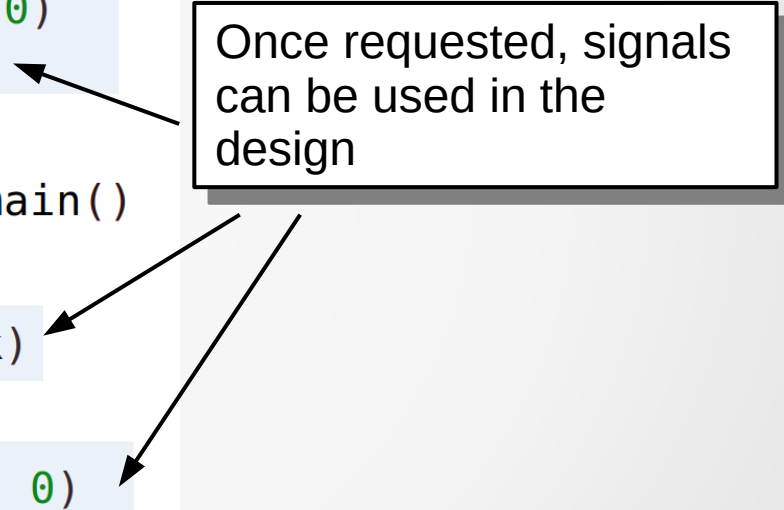
- Returns Signal() from platform resources

Migen/LiteX – Minimum project code

- Assign requested resources to Module's interface

```
class Tutor(Module):  
    def __init__(self, platform):  
        # Get pins from resources  
        rst = platform.request("user_btn", 0)  
        clk = platform.request("sys_clk")  
  
        # Create "sys" clock domain  
        self.clock_domain.cd_sys = ClockDomain()  
  
        # Assign clock pin to clock domain  
        self.comb += self.cd_sys.clk.eq(clk)  
  
        # Request led pin  
        led0 = platform.request("user_led", 0)  
  
        # Instance of Blink  
        blink = Blink(22)  
        self.submodules += blink  
        self.comb += led0.eq(blink.out)
```

Once requested, signals can be used in the design



Migen/LiteX – Minimum project code

- Assign requested resources to Module's interface

```
class Tutor(Module):  
    def __init__(self, platform):  
        # Get pins from resources  
        rst = platform.request("user_btn", 0)  
        clk = platform.request("sys_clk")  
  
        # Create "sys" clock domain  
        self.clock_domain.cd_sys = ClockDomain()  
  
        # Assign clock pin to clock domain  
        self.comb += self.cd_sys.clk.eq(clk)  
  
        # Request led pin  
        led0 = platform.request("user_led", 0)  
  
        # Instance of Blink  
        blink = Blink(22)  
        self.submodules += blink  
        self.comb += led0.eq(blink.out)
```

cd_sys is mandatory. At some point it has to be created

The clock signal has to be assigned

Migen/LiteX – Minimum project code

- Assign requested resources to Module's interface

```
class Tutor(Module):  
    def __init__(self, platform):  
        # Get pins from resources  
        rst = platform.request("user_btn", 0)  
        clk = platform.request("sys_clk")  
  
        # Create "sys" clock domain  
        self.clock_domain.cd_sys = ClockDomain()  
  
        # Assign clock pin to clock domain  
        self.comb += self.cd_sys.clk.eq(clk)  
  
        # Request led pin  
        led0 = platform.request("user_led", 0)  
  
        # Instance of Blink  
        blink = Blink(22)  
        self.submodules += blink  
        self.comb += led0.eq(blink.out)
```

← Add and use a module

Migen/LiteX – Minimum project code

- ▶ Add timing constraints

```
# Add a timing constraint  
platform.add_period_constraint(clk, 1e9/24e6)
```



This is the requested
clock signal

Migen/LiteX – Minimum project code

- Build the bitstream

```
platform = Platform()  
design = Tuto(platform)  
platform.build(design, build_dir='gateway')
```

Configured platform

Top level Module

Ask LiteX to build the
bitstream

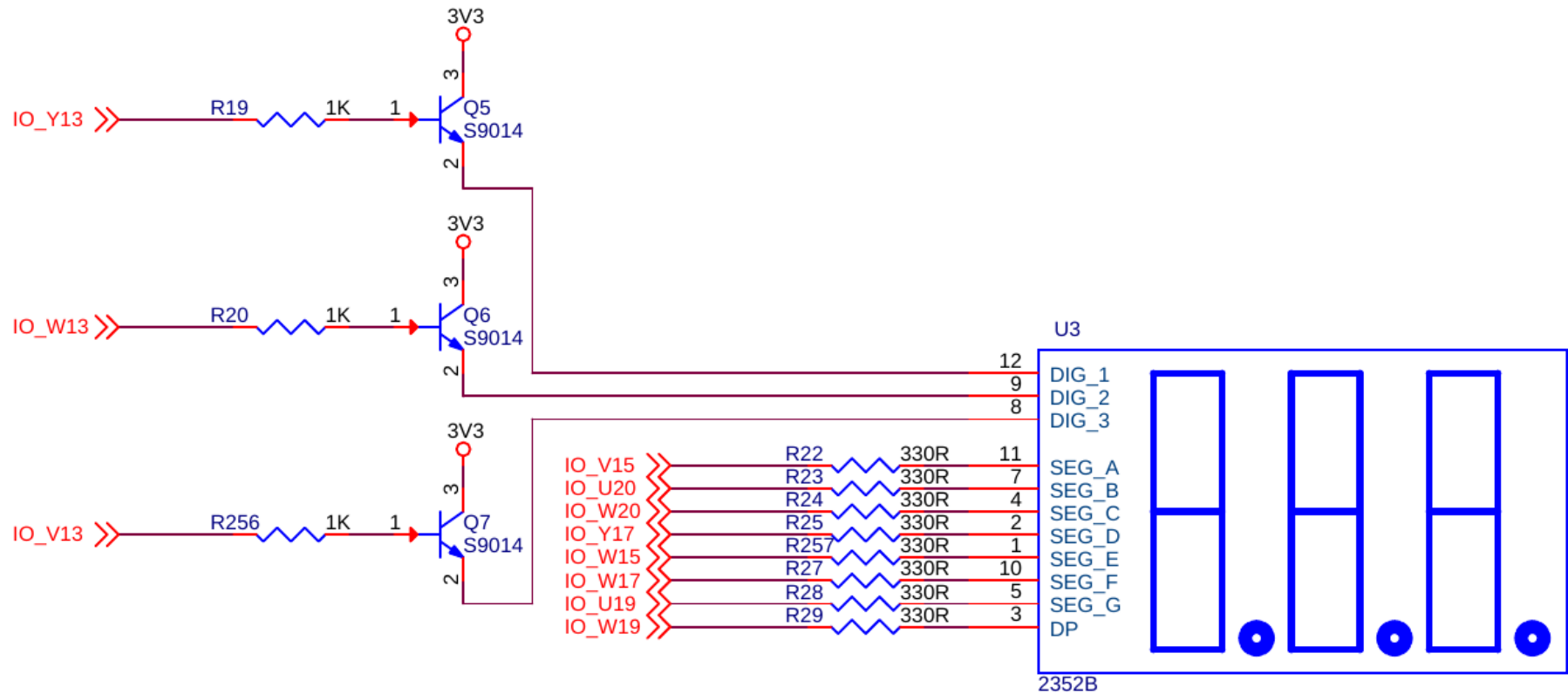
Now, let's practice !

Step0 – Led blinker

What you'll see:

- ▶ Platform definition
- ▶ Resources assignment
- ▶ Submodules
- ▶ Simulation
- ▶ Build

Step1 – 7 segment controller



Step1 – 7 segment controller

- ▶ Write a **SevenSegment** module to convert a 4 bits values to 7 outputs controlling segments
- ▶ Write a **SevenSegmentsController** module controlling the multiplexing.
 - One 12 bits input (3 digits, 4 bits per digit), input value
 - One 3 bits output to control each digit anode
 - One 7 bits output to control the segments

Step1 – 7 segment controller

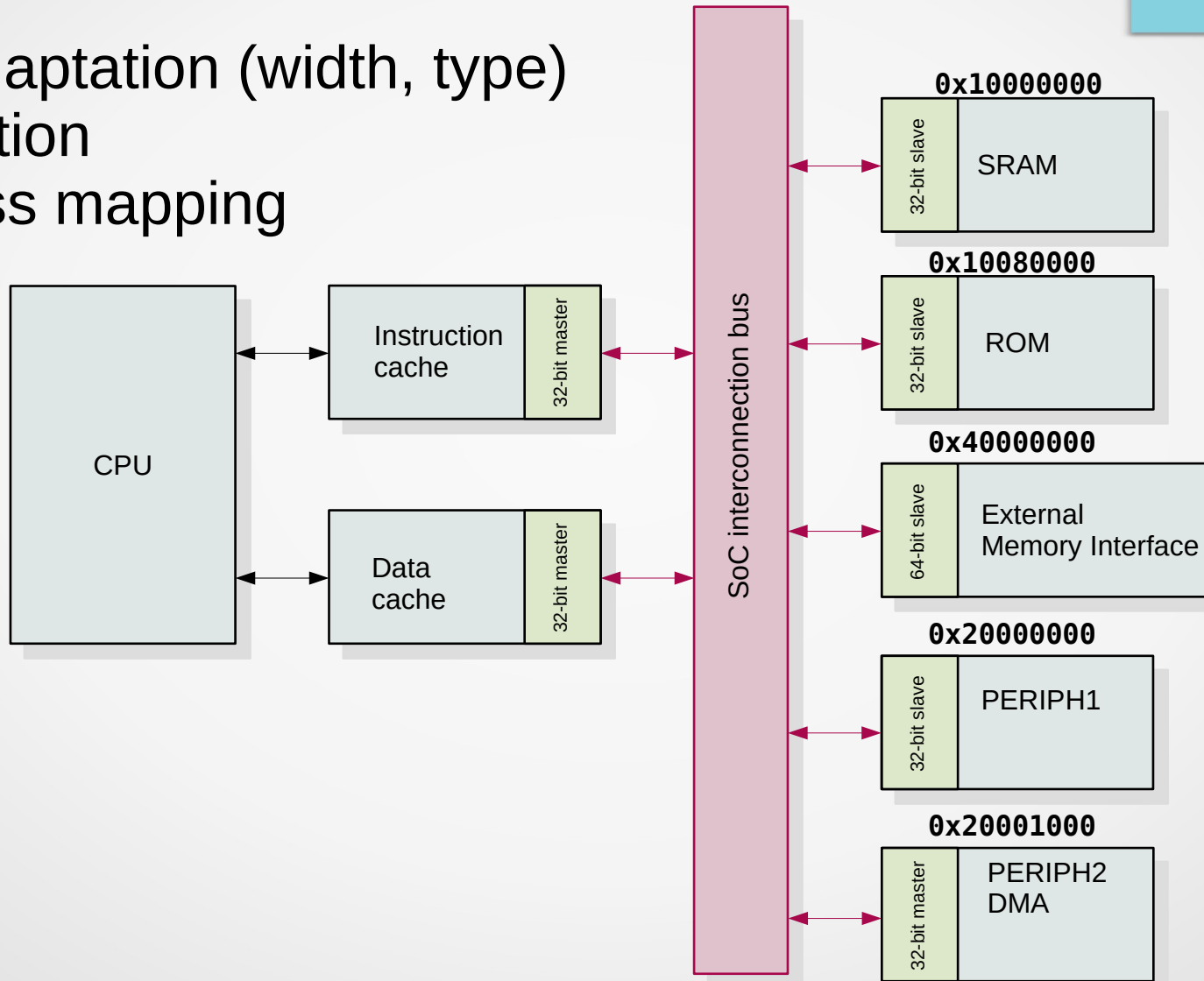
- ▶ Get to work !

Agenda

- ▶ Description of FPGAs
- ▶ Digital design challenges
- ▶ Migen: introduction and workshops
- ▶ LiteX: introduction and workshops
 - Key features
 - BIOS
 - LiteX tools
 - Workshop

System On Chip - SoC

- ▶ Bus adaptation (width, type)
- ▶ Arbitration
- ▶ Address mapping



What is LiteX

11 Open sources IP

litepcie, litedram, liteiclink,
liteeth, litesdcard, litevideo,
litescope, litesata, litejesd204b



litehyperbus, litespi

EnjoyDigital

104 Supported boards

- Platform definition
- Target example



supports

16 Softcores

serv, morlhx,
vexriscv, picorv32,
cortex_m1, ...

lites-boards

platforms
targets

Migen

Uses
Extends

LiteX

supports

provides

build

tools

soc

altera
anlogic
efinix
gowin
lattice
microsemi
quicklogic
xilinx

server
term
client
...

cores
interconnect
integration
software
doc

gpio
i2s
spi
pwm
led
tmds
...

EnjoyDigital



LiteX's key features

- ▶ Extends Migen with new concepts and libraries
- ▶ Build and configure SoC easily
 - Scale from no CPU to Linux capable SoC
 - Open sources IP
 - Easy interconnection of modules
 - Flexible SoC configuration
 - Unified build system across vendors
- ▶ Portability (abstraction of technology implementation)
- ▶ Debug infrastructure with LiteX Server, LiteScope and other tools
- ▶ BIOS with command line interface for system bring-up

LiteX – Example

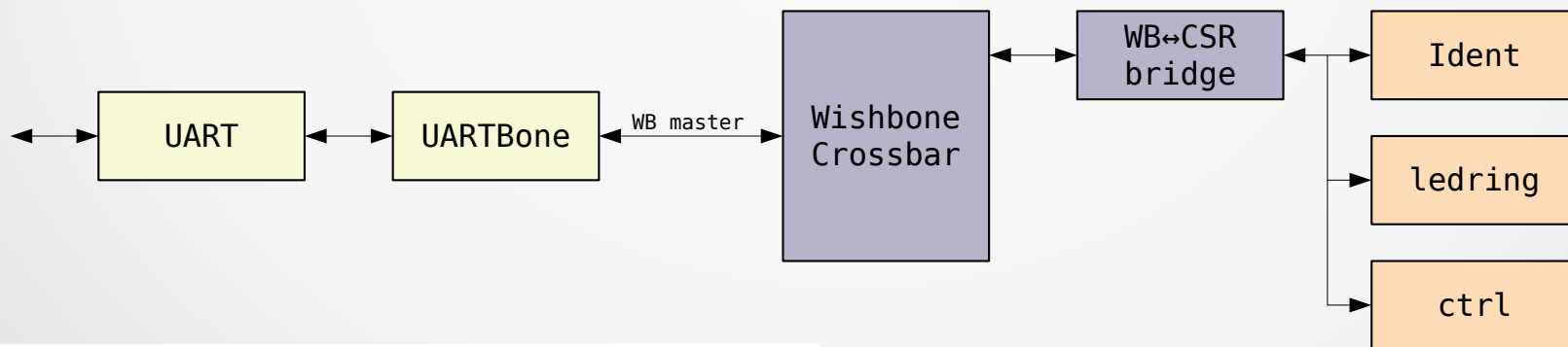
```
class BaseSoC(SoCMini):
    def __init__(self, sys_clk_freq=int(24e6), **kwargs):
        platform = Platform()

        # SoCMini -----
        SoCMini.__init__(self, platform, sys_clk_freq,
                        ident      = "LiteX SoC on Tang Nano",
                        ident_version = True)

        # CRG -----
        self.submodules.crg = _CRG(platform, sys_clk_freq)

        # UARTBone -----
        self.add_uartbone(baudrate=115200)

        # Leds -----
        self.submodules.ledring = RingControl(platform.request("do"), 12, sys_clk_freq)
        self.add_csr("ledring")
```



```
#-----
# Auto-generated by Migen (-----) & LiteX (6692c73d)
#-----
csr_base,ledring,0x00000000,,
csr_base,ctrl,0x00000800,,
csr_base,identifier_mem,0x00001000,,
```

Agenda

- ▶ Description of FPGAs
- ▶ Digital design challenges
- ▶ Migen: introduction and workshops
- ▶ **LiteX: introduction and workshops**
 - Key features
 - BIOS
 - LiteX tools
 - Workshop

LiteX – Software, BIOS

```
  / /  ( ) /  _ | / /
 / / / /  _ / - ) > <
 / _ / / \ _ \ / / | _ |
Build your hardware, easily!

(c) Copyright 2012-2021 Enjoy-Digital
(c) Copyright 2007-2015 M-Labs

BIOS built on Dec 13 2021 23:07:24
BIOS CRC passed (ce9bfe35)

Migen git sha1: -----
LiteX git sha1: 40c001d5

----- SoC -----
CPU:          VexRiscv @ 250MHz
BUS:          WISHBONE 32-bit @ 4GiB
CSR:          32-bit data
ROM:          128KiB
SRAM:         8KiB
FLASH:        8192KiB

----- Boot -----
Bootling from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
          Timeout
No boot medium found

----- Console -----

litex>
```

- ▶ Built-in BIOS with low level commands to test the SoC
- ▶ Uses picolibc
- ▶ Several boot sources (RAM, flash, ROM, serial, tftp, sata, sdcard)
- ▶ Not a full featured bootloader. Think of a first stage bootloader.

Agenda

- ▶ Description of FPGAs
- ▶ Digital design challenges
- ▶ Migen: introduction and workshops
- ▶ **LiteX: introduction and workshops**
 - Key features
 - BIOS
 - **LiteX tools**
 - Workshop

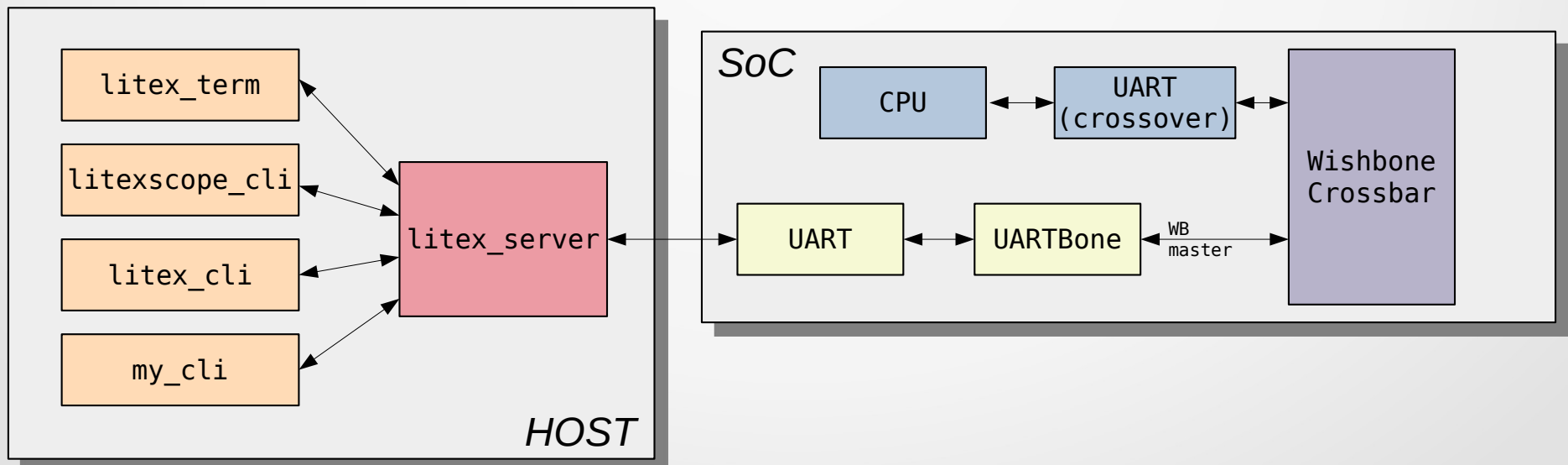
LiteX - Tools

- ▶ **litex_server** → proxy between tools and SoC interconnection crossbar
- ▶ **litex_term** → terminal emulator with SFL (Serial Flash Loader) capabilities
- ▶ **litex_cli** → simple read/write access to SoC interconnection crossbar
- ▶ **litescope_cli** → control tool for an embedded logic analyzer

LiteX – Tools, litex_server

- ▶ Allows simultaneous access to the SoC interconnect from tools
- ▶ Needs a bridge (UART, Ethernet, PCIe)
- ▶ Uses Etherbone protocol (“standardized” wishbone over IP)

```
$ litex_server --uart --uart-port /dev/ttyUSB2  
[CommUART] port: /dev/ttyUSB2 / baudrate: 115200 / tcp port: 1234
```



LiteX – Tools, litex_term

- ▶ Can interface the Serial Flash Loader (SFL) of the BIOS
- ▶ Only binary files (no elf)
- ▶ Default loading address is 0x40000000 (main_ram)

```
$ litex_term --kernel=demo.bin /dev/ttyUSB2

litex>
litex>
litex>
litex> serialboot
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[LXTERM] Received firmware download request from the device.
[LXTERM] Uploading demo.bin to 0x40000000 (6072 bytes)...
[LXTERM] Upload calibration... (inter-frame: 10.00us, length: 64)
[LXTERM] Upload complete (9.4KB/s).
[LXTERM] Booting the device.
[LXTERM] Done.
Executing booted program at 0x40000000

--===== Liftoff! =====--

LiteX minimal demo app built Dec 15 2021 13:23:59

Available commands:
help                - Show this command
reboot              - Reboot CPU
donut               - Spinning Donut demo
helloc              - Hello C
litex-demo-app>
```

LiteX – Tools, `litescli`

- ▶ Can read/write to arbitrary address
- ▶ Knows SoC registers (read from `csr.csv` file)
- ▶ Needs to connect to `liteserver`

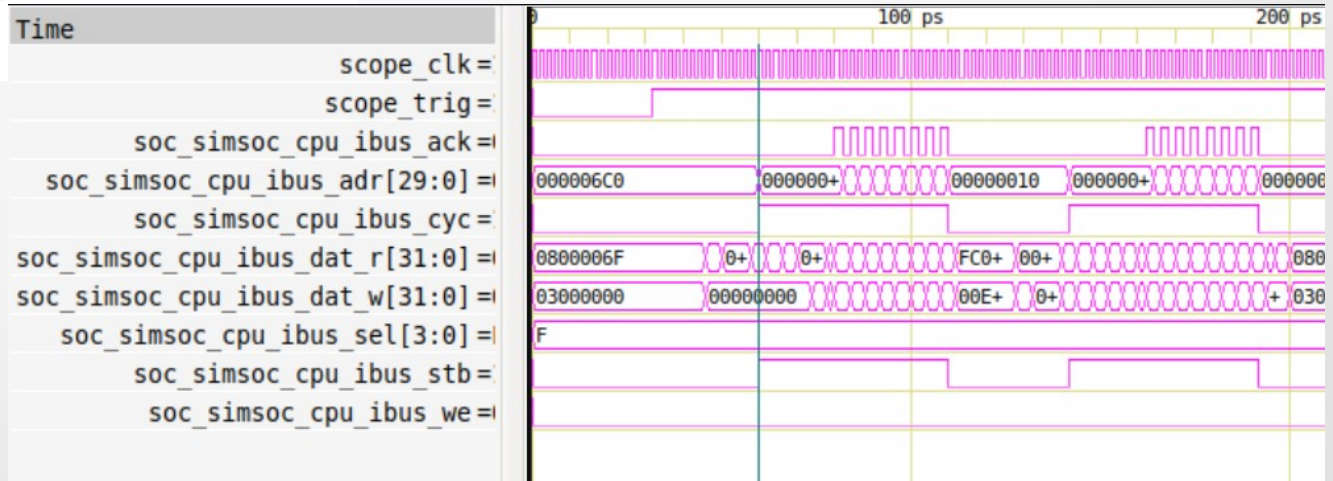
```
$ litescli --regs --filter=timer
0xf0001000 : 0x02faf080 timer0_load
0xf0001004 : 0x00000000 timer0_reload
0xf0001008 : 0x00000001 timer0_en
0xf000100c : 0x00000001 timer0_update_value
0xf0001010 : 0x00000000 timer0_value
0xf0001014 : 0x00000001 timer0_ev_status
0xf0001018 : 0x00000001 timer0_ev_pending
0xf000101c : 0x00000000 timer0_ev_enable
```

LiteX- Tools, litescope_cli

- ▶ **litescope** can be integrated to the design to observe internal signals
- ▶ **litescope_cli** can control **litescope** through **litex_server** (trigger)
- ▶ Needs **analyzer.csv** generated during build

```
$litescope_cli -r soc_simsoc_cpu_ibus_stb  
Exact: soc_simsoc_cpu_ibus_stb  
Rising edge: soc_simsoc_cpu_ibus_stb  
[running]...
```

```
[uploading]...  
[=====>] 100%  
[writing to dump.vcd]...
```



Agenda

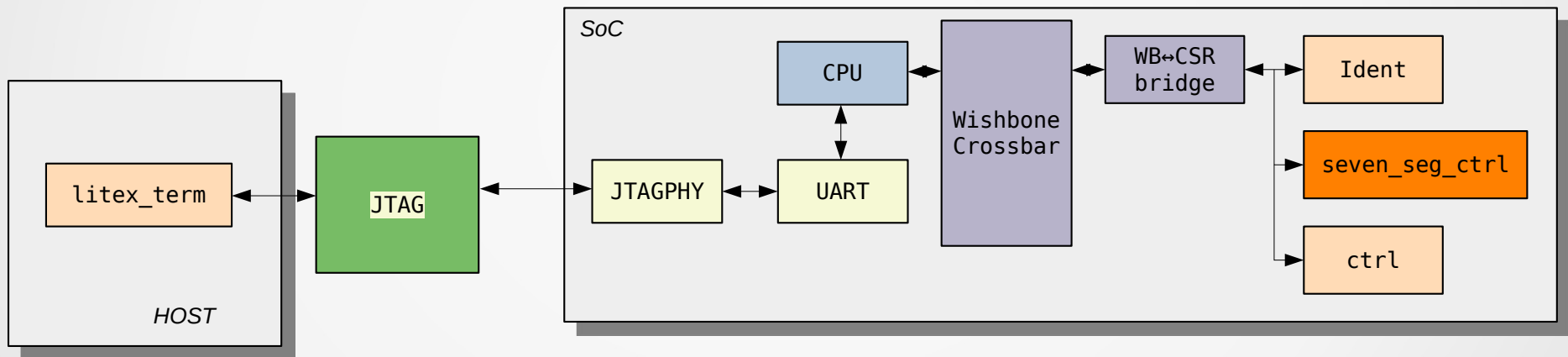
- ▶ Description of FPGAs
- ▶ Digital design challenges
- ▶ Migen: introduction and workshops
- ▶ **LiteX: introduction and workshops**
 - Key features
 - BIOS
 - LiteX tools
 - **Workshop**

Step2 – Litex demo

What you'll see:

- ▶ Argument parsing and SoC configuration
- ▶ BIOS over jtag_uart

Step2 – Litex demo



Step2_bis – Litex demo

What you'll see:

- ▶ Use `litex_server`
- ▶ Use `litex_cli`
- ▶ Control your SoC using python

Step2_bis – Litex demo

