V1.0

Introduction to digital design with Migen and Litex

Franck Jullien

@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
- LiteX: advanced topics

Digital Design – Base elements

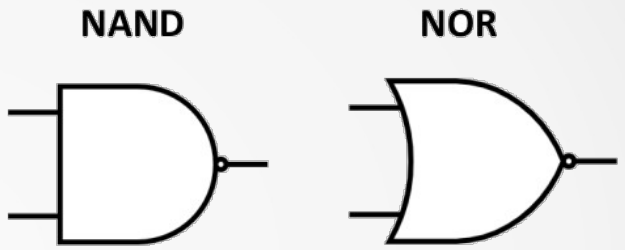
Hardware consist of a few simple building blocks

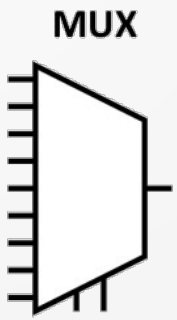
1. Combinatorial

- “Instant” state changes, e.g.:

  Classical gates (especially NAND & NOR)

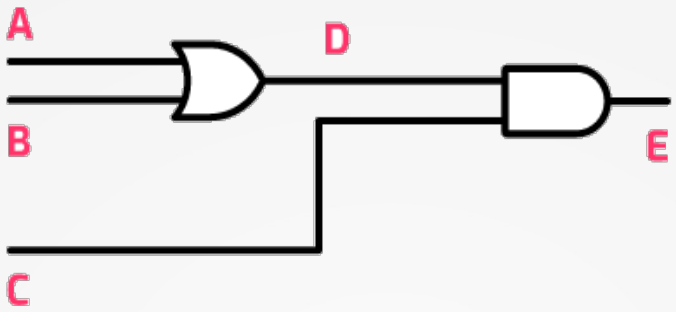
  Multiplexer (MUX)





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

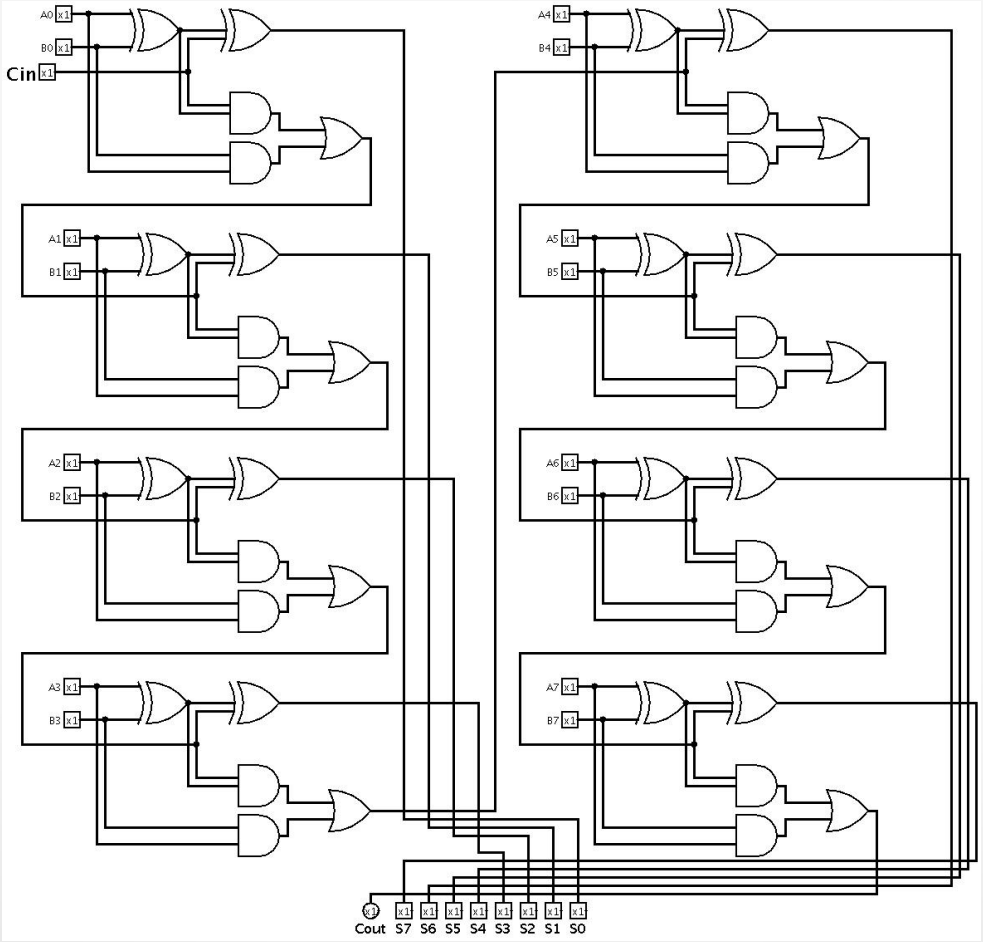
Digital Design – Truth table



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

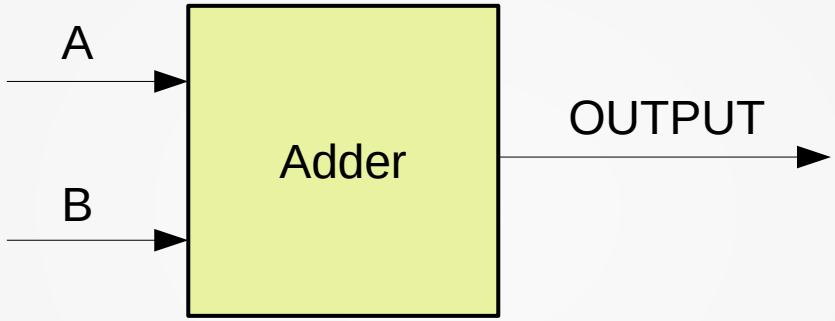
Digital Design – 8 bits adder

Design of an 8 bits adder



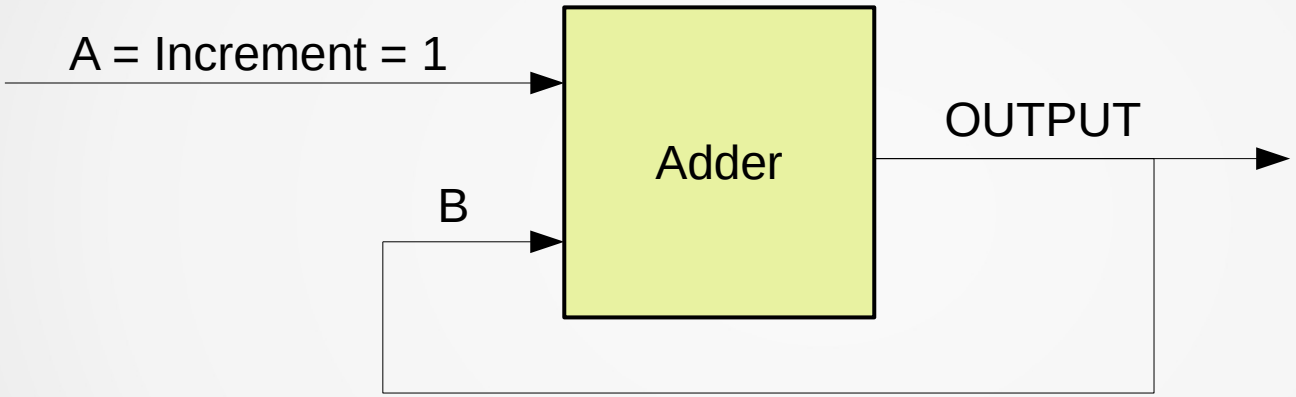
Digital Design – 8 bits counter

Design of an 8 bits counter



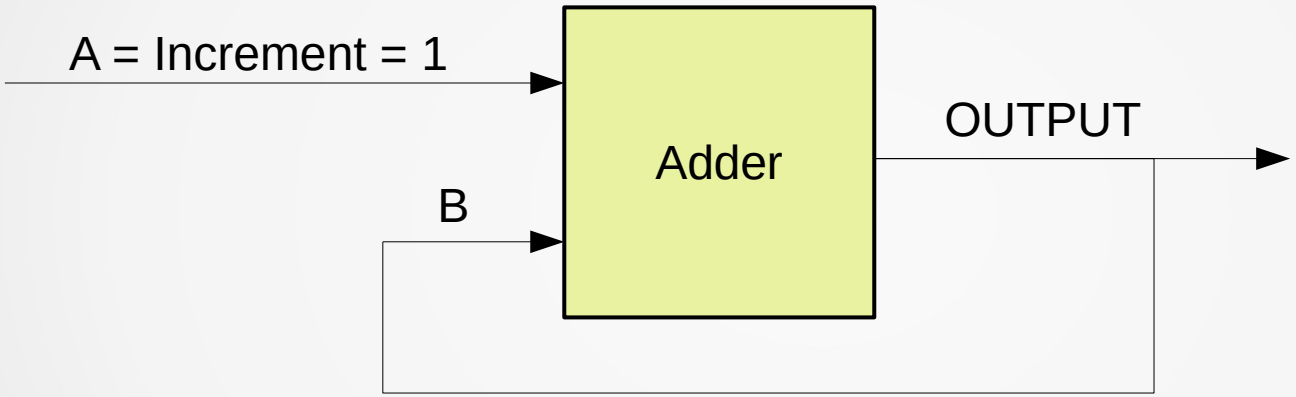
Digital Design – 8 bits counter

Design of an 8 bits counter



Digital Design – 8 bits counter

Design of an 8 bits counter

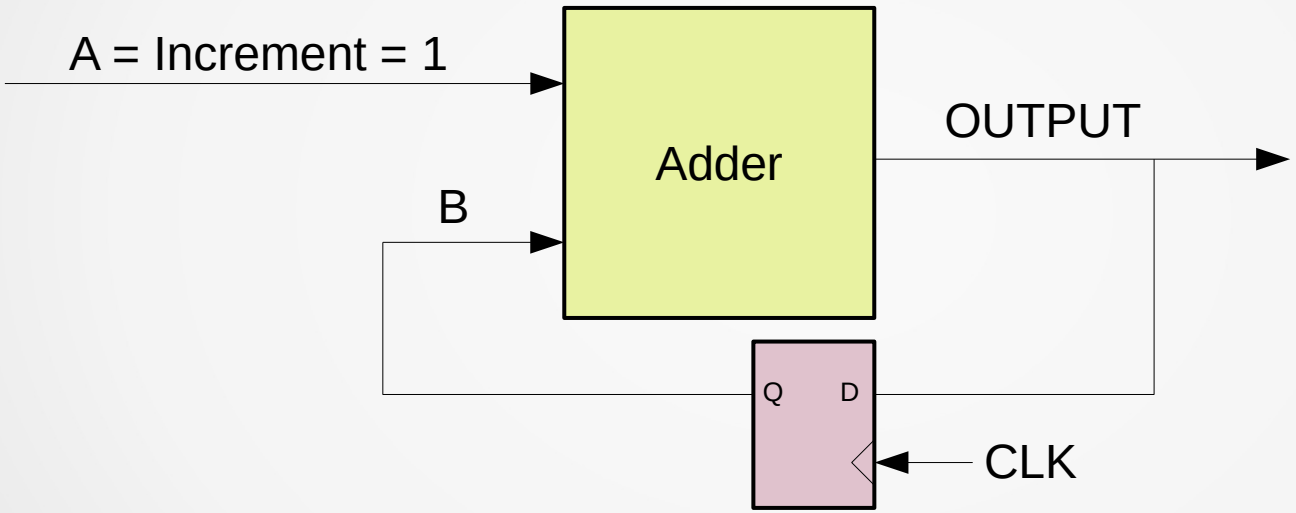


We have a infinite loop (combinatorial loop) !!

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.:

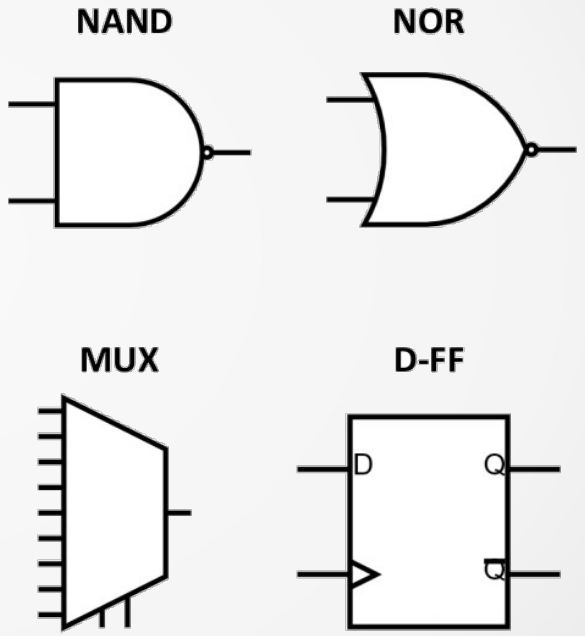
  Classical gates (especially NAND & NOR)

  Multiplexer (MUX)

2. Synchronous

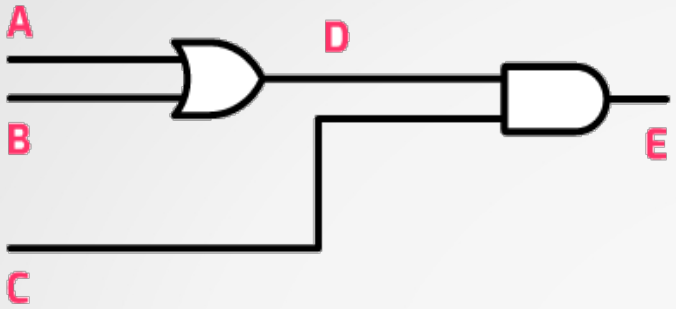
- “Clocked” state changes, e.g.:

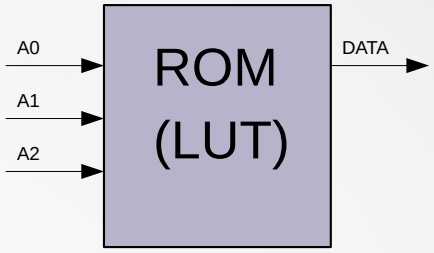
  Flip Flop (e.g. D-FF, register)



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

=

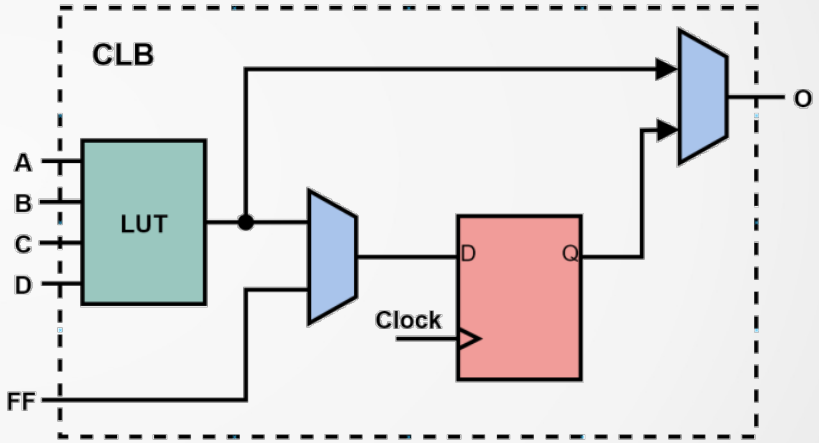
ADDR	DATA
000	0
001	0
010	0
011	1
100	0
101	1
110	0
111	1

Anatomy of FPGAs - CLB

FPGA are made of Configurable Logic Block (CLB)

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

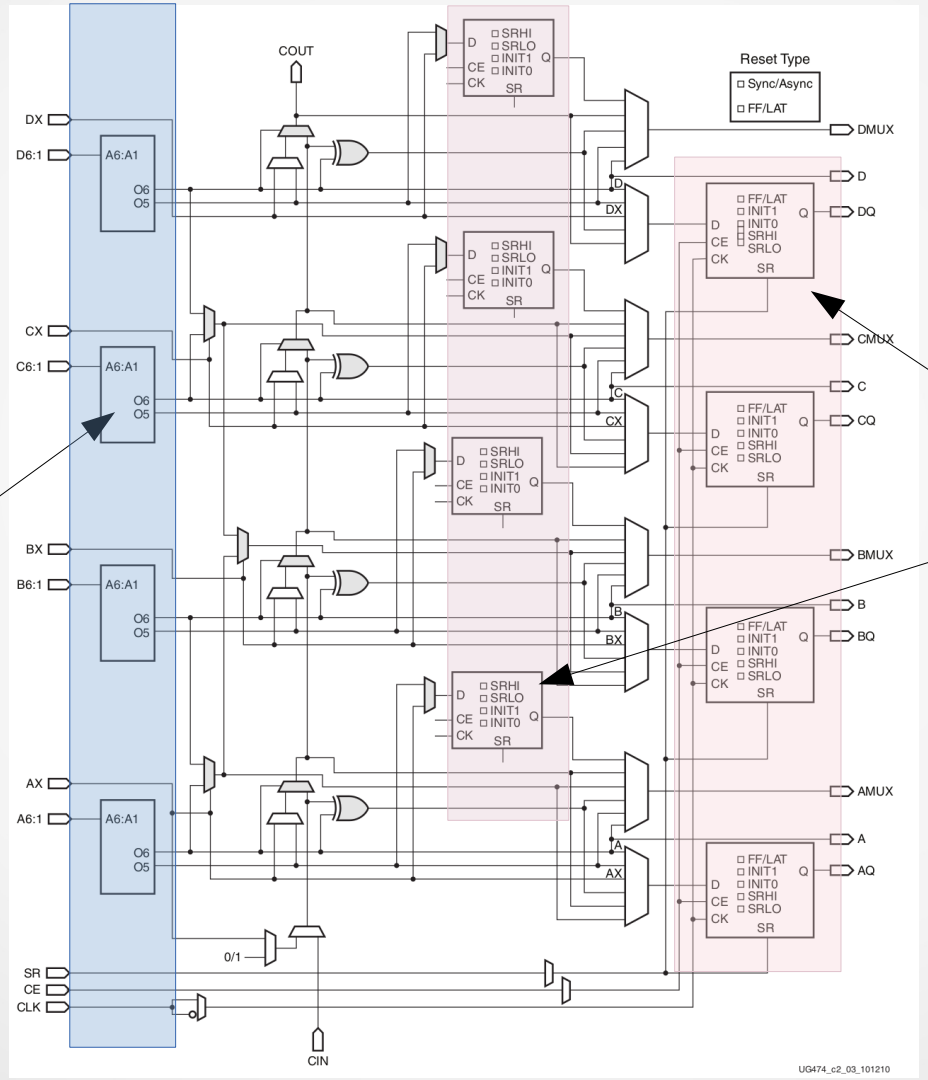
- LUT configuration is flexible
- D-type flip-flops configuration is flexible
- Flip-flops can take input from outside the CLB or from the LUT



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)



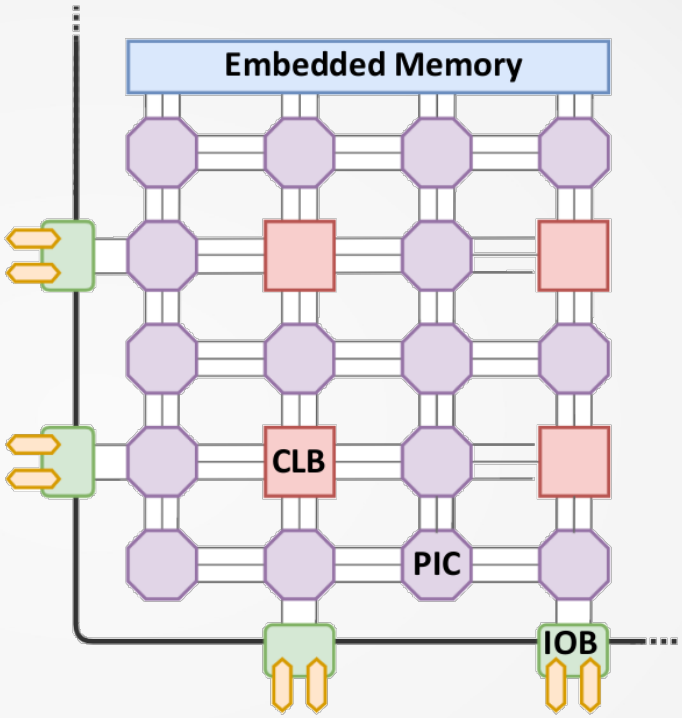
LUT

Flip-flops

ug474_7series_CLB.pdf

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

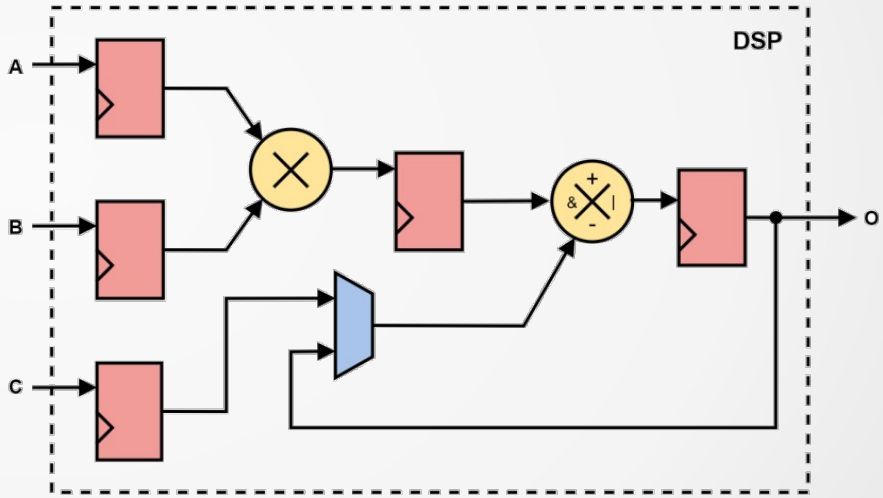
Hardened Cores (Also called “IP cores”)

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

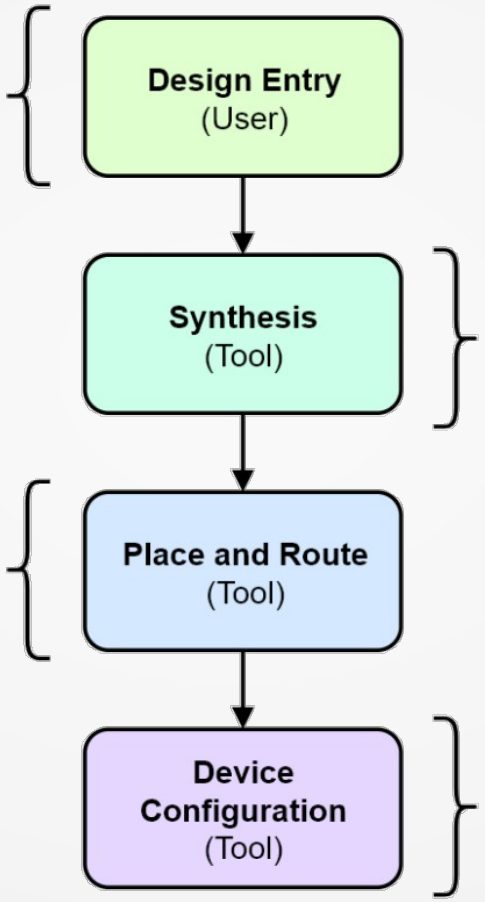
[1] ug473_7Series_Memory_Resources.pdf

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

- Functional description
- Interface description
- Behavioral Simulation



- Synthesis tool converts hardware description into **netlist** (building blocks and interconnect)
- Can perform logic optimization

Find best location of primitives for all elements in netlist

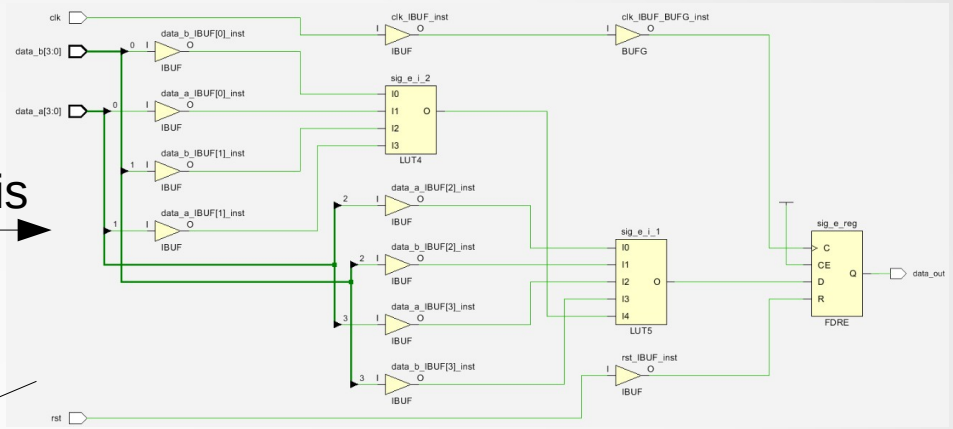
Generating bitstream for direct FPGA programming or for external memory configuration

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

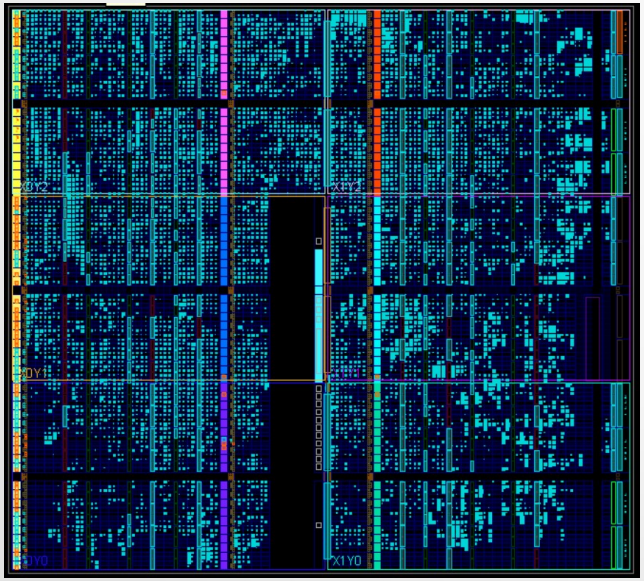
Anatomy of FPGAs - Classical Design Flow

```
    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

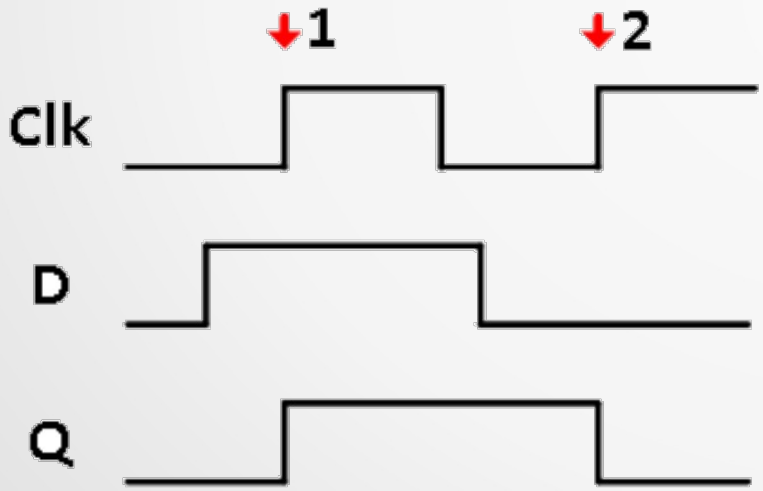
- Bitstream is stored in flash memory.
  → Instant ON
- Very small amount of logic resources
- No on-die hard IPs available (RAM, PLL,…)
- Only one voltage rail
- Available in TQFP package

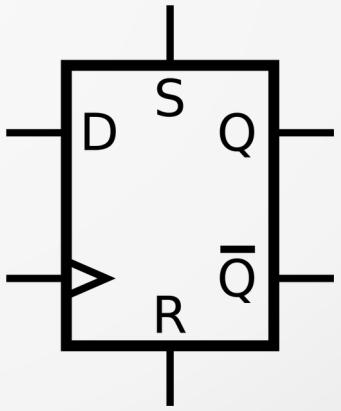
Agenda

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

Flip-Flops - Description

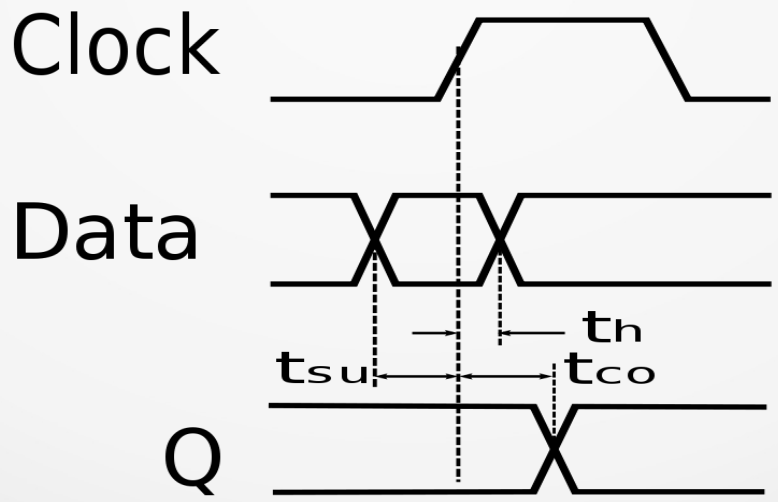
- There are a few different types of flip-flops (JK, T, D) but the one that is used most frequently is the D Flip-Flop.
- Sequential logic operates on the transitions of a clock. When a Flip-Flop sees a rising edge of the clock, it **registers** (copy and hold) the data from the Input D to the Output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

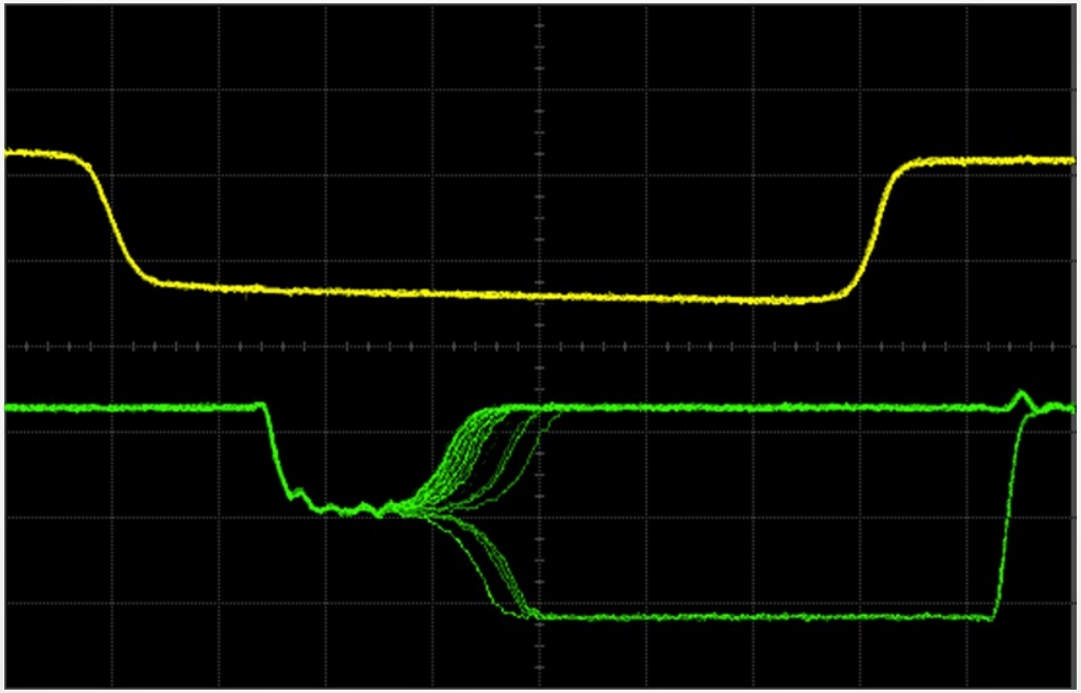
- Because of the construction of a flip-flop [1], the input must be held steady in a period around the rising edge of the clock.
- **Setup time** is the minimum amount of time the data input should be held steady before the clock event, so that the data is reliably sampled by the clock.
- **Hold time**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

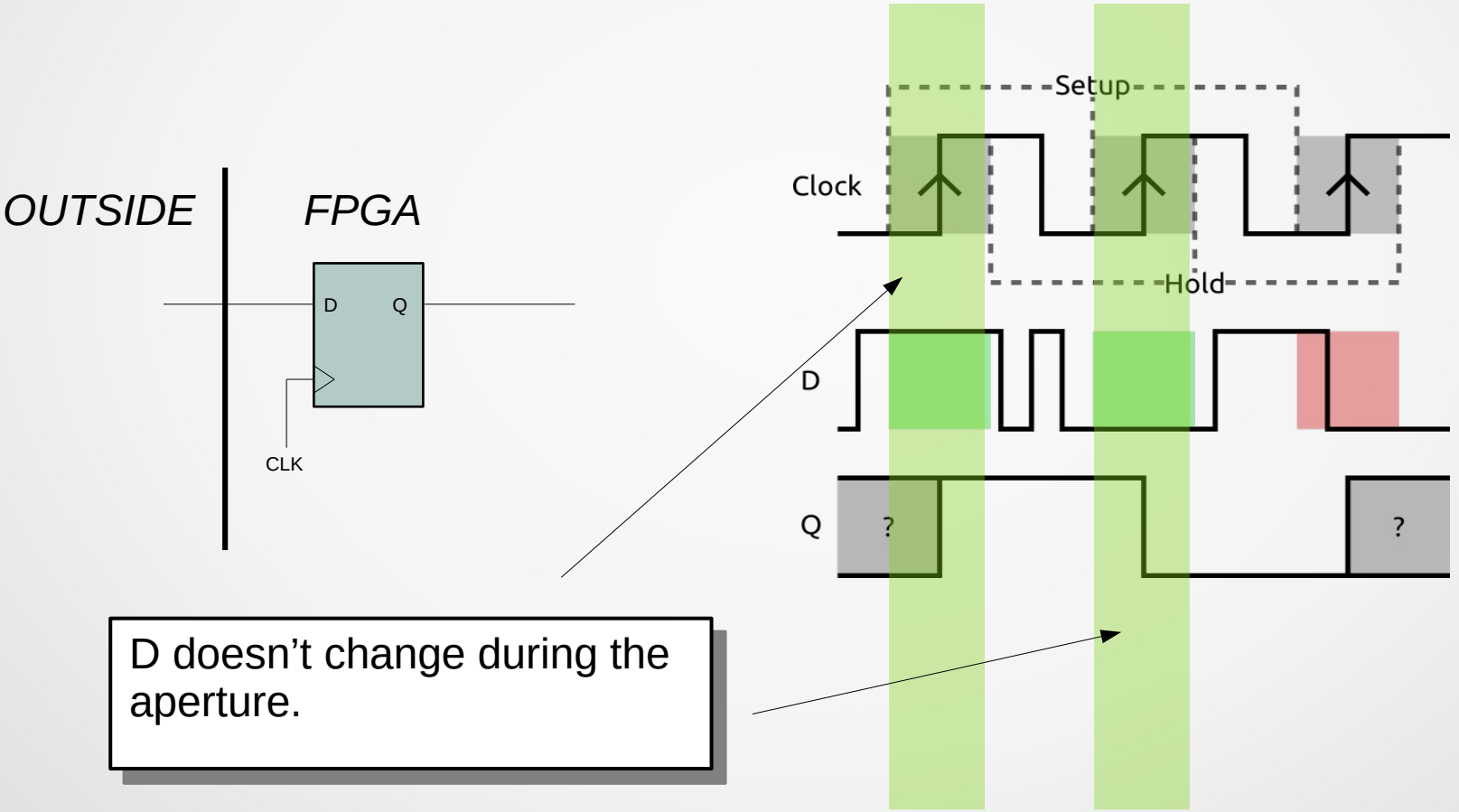
- If setup and hold time are not respected, flip-flops are subject to a problem called **metastability**
- The result is that the output may behave unpredictably, taking many times longer than normal to settle to one state or the other, or even oscillating several times before settling.



https://youtu.be/5PRuPVIjEcs

Three main reasons for metastability problem (1/3)

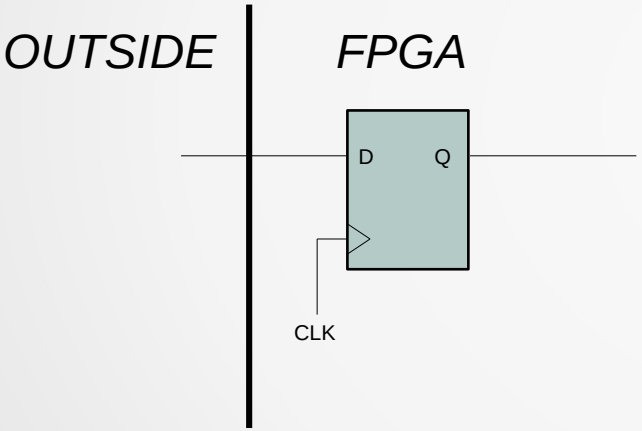
- An external signal (user input) is read inside the FPGA:

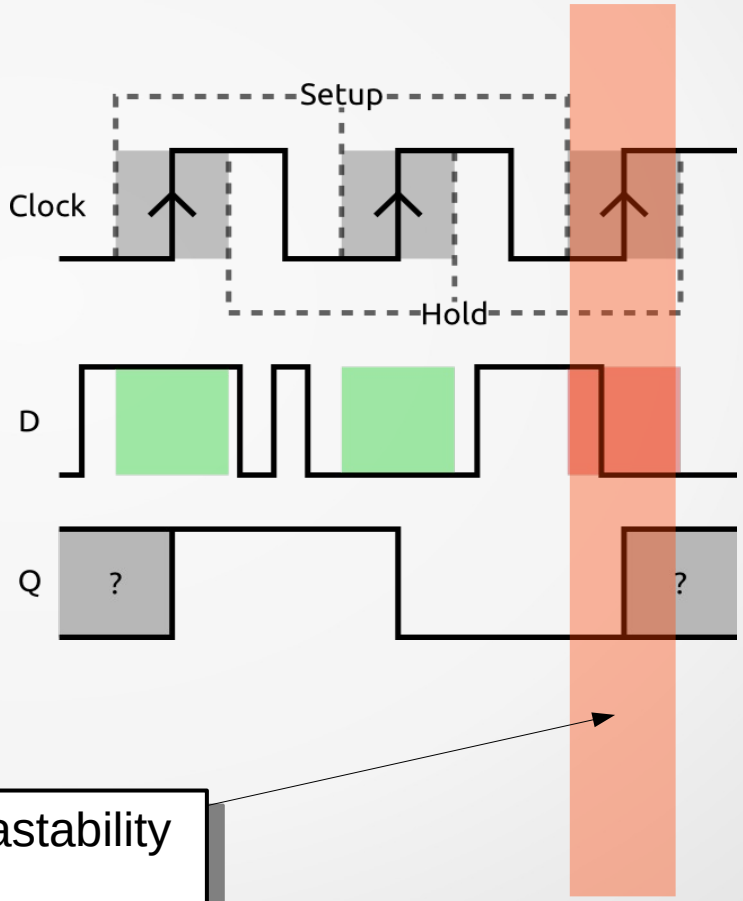


D doesn’t change during the aperture.

Three main reasons for metastability problem (1/3)

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

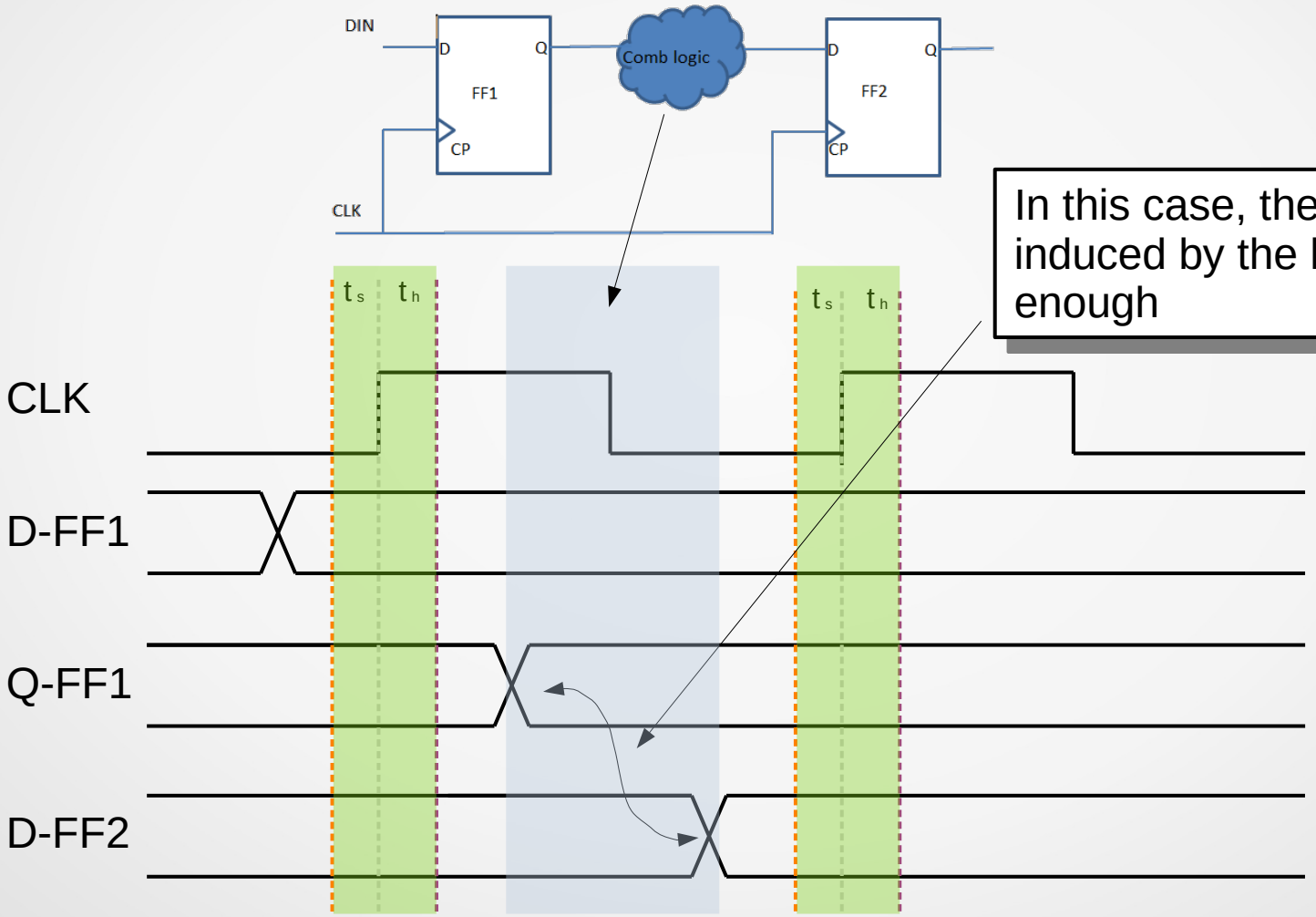




D changes during the aperture→Metastability

Three main reasons for metastability problem (2/3)

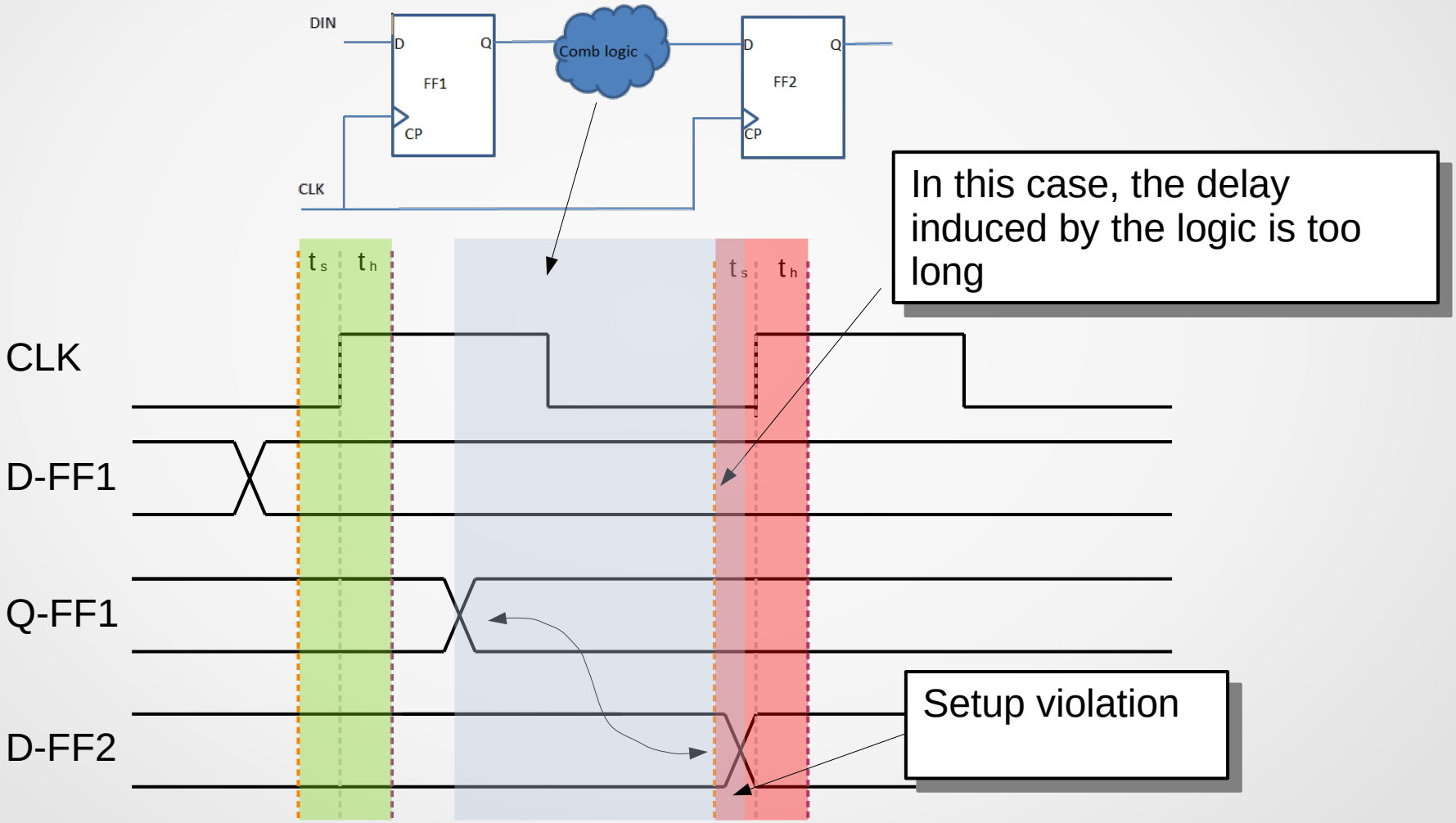
- Too much logic (delay) between flip-flops (setup violation):



In this case, the delay induced by the logic is short enough

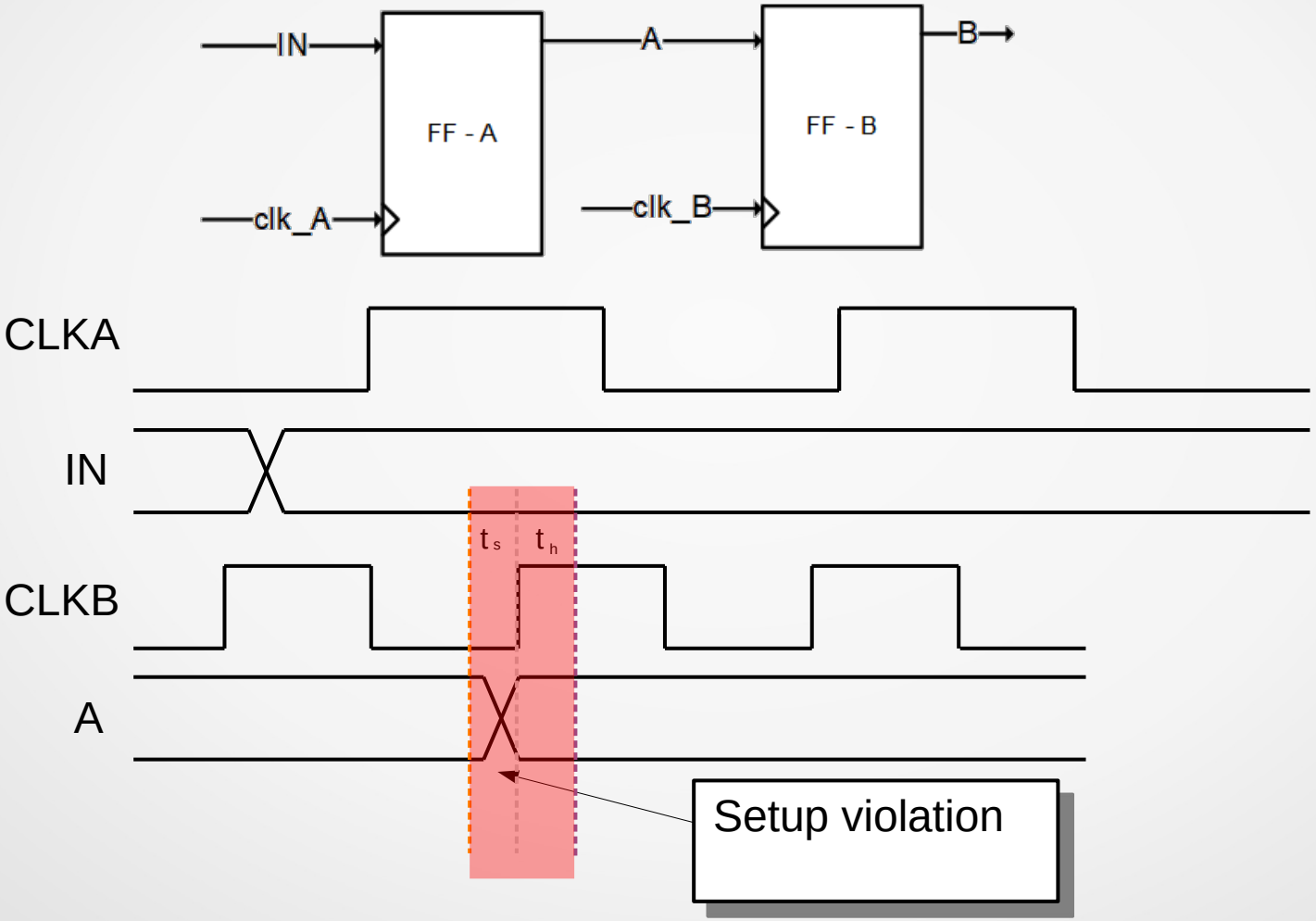
Three main reasons for metastability problem (2/3)

- 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.

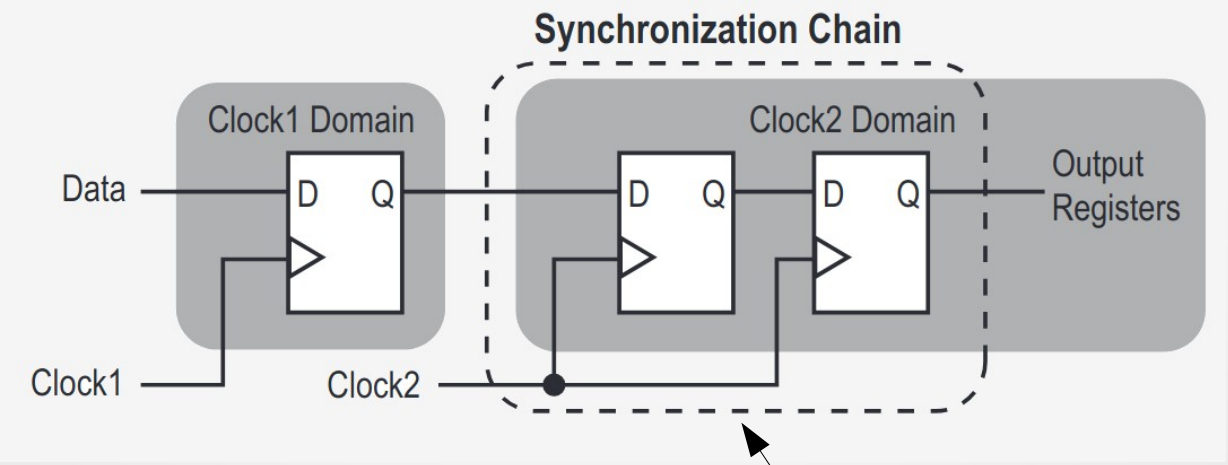
Who is responsible ?

- 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

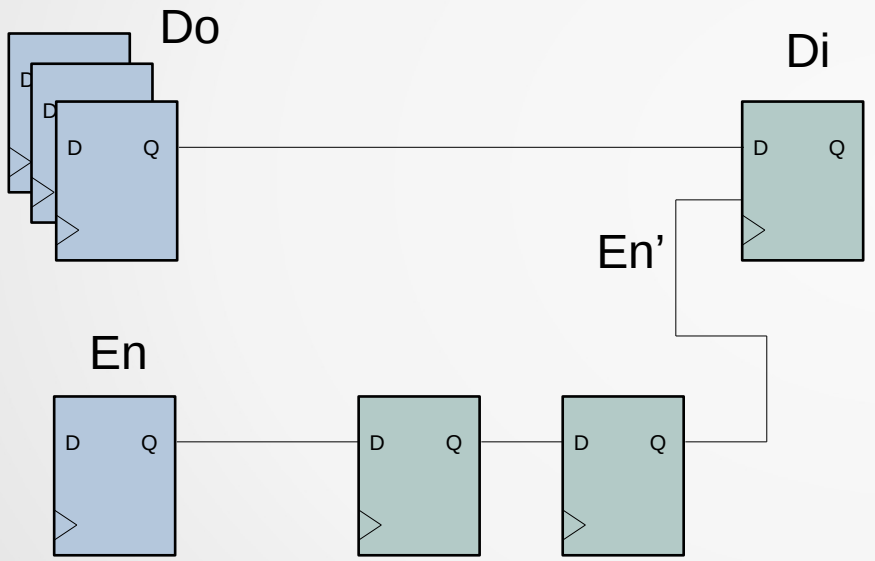
Clock Domain Crossing

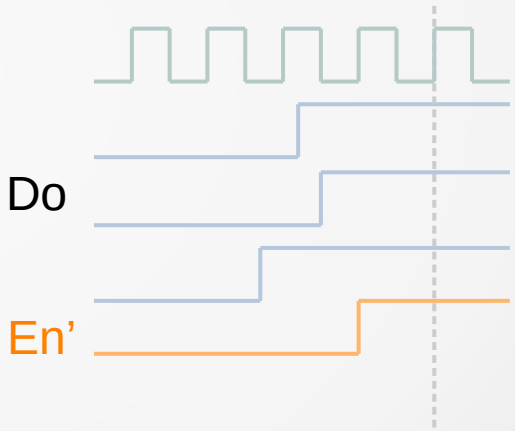
- Used when transferring datas (busses) across clock domain boundaries.

- Two methods:
  - Control based data synchronizers
  - FIFO based data synchronizers

Control based data synchronizers

- The enable signal is responsible to inform the receiving domain that data is stable and ready to be captured.

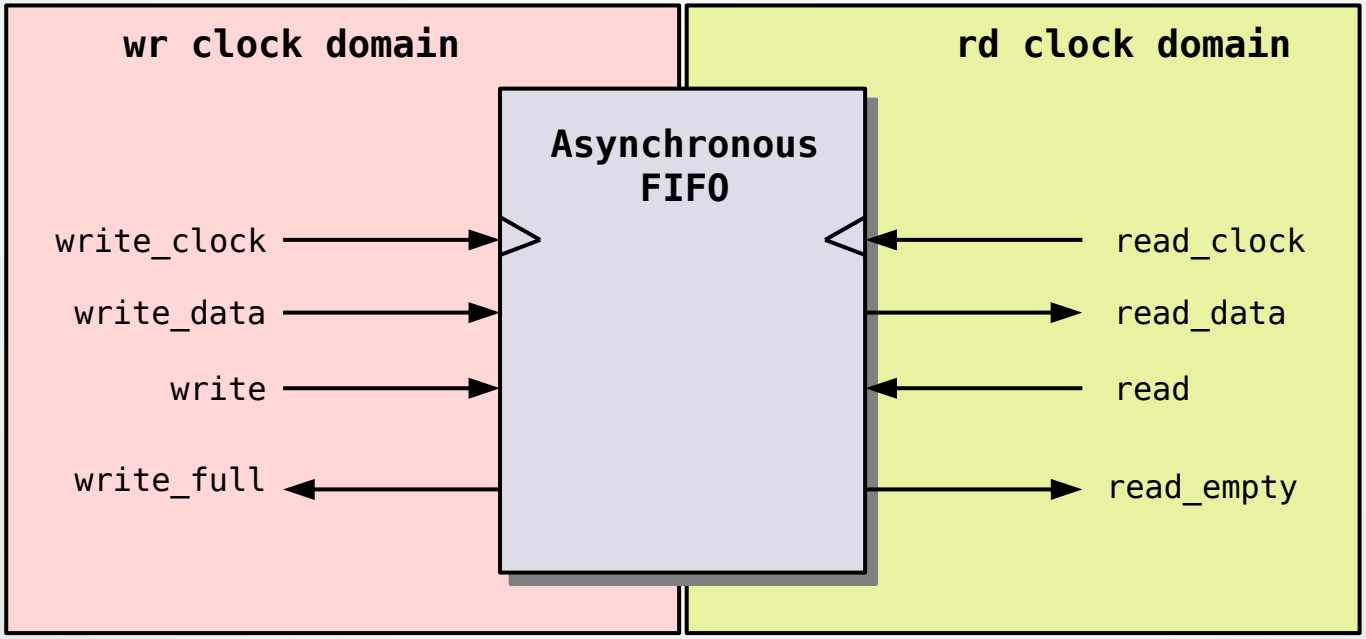




- Control based data synchronizer has limited bandwidth.

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
- LiteX: advanced topics

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

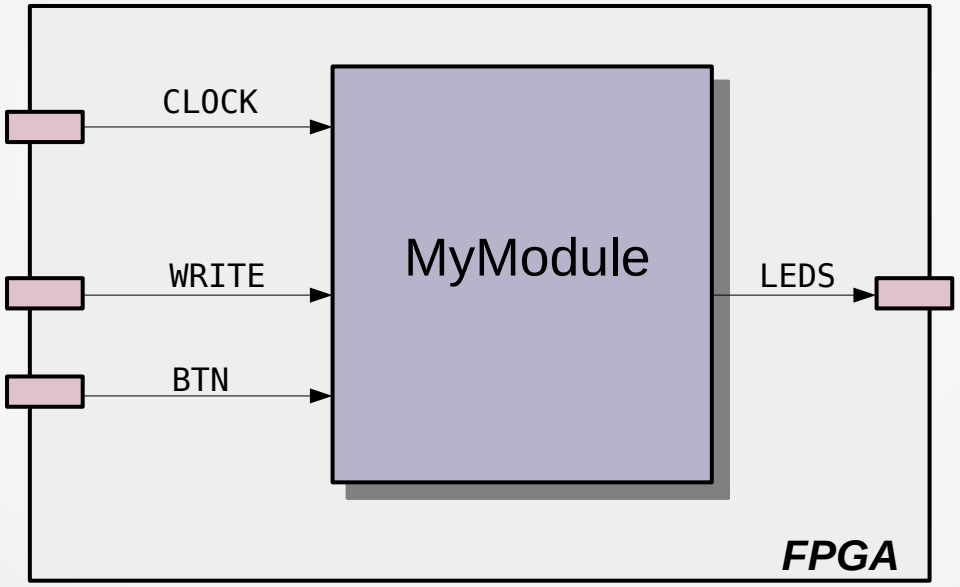
What is Migen

- An alternative HDL based on Python
- FHDL is a Python DSL (Domain Specific Language) defined by Migen and allow generating Verilog or instantiating Verilog/VHDL from Python code
- It basically uses Python to create a **list of combinatorial and synchronous assignments** and **generate a Verilog file** from these assignments.
- Migen has an **integrated simulator** that allows test benches to be written in Python

https://m-labs.hk/gateware/migen

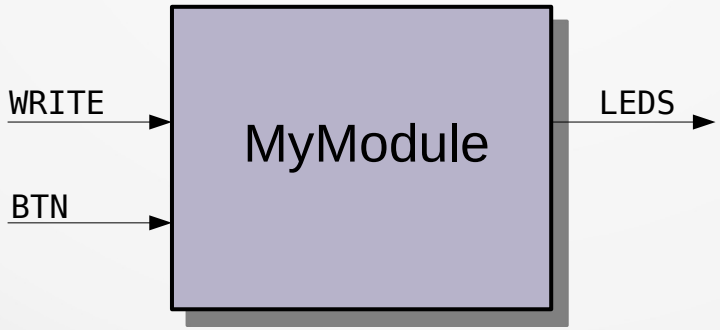
Migen – Concepts

- A **module** (as in verilog) is a block containing a functional description (Migen code) that uses input/outputs



Migen – Concepts

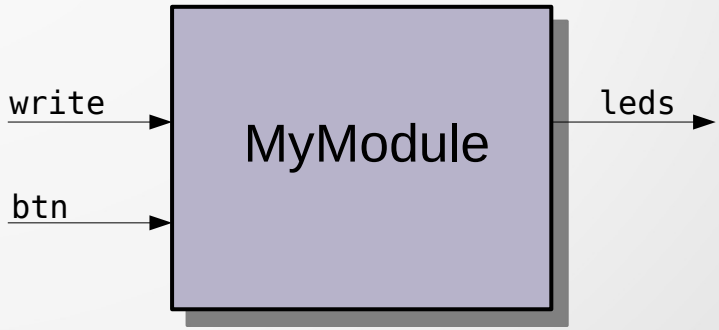
- Migen uses Python classes
- The most important class is **Module**
- A module has input / output signals and parameters
- The direction of signals in the interface is not explicit



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
```

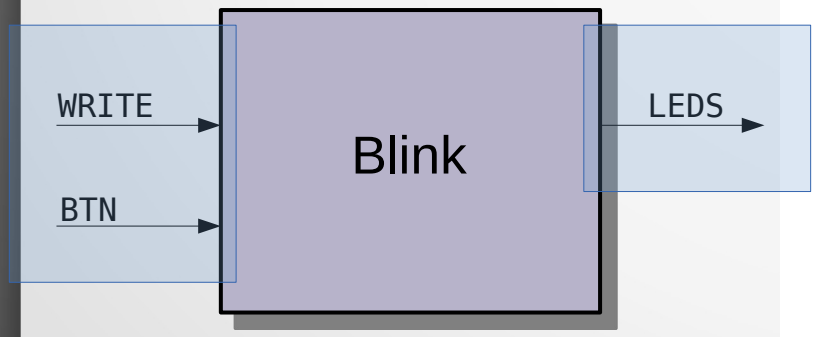
```
self.sync += s1.eq(1)
```

Agenda

- Migen: introduction and workshops
  - Concepts, Modules ans 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 ans 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

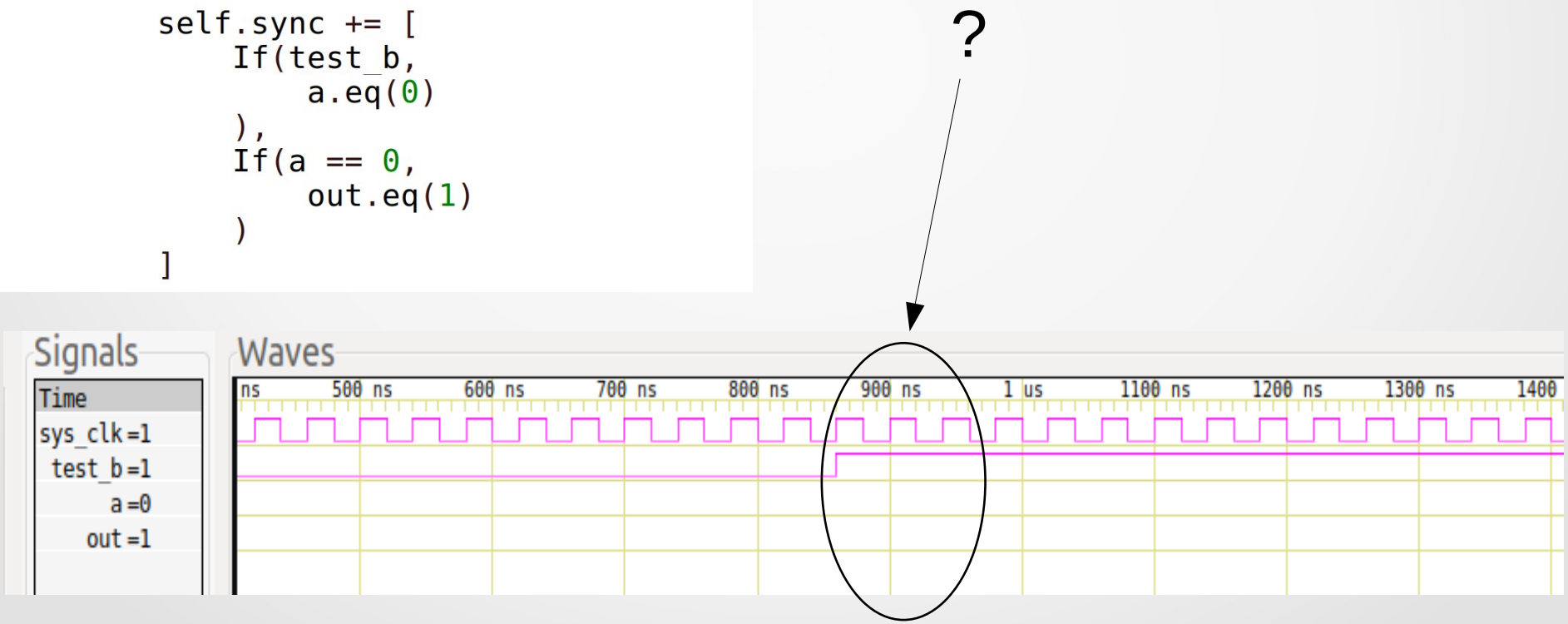
- ***sync*** → a list of synchronous assignments

```
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)
            )
        ]
```

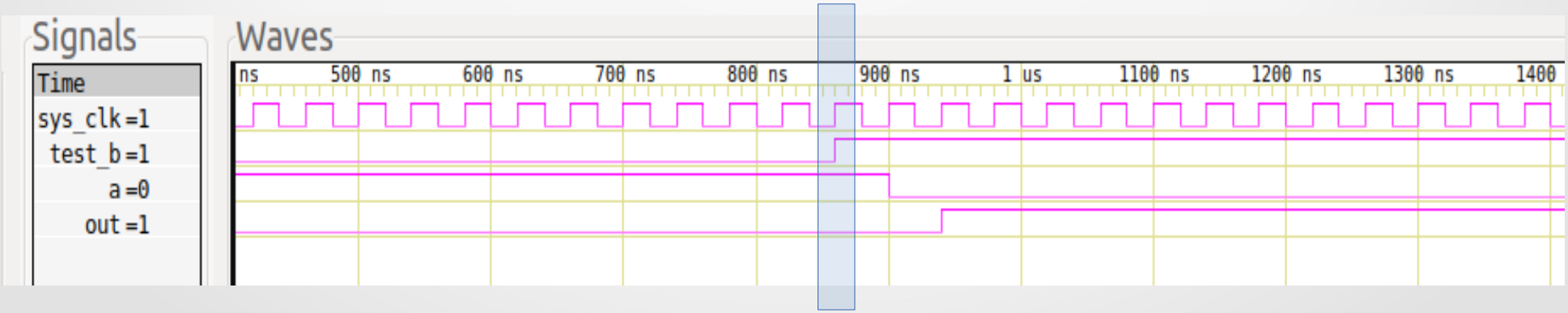


Migen – Attributes of Modules: sync

- ***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



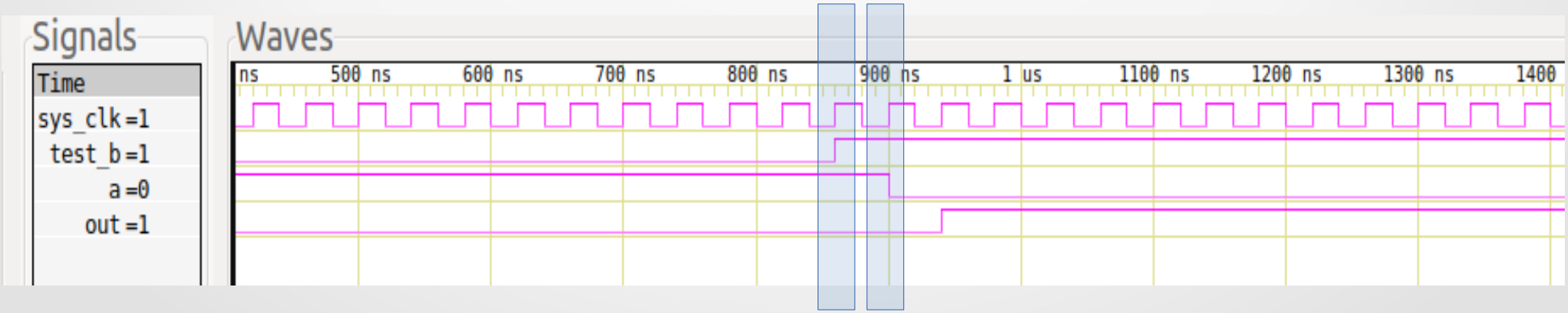
Migen – Attributes of Modules: sync

- ***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



Migen – Attributes of Modules: sync

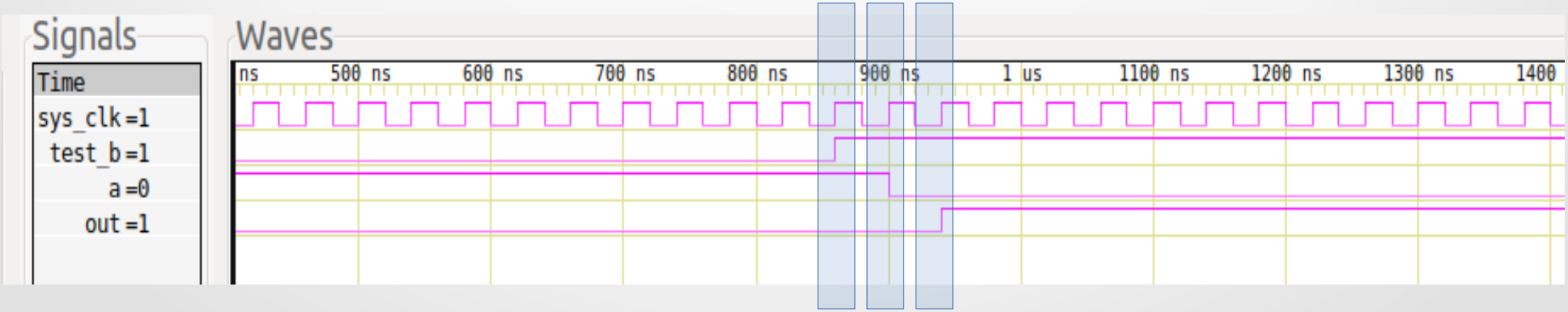
- ***sync*** → a list of synchronous assignments

```
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



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

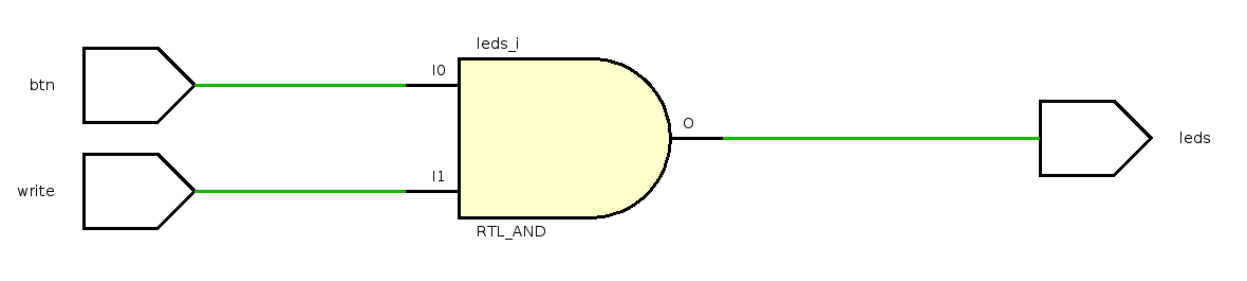
- Migen: introduction and workshops
  - Concepts, Modules ans signals
  - Blinker example
  - Attributes of Module()
  - Example of verilog output
  - Operators (If/Else and FSM)
  - Minimum project requirement (Migen/LiteX)
  - Workshop 1/2
  - Records
  - Simulation
  - Workshop 2/2

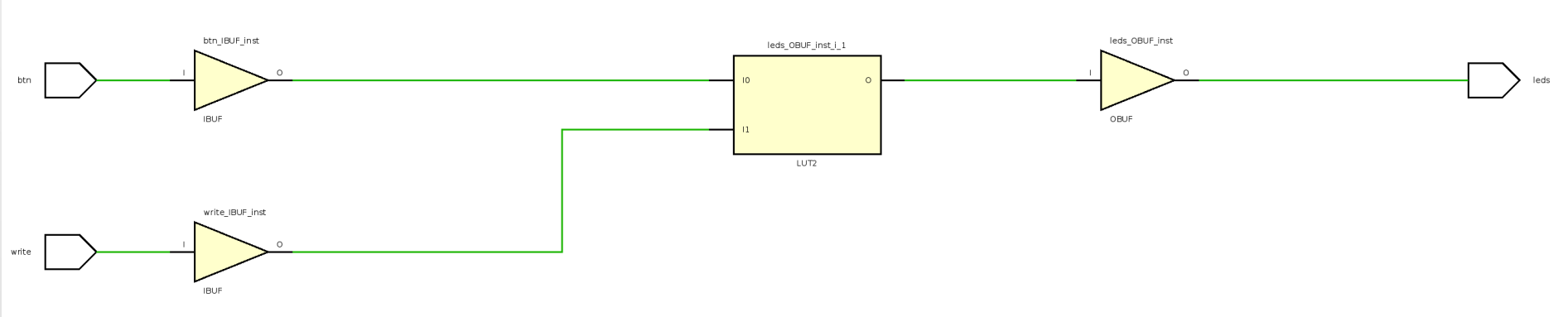
Migen – Combinatorial example


```
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



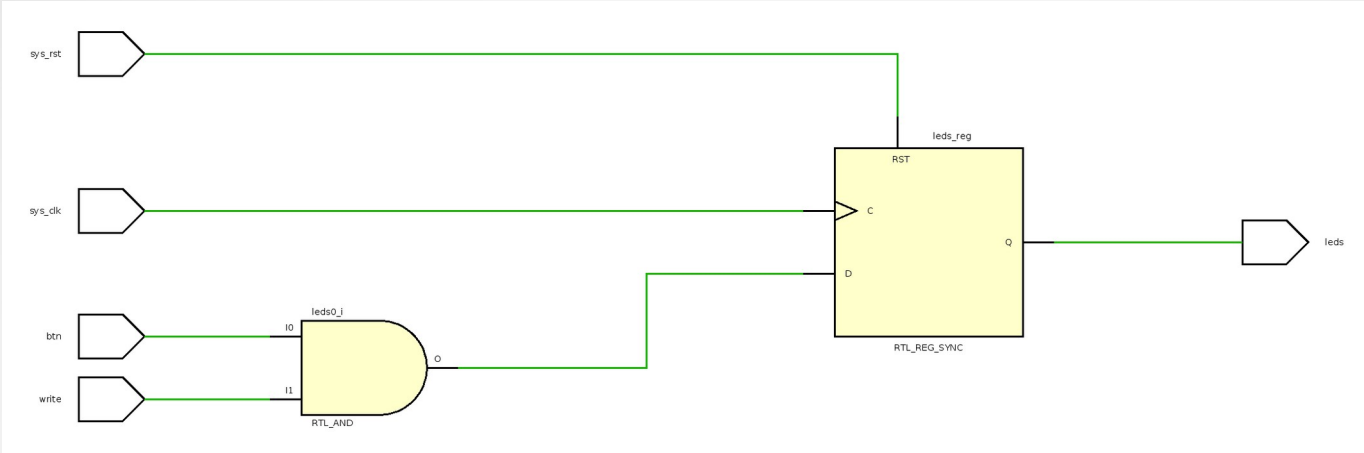


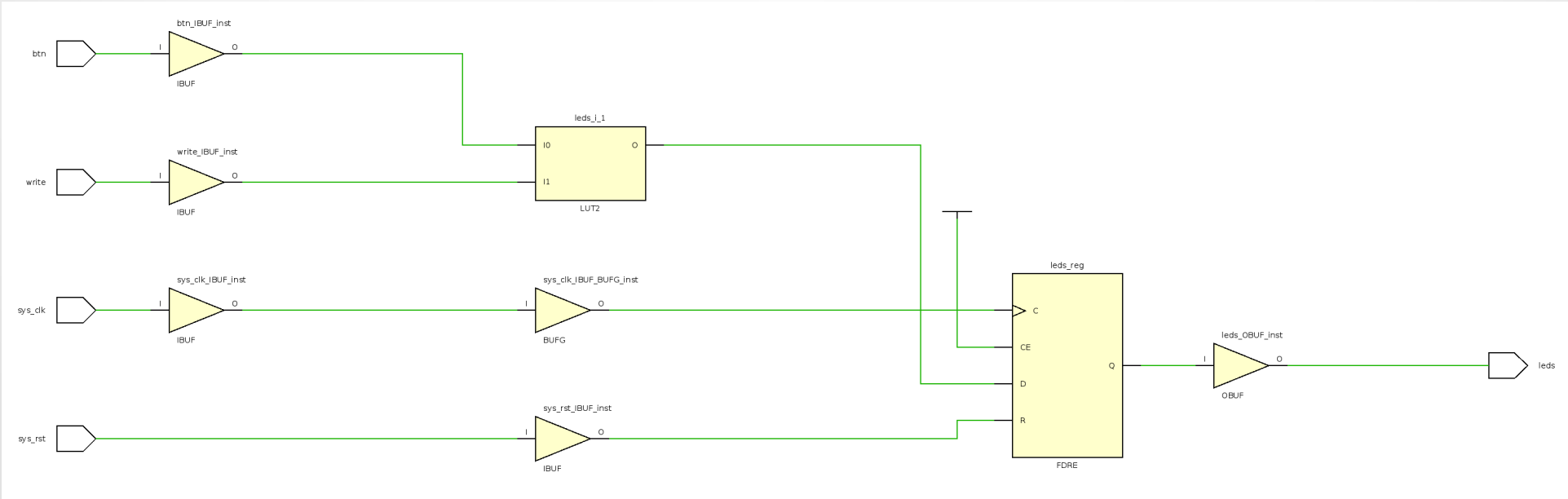
Migen – Synchronous example


```
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)
```

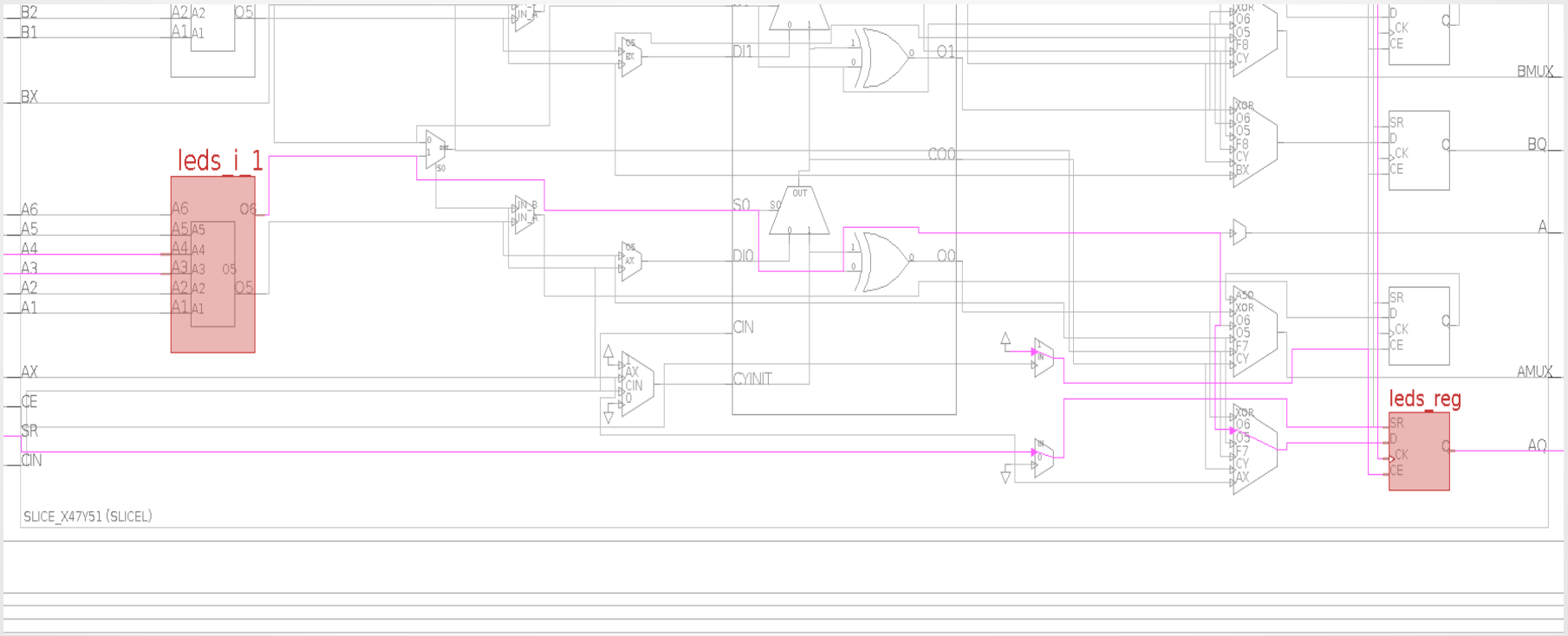
```
module top(
        output reg leds,
        input btn,
        input write,
        input sys_clk,
        input sys_rst
);

always @(posedge sys_clk) begin
        leds <= (btn & write);
        if (sys_rst) begin
                leds <= 1'd0;
        end
end

endmodule
```

Migen – Synchronous example





Migen – Synchronous example



Agenda

- Migen: introduction and workshops
  - Concepts, Modules ans 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

```
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 ans 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_reconfign_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 Tuto(Module):
    def __init__(self, platform):
        # Get pins from ressources
        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 Tuto(Module):
    def __init__(self, platform):
        # Get pins from ressources
        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

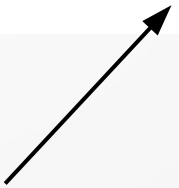
```
class Tuto(Module):
    def __init__(self, platform):
        # Get pins from ressources
        rst = platform.request("user_btn", 0)
        clk = platform.request("sys_clk")

        # 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

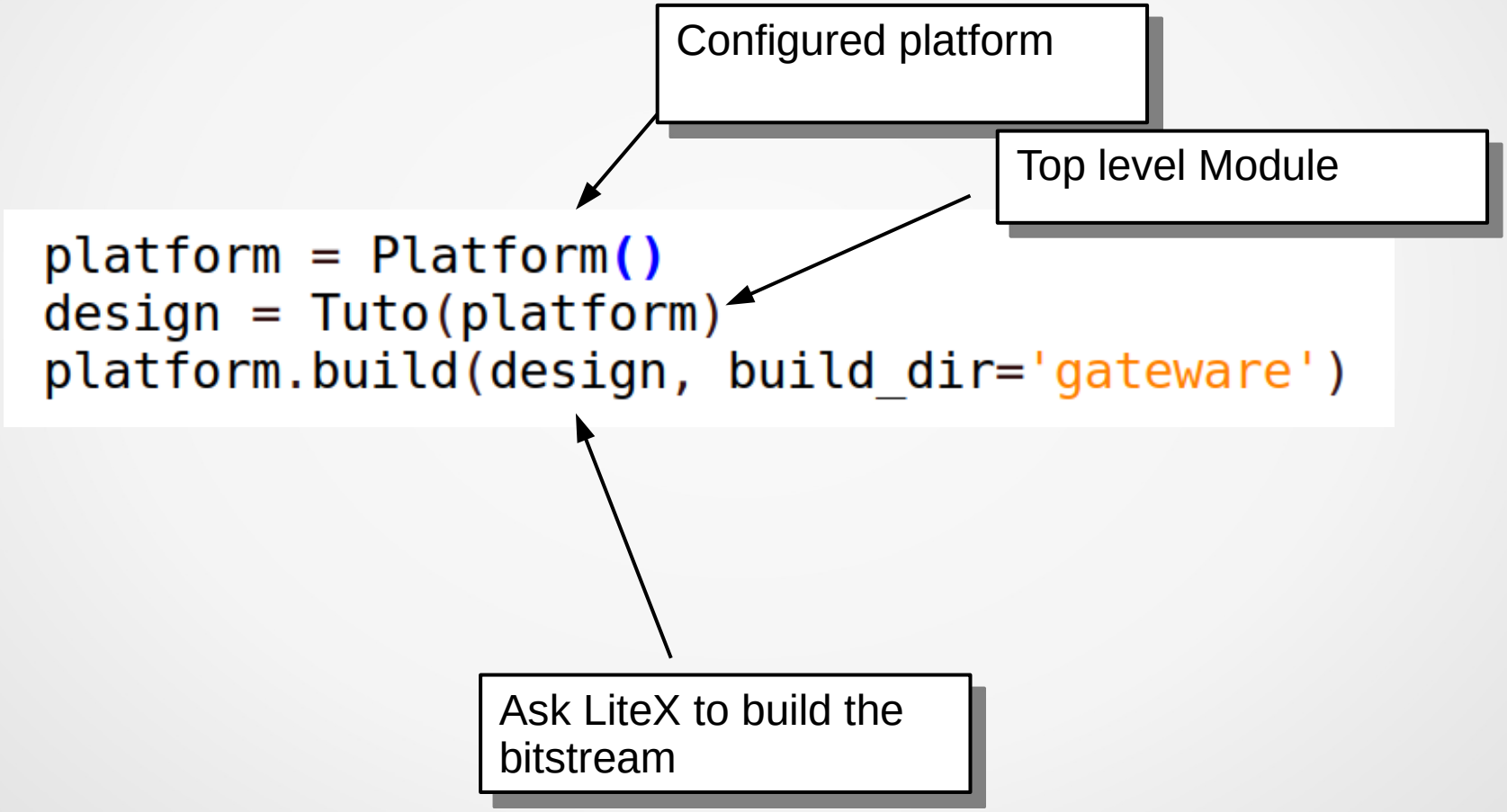
```
# Add a timing constraint
platform.add_period_constraint(clk, 1e9/24e6)
```



This is the requested clock signal

Migen/LiteX – Minimum project code

- Build the bitstream



Migen hands-on

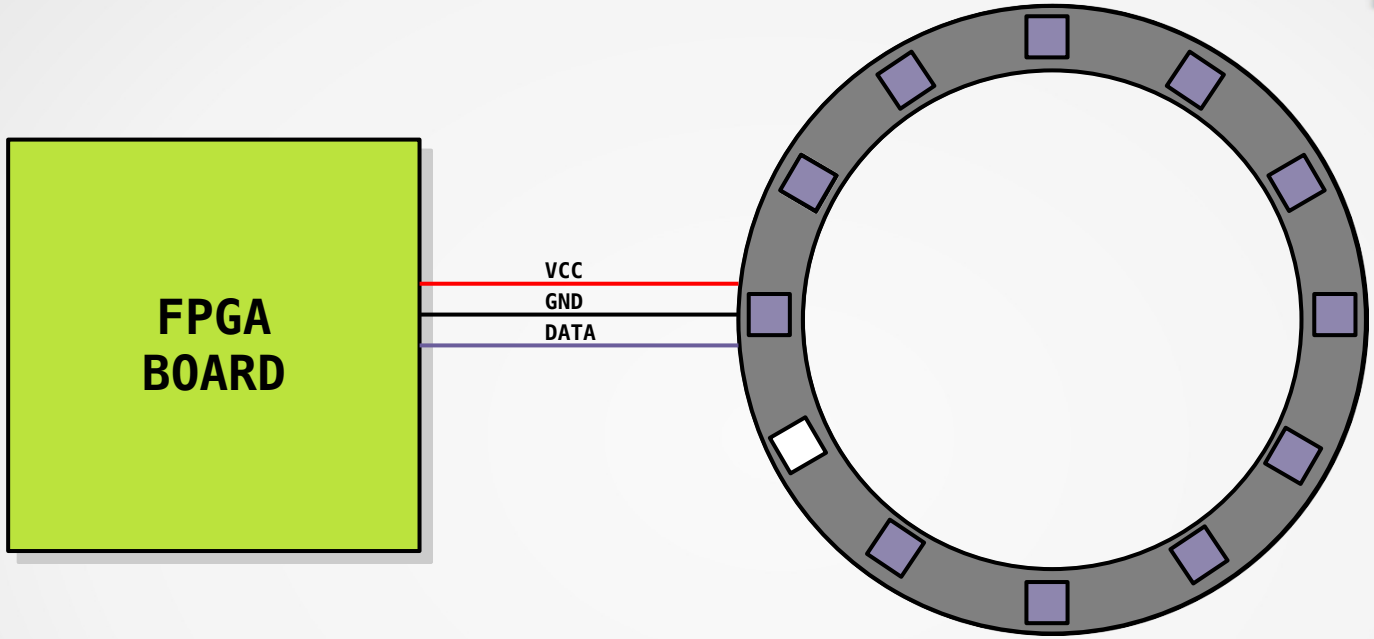
Now, let’s practice !

Step0 – Led blinker

What you'll see:

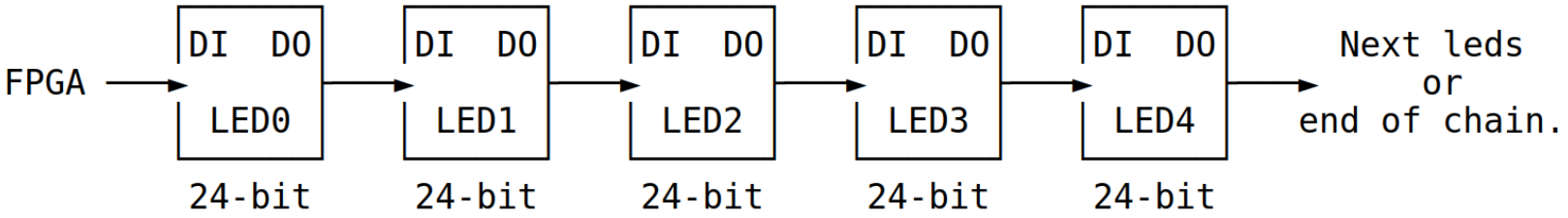
- Platform definition
- Resources assignment
- Submodules
- Simulation
- Build

Step1 – Introduction



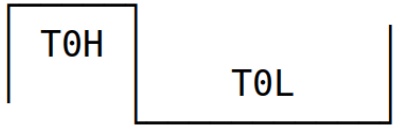
- Addressable LED ring
- Control over a single wire
- 24 bits per LED

Step1 – LED protocol



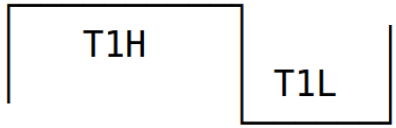
WS2812/NeoPixel Leds are smart RGB Leds controlled over a simple one wire protocol:
- Each Led will "digest" a 24-bit control word: (MSB) G-R-B (LSB).
- Leds can be chained through DIN->DOUT connection.

Each control sequence is separated by a reset code: Line low for > 50us.
Zeros are transmitted as:



T0H = 400ns +-150ns
T0L = 800ns +-150ns

Ones are transmitted as:



T1H = 850ns +-150ns
T1L = 450ns +-150ns

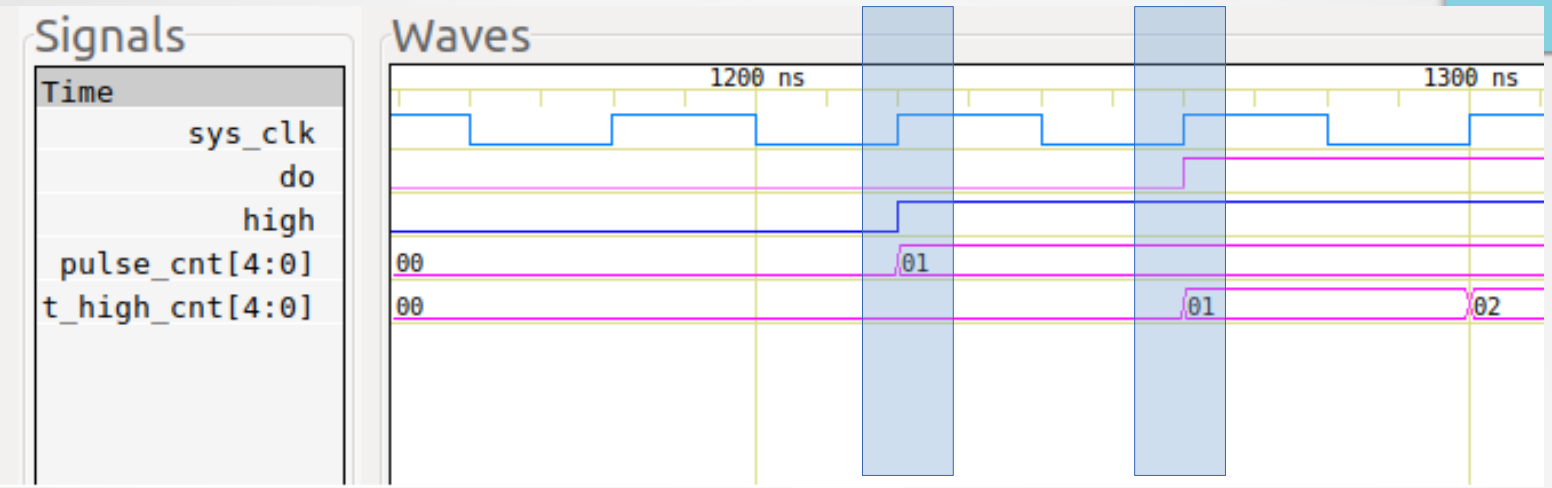
Step1 – Instructions

- Control the first LED
- Send 24 “Ones” pulses (equivalent to 0xFFFFFF, white color)
- Test with ./workshop_step1.py sim
- gtkwave sim.vcd &
- CTRL-R in gtkwave to update waveform after a new simulation

Step1 – Tip

```
self.sync += [
   if (pulse_cnt  < 24) {
       if (pulse_high) {
           output = 1
           cnt_high = cnt_high + 1
           if (cnt_high == t1h) {
               pulse_high = 0
               cnt_high = 0
           }
       } else {
           output = 0
           cnt_low = cnt_low + 1
           if (cnt_low == t1l) {
               pulse_high = 1
               cnt_low = 0
               pulse_cnt = pulse_cnt + 1
           }
       }
    }
```

Step1 – Observations

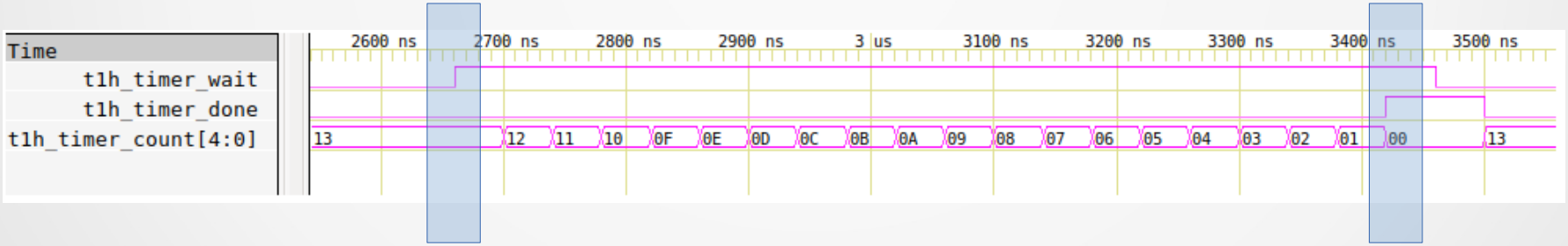



```
self.sync += [
    If(pulse_cnt < 24,
        If(high,
            self.do.eq(1),
            t_high_cnt.eq(t_high_cnt + 1),
            If(t_high_cnt == 19,
                t_high_cnt.eq(0),
```

- **Synchronous assignments take effect on the next cycle**
- If a signal is assigned multiple time in the same clock cycle, the last assignment is taken into account

Step2 – Instructions

- Simplify code from step1 using ***WaitTimer*** module from migen/genlib
- timer = WaitTimer(period)
  - Period is expressed in clock cycles
  - Two control signals: *wait* and *done*



- Compute timers period from time and frequency. Pure Python code can be used here.

Step2 - Observations

- **Don’t forget to add your module to submodules. Migen won’t complain !**

- Pure Python code can be used in Migen modules (configuration, genericity,...)

Step3 - Instructions

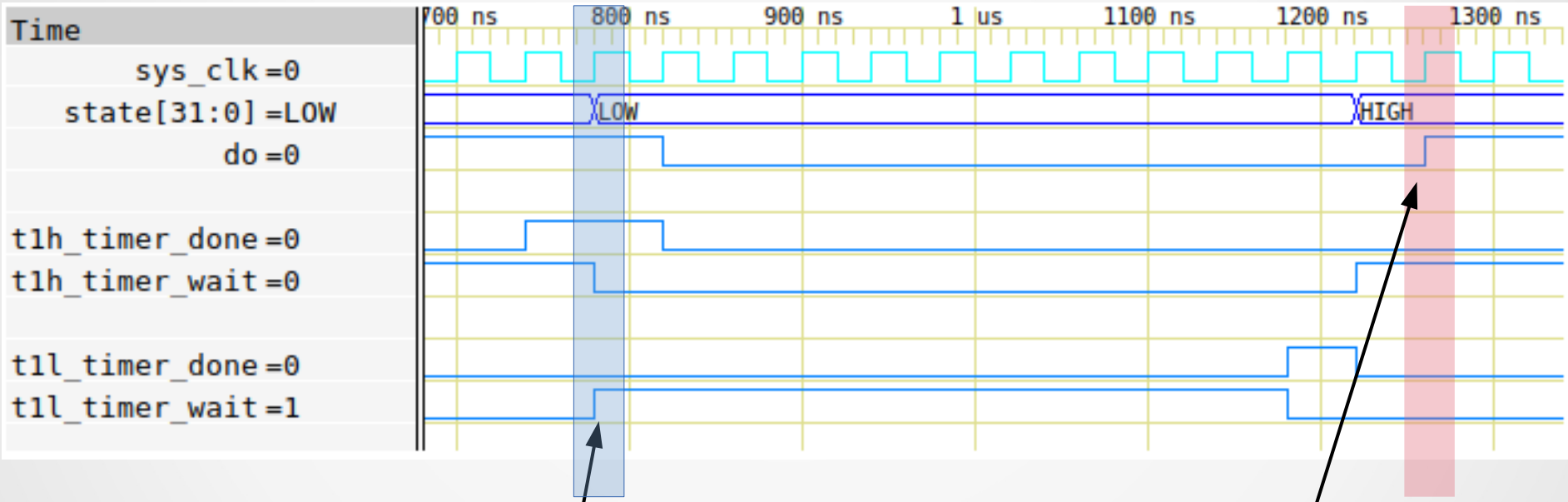
- Use Finite State Machine (FSM) to simplify your code

```
# Complete this FSM
self.submodules.fsm = fsm = FSM(reset_state="HIGH")
fsm.act("HIGH",
    # Your code here
)

fsm.act("LOW",
    # Your code here
)

fsm.act("RESET",
    # Your code here
)
```

- ***NextState(state)*** selects the next state
- ***NextValue(a, b)*** is equivalent to self.sync += a.eq(b) when the FSM is in the given state.
- ***a.eq(b)*** is equivalent to self.comb += a.eq(b) when the FSM is in the given state. When it's not, a.eq(0)

Step3 - Observations

- Direct assignment in FSM are combinatorial

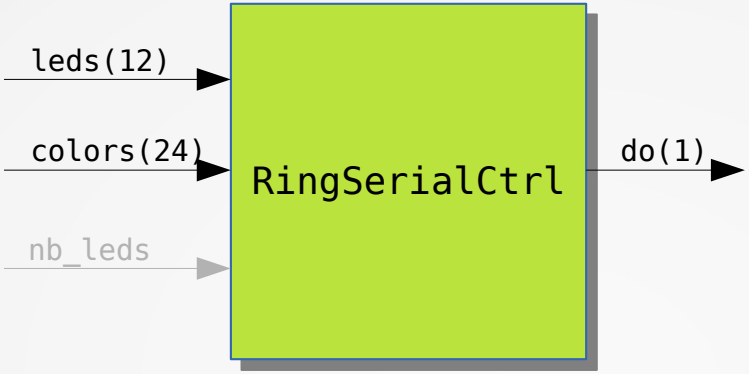



```
fsm.act("LOW",
    NextValue(self.do, 0),
    t1l_timer.wait.eq(1),
    If(t1l_timer.done,
        NextValue(pulse_cnt, pulse_cnt + 1),
        NextState("HIGH"),
```

```
fsm.act("HIGH",
    NextValue(self.do, 1),
    t1h_timer.wait.eq(1),
    If(t1h_timer.done,
        NextState("LOW"),
    )
)
```

Step4 - Instructions

- Design a RingSerialCtrl module



- leds is a 12 bits input, each bit controls a led (on/off)
- colors is a 24 bits input that controls the ring color
- nb_leds is a parameters to configure how much LED the ring has

Note:
You can access individual bits in a Signal() using Python indexes and slices:

- `led[1]` is the second bit of led,
- `led[-1]` is the MSB,
- `led[0:3]` are the 3 lower bits of led.

Step4 - Tip

```
fsm.act("RST",
            # wait for trst
            # when done, init variables and go to LED-SHIFT
        )
fsm.act("LED-SHIFT",
            # init bit counter
            # increment led_count counter
            # shift led pattern
            # check if led should be lit (assign 0 or color)
            # if next led go to BIT-TEST else RST
        )
fsm.act("BIT-TEST",
            # if data(MSB) == 1 go to ONE_SEND
            # else go to ZERO_SEND
            # data = data << 1

        )
fsm.act("ZERO-SEND",
            # send bit zero pattern (timer)
            # go to BIT-SHIFT

        )
fsm.act("ONE-SEND",
            # send bit one pattern (timer)
            # go to BIT-SHIFT
        )
fsm.act("BIT-SHIFT",
            # shift color data
            # check if 24 bits sent
            # if yes go to LED-SHIFT
            # else go to BIT-TEST
        )
```

Step4 - Observations

- You can access individual bits in a Signal using Python indexes and slices:

  - led[1] is the second bit of leds,
  - led[-1] is the MSB,
  - **led[0:3] are the 3 lower leds bits. Bit 3 is excluded !**

Different with V*HDL where bit vectors are represented from MSB to LSB:

  - my_vhdl_signal(11 downto 0)
  - my_verilog_signal[11:0]

Step5 - Instructions

- Create a new module RingControl to control RingSerialCtrl and make LEDs spins

- Put both modules in a separate file named ring.py

Bonus1:

- Use Array([a, b, …]) to light LED following the pattern defined in this array of values

Bonus2:

- Add a build time option to use simple LED spin or array mode (using pure Python syntax)

Step5 - Tips

Python statement can be used inside Modules:

```
if (something == True):
    self.comb += out.eq(test1)
else:
    self.comb += out.eq(test2)
```

Step5 - Observation

- Organize your files as much as possible
- Use Python to configure your design

Agenda

- Description of FPGAs
- Digital design challenges
- Migen: introduction and workshops
  - Records
  - Simulation
- LiteX: introduction and workshops
- LiteX: advanced topics

Migen/LiteX – Records

- Records are structures of Signal() objects
- Records are described with a layout (list of tuples)

```
layout1 = [("a", 1), ("b", 4)]
layout2 = [("c", layout1), ("d", 8)]


rec = Record(layout1)
self.sync += m.b.eq(out)

new = Record(layout2)
self.sync += new.c.b.eq(out)
```

```
class HyperRAMX2(Module):
    def __init__(self, pads, latency = 6):
        ...

        self.dly_io  = Record([("loadn", 1),("move", 1),("direction", 1)])
        self.dly_clk = Record([("loadn", 1),("move", 1),("direction", 1)])

        ...
```

Migen/LiteX – Records

- IO Resources can be Records (often, they are)
- Subsignal is used

```
_io = [
    # Clk / Rst
    ("sys_clk",   0, Pins("35"), IOStandard("LVCMOS33")),

    ...

    # Serial
    ("serial", 0,
        Subsignal("tx", Pins("8")),
        Subsignal("rx", Pins("9")),
        IOStandard("LVCMOS33")
    ),
]
```

```
ser = platform.request("serial")
print(type(ser))
print(ser)
print(ser.flatten())
```

```
<class 'migen.genlib.record.Record'>
<Record tx:rx at 0x7fc0500967f0>
[<Signal serial_tx at 0x7fc050096880>, <Signal serial_rx at 0x7fc050096970>]
```

```
self.comb += ser.tx.eq(temp1)
```

Migen/LiteX – Records

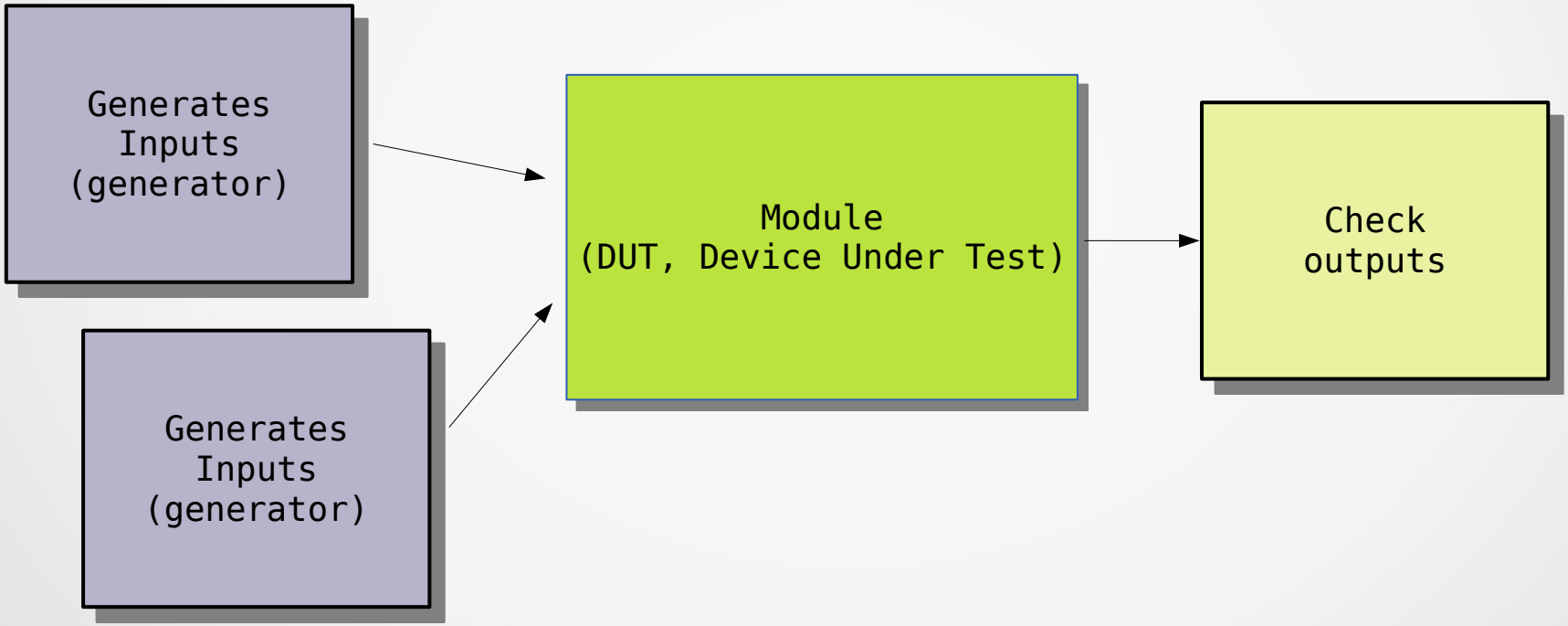
- Signals of a Record are attributes of it
- Testing attributes can be part of the configuration

```
class HyperRAM(Module):
    def __init__(self, pads, latency=6):
        self.pads = pads
        ...

        # # #

        clk       = Signal()
        clk_phase = Signal(2)
        cs        = Signal()
        ca        = Signal(48)
        ca_active = Signal()
        sr        = Signal(48)
        dq        = self.add_tristate(pads.dq)   if not hasattr(pads.dq,   "oe") else pads.dq
        rwds      = self.add_tristate(pads.rwds) if not hasattr(pads.rwds, "oe") else pads.rwds
        dw        = len(pads.dq)                 if not hasattr(pads.dq,   "oe") else len(pads.dq.o)

        assert dw in [8, 16]
```

Agenda

- Description of FPGAs
- Digital design challenges
- Migen: introduction and workshops
  - Records
  - Simulation
- LiteX: introduction and workshops
- LiteX: advanced topics

Migen/LiteX – Simulation



Migen/LiteX – Simulation

- Migen has an integrated simulator
- Test benches (generators) execute concurrently
- Use ***yield*** to communicate with the simulator. There are four basic patterns:
  - Reads: state of a signal can be read using **(yield signal)**
  - Writes: state of a signal after next clock is set with **yield signal.eq(value)**
  - Clocking: simulation can be advanced one clock cycle using **yield**
  - Composition: control can be transferred to another function using **yield from run_other()**
- Run with **run_simulation(dut, bench)** where dut is the module under test and bench are the generators functions.
- Can generate a VCD file containing a dump of the signals inside dut

Migen/LiteX – Simulation

Module under test

```
class ORGate(Module):
def __init__(self):
  self.a = Signal()
  self.b = Signal()
  self.x = Signal()

  ###

  self.comb += self.x.eq(self.a | self.b)
```

```
dut = ORGate()
```

Test Bench (generator)

```
def testbench():
  yield dut.a.eq(0)
  yield dut.b.eq(0)
  yield
  assert (yield dut.x) == 0

  yield dut.a.eq(0)
  yield dut.b.eq(1)
  yield
  assert (yield dut.x) == 1
```

```
run_simulation(dut, testbench())
```

Simulation

Migen/LiteX – Simulation

Module under test

```
class ORGate(Module):
def __init__(self):
  self.a = Signal()
  self.b = Signal()
  self.x = Signal()

  ###

  self.comb += self.x.eq(self.a | self.b)
```

```
dut = ORGate()
```

```
def check_case(a, b, x):
  yield dut.a.eq(a)
  yield dut.b.eq(b)
  yield
  assert (yield dut.x) == x
```

subroutine

```
def testbench():
  yield from check_case(0, 0, 0)
  yield from check_case(0, 1, 1)
  yield from check_case(1, 0, 1)
  yield from check_case(1, 1, 1)
```

Test Bench (generator)

```
run_simulation(dut, testbench())
```

Simulation

Migen/LiteX – Simulation

- Multiple generators can run in parallel
- Can be multiple clock domains
- Don’t forget ***yield, yield from*** (Migen won’t complain)
- Signals must not be driven concurrently

```
def main():
        ring = RingSerialCtrl(4, 24e6)

        generators = {
            "sys" : [ change_nb_led_and_color(ring),
                      control_out(ring),
                      detect_reset(ring),
                    ]
        }

        run_simulation(ring, generators, clocks={"sys": 1e9/24e6}, vcd_name="sim.vcd")
```

Step6 – Write a testbench for RingSerialCtrl

What you'll learn:

- Use generators
- Write complex test benches

Step6 – Write a testbench for RingSerialCtrl

- Write a generator to set a random color to a random LED
- Write a generator to detect the timeout condition
- Write a generator to print which value is set on each LED

```
$ ./test_ring_step6.py

Set      LED1 = 0x5c8035

Detected LED0 = 0x0
Detected LED1 = 0x5c8035
Detected LED2 = 0x0
Detected LED3 = 0x0

Set      LED3 = 0xb1b772

Detected LED0 = 0x0
Detected LED1 = 0x0
Detected LED2 = 0x0
Detected LED3 = 0xb1b772

...
```

Agenda

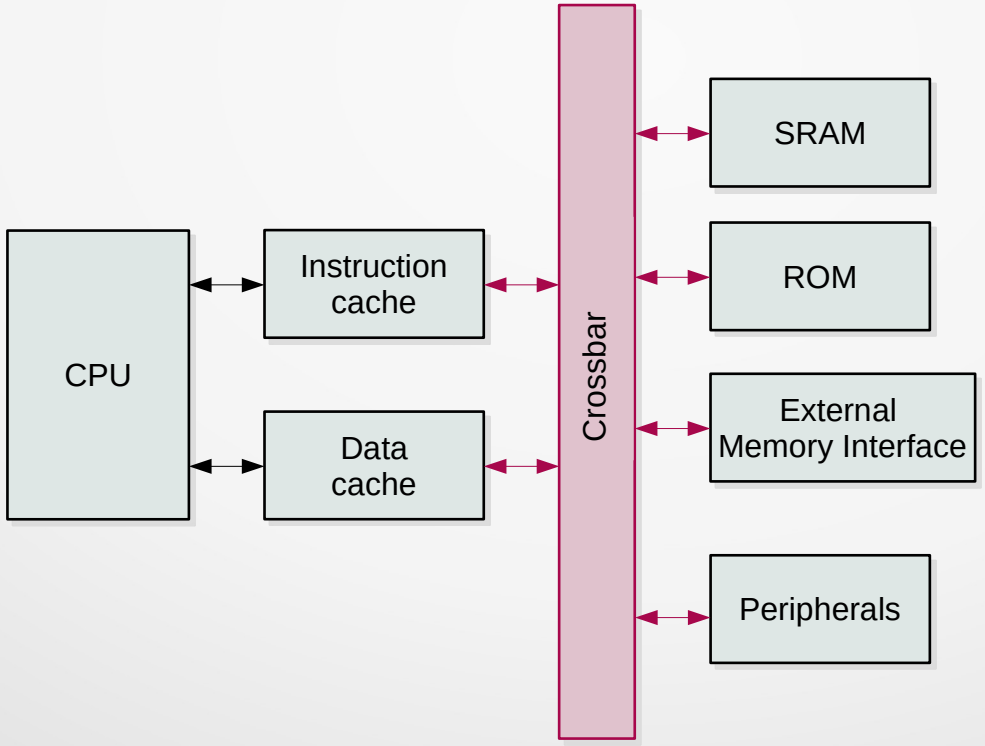
- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
  - Presentation of SoCs and LiteX
  - Examples of peripherals integration
  - How to build a SoC
  - Software in LiteX
  - LiteX tools
  - Workshop
- Litex: advanced topics

System On Chip - SoC

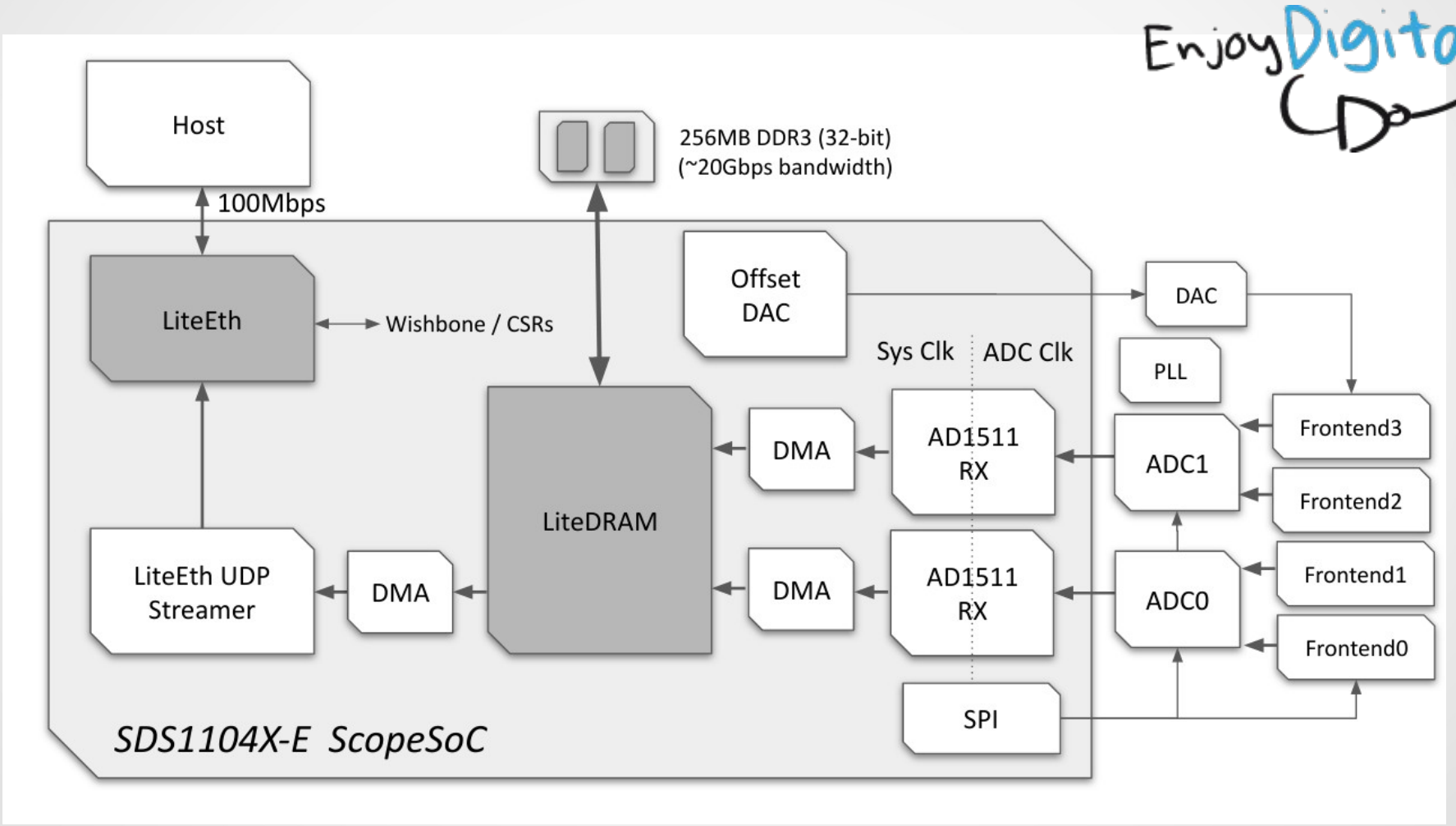
System on a chip

From Wikipedia, the free encyclopedia

A **system on a chip** (**SoC**; /ˌɛsˌoʊˈsiː/ *es-oh-SEE* or /sɒk/ *sock*[nb 1]) is an integrated circuit (also known as a "chip") that integrates all or most components of a computer or other electronic system.

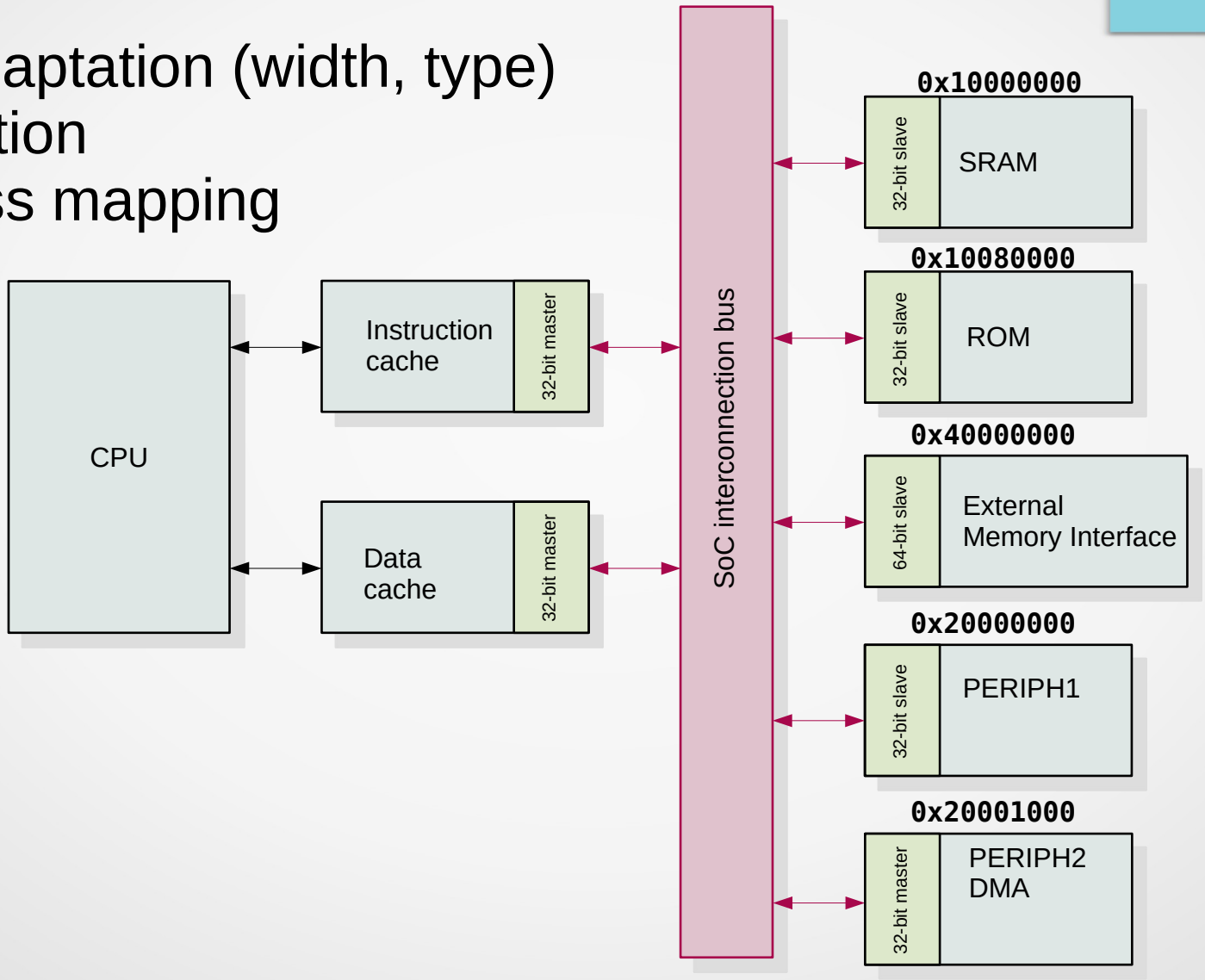


System On Chip - SoC

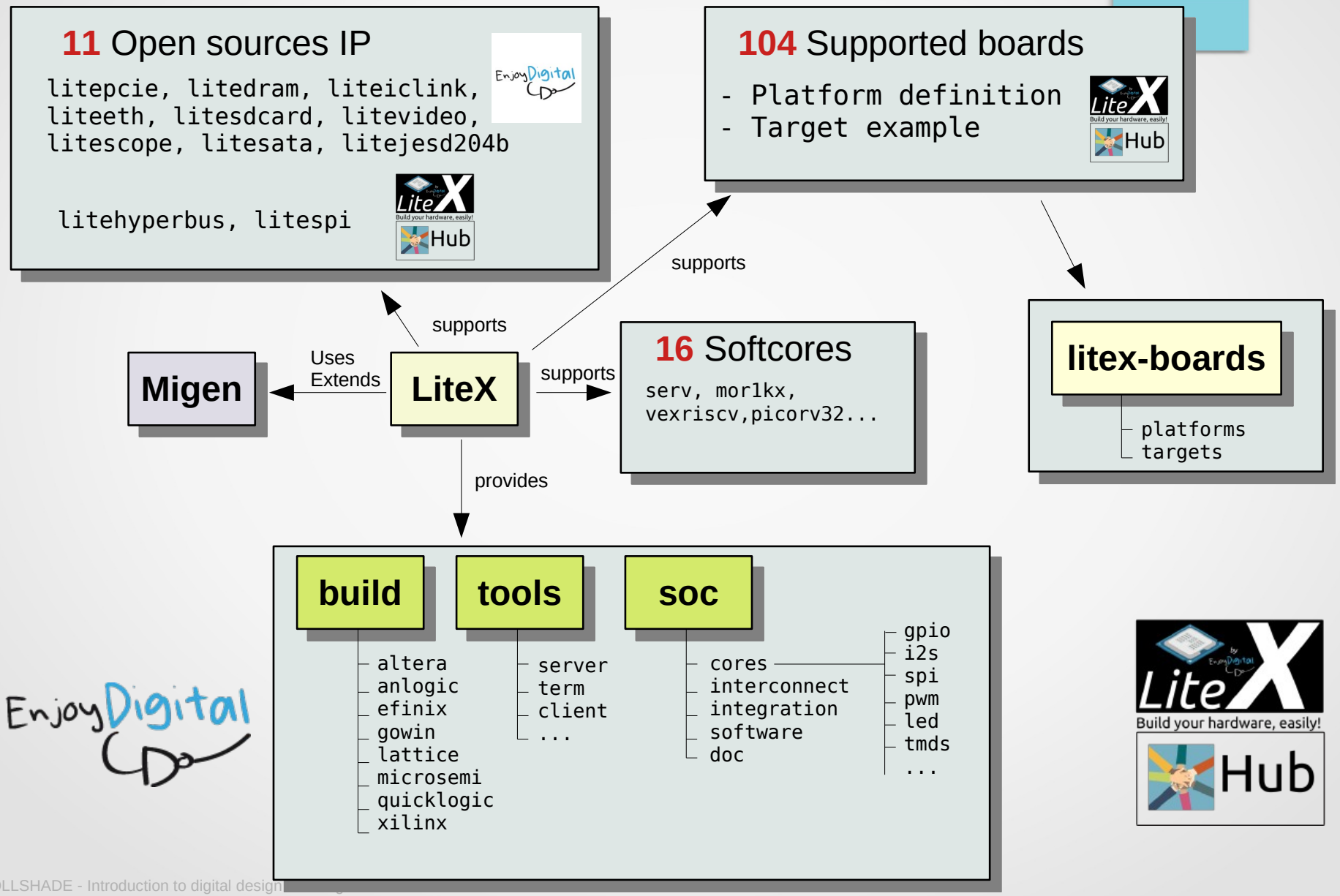


System On Chip - SoC

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



What is LiteX



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 – Busses

- SoC interconnections are made with **Wishbone** buses (open sources standard). It can be configured to use **AXI-Lite** (AXI is a royalty free protocol available from ARM)
- **CSR** (Control and Status Registers) bus is a simple bus protocol used to handle low bandwidth transactions
- **Litex streams** is an interface to connect streaming components (data flow exchange)
- Bridges are available to interconnect all supported bus
- The CSR bus is automatically bridged to the Wishbone address space

LiteX – SoC classes

`SoC(Module)→LiteXSoC→SoCCore→SoCMini→LiteXCore`

- SoC is where busses, RAM, ROM, CPU and timer are added (via methods) as submodules,

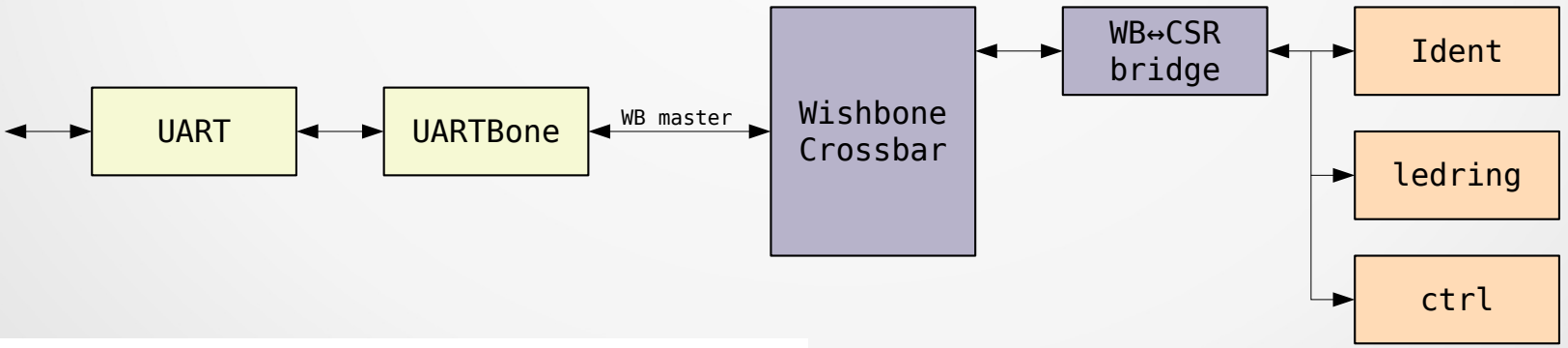
- LiteXSoc has a set of methods to add features to SoCCore: *add_identifier*, *add_uart*, *add_sdram*, *add_ethernet*,…

- **SoCCore** takes a set of arguments that defines a SoC based on LiteXSoc and provides methods to extends this SoC. **This is the class that you might use to create a SoC.**

- **SoCMini** is a version of SoCCore with minimum features enabled (by default: no CPU, no RAM, no UART, no TIMER)

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,,
```

LiteX – SoCCore

- Configuration of core functions with arguments

```
def __init__(self, platform, clk_freq,
    # Bus parameters
    bus_standard             = "wishbone",
    ...
    # CPU parameters
    cpu_type                 = "vexriscv",
    ...
    # ROM parameters
    integrated_rom_size      = 0,
    ...
    # SRAM parameters
    integrated_sram_size     = 0x2000,
    ...
    # Identifier parameters
    ident                    = "",
    ...
    # UART parameters
    with_uart                = True,
    ...
    # Timer parameters
    with_timer               = True,
    ...
    # Controller parameters
    with_ctrl                = True,
```

```
# SoCCore ----------------------------------------------
SoCCore.__init__(self, platform, sys_clk_freq,
    ident          = "LiteX SoC on DE10-Lite",
    ident_version  = True,
    **kwargs)
```

LiteX – SoCCore

- Add peripherals with methods from SoCCore:
  - `add_csr`
  - `add_wb_master`
  - `add_wb_slave`
- Add peripherals with methods from LiteXSoc:
  - `add_spi_flash`
  - `add_sdram`
  - `add_ethernet`
  - ….

Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- Litex: introduction and workshops
  - Presentation of SoCs and LiteX
  - Examples of peripherals integration
  - How to build a SoC
  - Software in LiteX
  - LiteX tools
  - Workshop
- Litex: advanced topics

LiteX – SDRAM

- Add DRAM memory to the system
- Use LiteDRAM and supports SDR, DDR2, DDR3, DDR4 and LPDDR
- Needs a PHY module
- Signals have to be named a certain way

```
("ddram", 0,
    Subsignal("a", Pins("J7 J6 H5 L7 F3 H4 H3 H6 D2 D1 F4 D3 G6")),
    Subsignal("ba", Pins("F2 F1")),
    Subsignal("cke", Pins("H7")),
    Subsignal("ras_n", Pins("L5")),
    Subsignal("cas_n", Pins("K5")),
    ....
```

```
self.submodules.ddrphy = s7ddrphy.K7DDRPHY(platform.request("ddram"),
    memtype      = "DDR3",
    nphases      = 4,
    sys_clk_freq = sys_clk_freq)
self.add_sdram("sdram",
    phy           = self.ddrphy,
    module        = H5TC4G63CFR(sys_clk_freq, "1:4"),
    l2_cache_size = kwargs.get("l2_size", 8192)
)
```

LiteX – Ethernet

- Add Ethernet to the system
- Use LiteETH and supports MII, RMII, GMII, RGMII, 1000BASEX, XGMII
- Needs a PHY module
- Signals have to be named a certain way

```
("eth clocks", 0,
    Subsignal("tx",  Pins("M28")),
    Subsignal("gtx", Pins("K30")),
    Subsignal("rx",  Pins("U27")),
    IOStandard("LVCMOS25")
),
("eth", 0,
    Subsignal("rst n",   Pins("L20")),
    Subsignal("int n",   Pins("N30")),
    Subsignal("mdio",    Pins("J21")),
    Subsignal("mdc",     Pins("R23")),
    ....
```

```
self.submodules.ethphy = LiteEthPHY(
    clock_pads = self.platform.request("eth_clocks"),
    pads       = self.platform.request("eth"),
    clk_freq   = self.clk_freq)
self.add_ethernet(phy=self.ethphy)
```

LiteX – Others

- add_spi_flash
- add_spi_sdcard
- add_sdcard
- add_sata
- add_pcie
- add_video_colorbars
- add_video_terminal
- add_video_framebuffer

- Once again, there is no documentation :(

Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
  - Presentation of SoCs and LiteX
  - Examples of peripherals integration
  - How to build a SoC
  - Software in LiteX
  - LiteX tools
  - Workshop
- LiteX: advanced topics

LiteX – Add a SoC to the project

- Create a class that inherits from SoCCore or SoCMini
- Set parameters
- **Don’t forget to add a .crg submodule !**

```
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,
            with_uart            = True,
            cpu_type             = "serv",
            integrated_sram_size = 0x1000,
            integrated_rom_size  = 0x10000,
            with_timer           = True)

        # CRG --------------------------------------------------
        self.submodules.crg = CRG(platform)
```

- All other submodules will be added in this class

LiteX – Build a SoC

- Add your own arguments (used locally)

```
def main():
    # Board specific arguments
    parser = argparse.ArgumentParser(description="LiteX SoC on Tang Nano")
    parser.add_argument("--build",       action="store_true", help="Build bitstream")
    parser.add_argument("--load",        action="store_true", help="Load bitstream")
    parser.add_argument("--flash",       action="store_true", help="Flash Bitstream")

    # Builder adds its own arguments
    builder_args(parser)

    # SoCCore adds its own arguments
    soc_core_args(parser)

    # args attributes are set with parsed arguments
    args = parser.parse_args()

    # soc_core_argdict filters and adapt arguments passed
    # to SoCCore
    soc = BaseSoC(
        sys_clk_freq      = 24e6,
        **soc_core_argdict(args)
    )

    # Filter and adapt arguments passed to Builder
    builder = Builder(soc, **builder_argdict(args))

    # Build the SoC is --build
    builder.build(run=args.build)

    # Load the bitstream if --load
    if args.load:
        prog = soc.platform.create_programmer()
        prog.load_bitstream(os.path.join(builder.gateware_dir, "impl", "pnr", "project.fs"))
```

LiteX – Build a SoC

- Builder has a set of arguments

```
def main():
    # Board specific arguments
    parser = argparse.ArgumentParser(description="LiteX SoC on Tang Nano")
    parser.add_argument("--build",       action="store_true", help="Build bitstream")
    parser.add_argument("--load",        action="store_true", help="Load bitstream")
    parser.add_argument("--flash",       action="store_true", help="Flash Bitstream")

    # Builder adds its own arguments
    builder_args(parser)

    # SoCCore adds its own arguments
    soc_core_args(parser)

    # args attributes are set with parsed arguments
    args = parser.parse_args()

    # soc_core_argdict filters and adapt arguments passed
    # to SoCCore
    soc = BaseSoC(
        sys_clk_freq      = 24e6,
        **soc_core_argdict(args)
    )

    # Filter and adapt arguments passed to Builder
    builder = Builder(soc, **builder_argdict(args))

    # Build the SoC is --build
    builder.build(run=args.build)

    # Load the bitstream if --load
    if args.load:
        prog = soc.platform.create_programmer()
        prog.load_bitstream(os.path.join(builder.gateware_dir, "impl", "pnr", "project.fs"))
```

LiteX – Builder arguments

```
--output-dir            -> Base Output directory (customizable with --{gateware,software,include,generated}-dir)
--gateware-dir          -> Output directory for Gateware files
--software-dir          -> Output directory for Software files
--include-dir           -> Output directory for Header files
--generated-dir         -> Output directory for Generated files
--no-compile-software   -> Disable Software compilation
--no-compile-gateware   -> Disable Gateware compilation
--csr-csv               -> Write SoC mapping to the specified CSV file
--csr-json              -> Write SoC mapping to the specified JSON file
--csr-svd               -> Write SoC mapping to the specified SVD file
--memory-x              -> Write SoC Memory Regions to the specified Memory-X file
--doc                   -> Generate SoC Documentation
```

- **--csr-csv** generates a file with CSR addresses and is used by all LiteX tools

LiteX – Build a SoC

- SoCCore has a set of arguments

```
def main():
    # Board specific arguments
    parser = argparse.ArgumentParser(description="LiteX SoC on Tang Nano")
    parser.add_argument("--build",       action="store_true", help="Build bitstream")
    parser.add_argument("--load",        action="store_true", help="Load bitstream")
    parser.add_argument("--flash",       action="store_true", help="Flash Bitstream")

    # Builder adds its own arguments
    builder_args(parser)

    # SoCCore adds its own arguments
    soc_core_args(parser)

    # args attributes are set with parsed arguments
    args = parser.parse_args()

    # soc_core_argdict filters and adapt arguments passed
    # to SoCCore
    soc = BaseSoC(
        sys_clk_freq      = 24e6,
        **soc_core_argdict(args)
    )

    # Filter and adapt arguments passed to Builder
    builder = Builder(soc, **builder_argdict(args))

    # Build the SoC is --build
    builder.build(run=args.build)

    # Load the bitstream if --load
    if args.load:
        prog = soc.platform.create_programmer()
        prog.load_bitstream(os.path.join(builder.gateware_dir, "impl", "pnr", "project.fs"))
```

LiteX – SoCCore arguments

```
--bus-standard              -> Select bus standard
--bus-data-width            -> Bus data-width (default=32)
--bus-address-width         -> Bus address-width (default=32)
--bus-timeout               -> Bus timeout in cycles (default=1e6)

--cpu-type                  -> Select CPU (default=vexriscv)
--cpu-variant               -> CPU variant (default=standard)
--cpu-reset-address         -> CPU reset address (default=None : Boot from Integrated ROM)
--cpu-cfu                   -> Optional CPU CFU file/instance to add to the CPU

--no-ctrl                   -> Disable Controller (default=False)

--integrated-rom-size       -> Size/Enable the integrated (BIOS) ROM (default=128KB, automatically resized to BIOS size when smaller)
--integrated-rom-init       -> Integrated ROM binary initialization file (override the BIOS when specified)

--integrated-sram-size      -> Size/Enable the integrated SRAM (default=8KB)

--integrated-main-ram-size  -> size/enable the integrated main RAM")

--csr-data-width            -> CSR bus data-width (8 or 32, default=32)
--csr-address-width         -> CSR bus address-width
--csr-paging                -> CSR bus paging
--csr-ordering              -> CSR registers ordering (default=big)

--ident                     -> SoC identifier (default=\"\")
--ident-version             -> Add date/time to SoC identifier (default=False)")

--no-uart                   -> Disable UART (default=False)
--uart-name                 -> UART type/name (default=serial)
--uart-baudrate             -> UART baudrate (default=115200)
--uart-fifo-depth           -> UART FIFO depth (default=16)

--no-timer                  -> Disable Timer (default=False)
--timer-uptime              -> Add an uptime capability to Timer (default=False)

--l2-size                   -> L2 cache size (default=8192)
```

Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
  - Presentation of SoCs and LiteX
  - Examples of peripherals integration
  - How to build a SoC
  - Software in LiteX
  - LiteX tools
  - Workshop
- LiteX: advanced topics

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 ==================--
Booting 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.

LiteX – Software, baremetal

- Build your own baremetal application using provided software libraries (spi, fatfs, sata, ethernet,...)
- BIOS can load the application
- Application can be loaded in ROM during build:

```
--integrated-rom-init=”myfile.bin”
```

LiteX – Software, generated files

```
.
└── generated
    ├── csr.h
    ├── git.h
    ├── mem.h
    ├── output_format.ld
    ├── regions.ld
    ├── soc.h
    └── variables.mak
```

- **csr.h** provides helper functions and definitions for all CSR peripherals
- **git.h** provides the git hash of the Litex version used to build the SoC
- **mem.h** definition of memory map as C defines
- **output_format.ld** and **regions.ld** are for the linker script
- **soc.h** provides the configuration of the SoC
- **variables.mak** are used by Makefiles

LiteX – Software, linker script

- Memory regions defined in generated/regions.ld

```
MEMORY {
        rom : ORIGIN = 0x00000000, LENGTH = 0x00020000
        sram : ORIGIN = 0x10000000, LENGTH = 0x00002000
        main_ram : ORIGIN = 0x40000000, LENGTH = 0x00001000
        csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```

- An example of linker script can be found in litex/soc/software/demo

- In general, your program will be placed in the main RAM before execution (by the BIOS)

- Program can also replace the BIOS in rom

- Need to adapt the linker script

Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
  - Presentation of SoCs and LiteX
  - Examples of peripherals integration
  - How to build a SoC
  - Software in LiteX
  - LiteX tools
  - Workshop
- LiteX: advanced topics

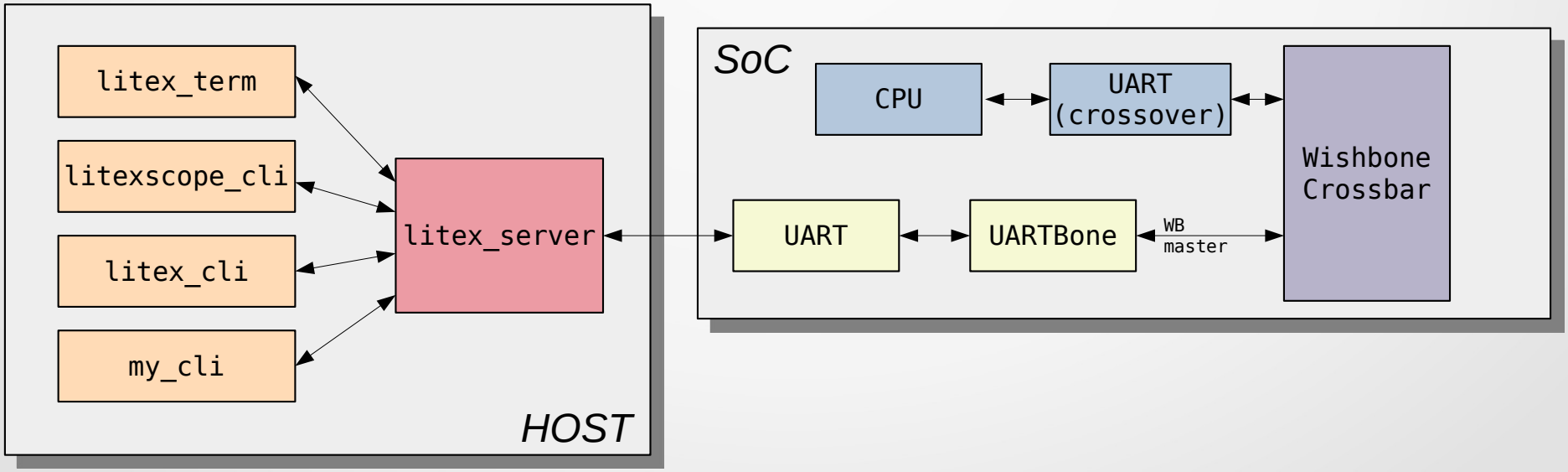
LiteX - Tools

- **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, litex_cli

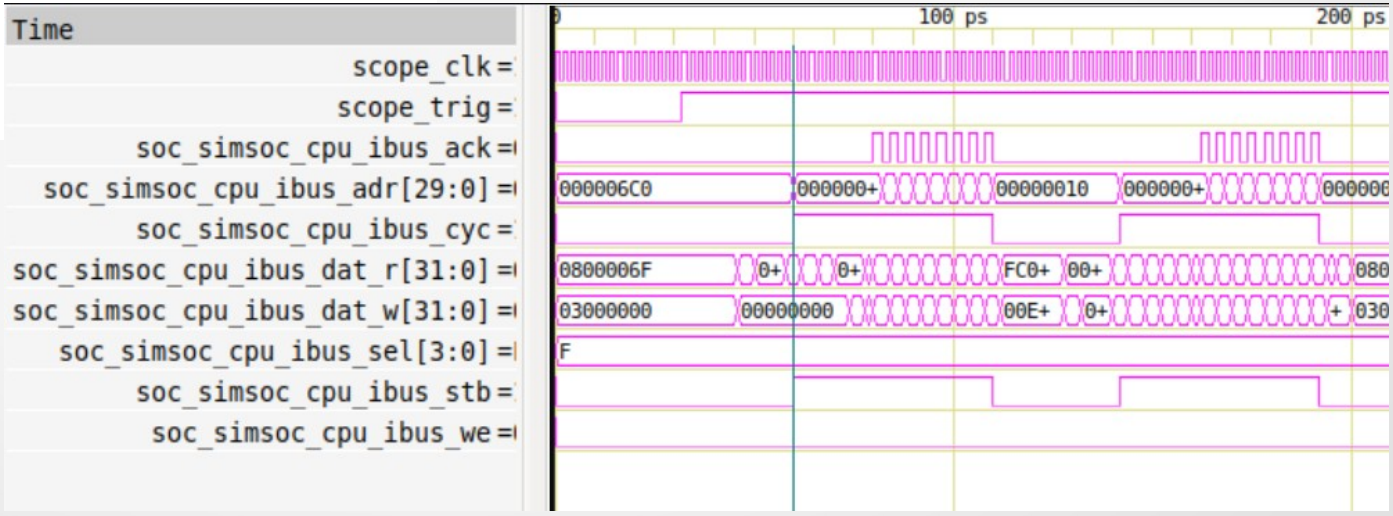
- Can read/write to arbitrary address
- Knows SoC registers (read from csr.csv file)
- Needs to connect to litex_server

```
$ litex_cli --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]...
[====================>]
[writing to dump.vcd]..
```



Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
  - Presentation of SoCs and LiteX
  - Examples of peripherals integration
  - How to build a SoC
  - Software in LiteX
  - LiteX tools
  - Workshop
- LiteX: advanced topics

LiteX – Workshop / Lessons

step7 – Build a simple SoC

step8 – Add CSR to RingControl and use litex_cli to control the leds

step9 – Add add use LiteScope and litescope_cli

step10 – Write a C program to control the leds and run it from the BIOS then run it from ROM

step11 – Add a PLL and clock the RingControl faster than the system

step12 – Add a wishbone interface to RingControl and use it

Now, let’s practice !

Step7 – Build a SoC

What you'll see:

- Derive and configure a SoC class
- Setup argument for local usage
- Use Build class
- Use of programmer

Step7 – Build a SoC

- Use --help to see all available arguments
- Try to build without a crg
- Load the bitstream and run **litex_server** and use **litex_cli** to reads the available registers and the SoC’s identifier

Step7 – Observations

- self.submodule.crg is mandatory
- Look at build logs

```
INFO:SoC:--------------------------------------------------------------------------------
INFO:SoC:Finalized SoC:
INFO:SoC:--------------------------------------------------------------------------------
INFO:SoC:32-bit wishbone Bus, 4.0GiB Address Space.
IO Regions: (1)
io0                 : Origin: 0x00000000, Size: 0x100000000, Mode: RW, Cached: False Linker: False
Bus Regions: (1)
csr                 : Origin: 0x00000000, Size: 0x00010000, Mode: RW, Cached: False Linker: False
Bus Masters: (1)
- uartbone
Bus Slaves: (1)
- csr
INFO:SoC:32-bit CSR Bus, 32-bit Aligned, 16.0KiB Address Space, 2048B Paging, big Ordering (Up to 32 Locations).
CSR Locations: (2)
- ctrl           : 0
- identifier_mem : 1
INFO:SoC:IRQ Handler (up to 32 Locations).
```

Step8 – Add CSR

What you'll learn:

- What CSR are
- Add and use CSR in a module
- Read/Write CSR from litex_cli

Step8 – What CSR are

- **C**ontrol and **S**tatus **R**egisters
- Registers placed on a simple bus accessible from Wishbone (bridged)
- Not aimed to do fast data transfers

Step8 – How to use CSR

- Inherit from **AutoCSR**

```
class MyModule(Module, AutoCSR):
    def __init__(self, param1, param2):
        self.version      = CSRConstant(0102)
        self.val_from_cpu = CSRStorage(24, reset=0x400000, description="This reg can be r/w by the CPU")
        self.val_to_cpu   = CSRStatus(2, description="This reg is written by the module, ro by CPU")
        self.val_w_fields = CSRStorage(description="CSR with fields",
                fields=[
                    CSRField("enable", size=1, description="Description field enable"),
                    CSRField("reset",  size=1, description="Description field reset", pulse=1)
                ])
        ...

        self.sync += [
            self.out.eq(self.val_from_cpu.storage),
            If(self.tx_ctl.fields.reset,
                ....
        ]
```

Step8 – How to use CSR

- Inherit from **AutoCSR**
- **CSRConstant** → Optimized away, values are set in generated software files

```
class MyModule(Module, AutoCSR):
    def __init__(self, param1, param2):
        self.version      = CSRConstant(0102)
        self.val_from_cpu = CSRStorage(24, reset=0x400000, description="This reg can be r/w by the CPU")
        self.val_to_cpu   = CSRStatus(2, description="This reg is written by the module, ro by CPU")
        self.val_w_fields = CSRStorage(description="CSR with fields",
                fields=[
                    CSRField("enable", size=1, description="Description field enable"),
                    CSRField("reset",  size=1, description="Description field reset", pulse=1)
                ])
        ...

        self.sync += [
            self.out.eq(self.val_from_cpu.storage),
            If(self.tx_ctl.fields.reset,
                ....
        ]
```

Step8 – How to use CSR

- Inherit from **AutoCSR**
- **CSRConstant** → Optimized away, values are set in generated software files
- **CSRStorage** → Register read/written by the CPU

```
class MyModule(Module, AutoCSR):
    def __init__(self, param1, param2):
        self.version       = CSRConstant(0102)
        self.val_from_cpu = CSRStorage(24, reset=0x400000, description="This reg can be r/w by the CPU")
        self.val_to_cpu    = CSRStatus(2, description="This reg is written by the module, ro by CPU")
        self.val_w_fields = CSRStorage(description="CSR with fields",
                fields=[
                    CSRField("enable", size=1, description="Description field enable"),
                    CSRField("reset",  size=1, description="Description field reset", pulse=1)
                ])
        ...

        self.sync += [
            self.out.eq(self.val_from_cpu.storage),
            If(self.tx_ctl.fields.reset,
                ....
        ]
```

Step8 – How to use CSR

- Inherit from **AutoCSR**
- **CSRConstant** → Optimized away, values are set in generated software files
- **CSRStorage** → Register read/written by the CPU
- **CSRStatus** → Register read only from the CPU

```
class MyModule(Module, AutoCSR):
    def __init__(self, param1, param2):
        self.version      = CSRConstant(0102)
        self.val_from_cpu = CSRStorage(24, reset=0x400000, description="This reg can be r/w by the CPU")
        self.val_to_cpu   = CSRStatus(2, description="This reg is written by the module, ro by CPU")
        self.val_w_fields = CSRStorage(description="CSR with fields",
                fields=[
                    CSRField("enable", size=1, description="Description field enable"),
                    CSRField("reset",  size=1, description="Description field reset", pulse=1)
                ])
        ...

        self.sync += [
            self.out.eq(self.val_from_cpu.storage),
            If(self.tx_ctl.fields.reset,
                ....
        ]
```

Step8 – How to use CSR

- **CSRStorage** and **CSRStatus** values must be accessed using their **storage** attribute

```
class MyModule(Module, AutoCSR):
    def __init__(self, param1, param2):
        self.version      = CSRConstant(0102)
        self.val_from_cpu = CSRStorage(24, reset=0x400000, description="This reg can be r/w by the CPU")
        self.val_to_cpu   = CSRStatus(2, description="This reg is written by the module, ro by CPU")
        self.val_w_fields = CSRStorage(description="CSR with fields",
                fields=[
                    CSRField("enable", size=1, description="Description field enable"),
                    CSRField("reset",  size=1, description="Description field reset", pulse=1)
                ])
        ...

        self.sync += [
            self.out.eq(self.val_from_cpu.storage),
            If(self.tx_ctl.fields.reset,
                ....
        ]
```

Step8 – How to use CSR

- **CSRStorage** and **CSRStatus** values must be accessed using their **storage** attribute
- **CSRField** are structured representation of a CSR
- **CSRField** is a Signal() and can be used directly

```
class MyModule(Module, AutoCSR):
    def __init__(self, param1, param2):
        self.version      = CSRConstant(0102)
        self.val_from_cpu = CSRStorage(24, reset=0x400000, description="This reg can be r/w by the CPU")
        self.val_to_cpu   = CSRStatus(2, description="This reg is written by the module, ro by CPU")
        self.val_w_fields = CSRStorage(description="CSR with fields",
                fields=[
                    CSRField("enable", size=1, description="Description field enable"),
                    CSRField("reset",  size=1, description="Description field reset", pulse=1)
                ])
        ...

        self.sync += [
            self.out.eq(self.val_from_cpu.storage),
            If(self.tx_ctl.fields.reset,
                ....
        ]
```

Step8 – How to use CSR

- CSR regions must be added to the SoC with **add_csr()**

```
class BaseSoC(SoCCore):
    def __init__(self, sys_clk_freq=int(100e6), mode=mode.DOUBLE, **kwargs):
        platform = Platform()

        SoCCore.__init__(self, platform, sys_clk_freq,
            ident          = "LiteX SoC on Arty A7-35",
            ident_version = True,
            **kwargs
        )

        self.submodules.crg = CRG(platform, sys_clk_freq)

        self.submodules.mymodule = MyModule(...)

        self.add_csr("mymodule")
```

Step8 – Documentation

- Documentation can be generated from CSR definition (- -doc)
- Fields can improve code readability and documentation
- You can add documentation for a module if you inherit from **AutoDoc**

```
class Timer(Module, AutoCSR, AutoDoc):
    with_uptime = False
    def __init__(self, width=32):
        self.intro = ModuleDoc("""Timer

    Provides a generic Timer core.

    The Timer is implemented as a countdown timer that can be used in various modes:
    ....
```

Example of a generated doc

https://github.com/enjoy-digital/litex/wiki/SoC-Documentation

Step8 – Let’s get to work

- Add a CSR to RingControl to control LED’s color
- Add RingControl to the SoC
- Use litex_cli to change the color of the LEDs

Bonus:

- Add a command line argument to control the mode at build time

Step8 – Observation

- submodules must be named to have CSR
- Default csr paging is 0x800 (2048 bytes), 32 bits, big endian and mapped at address 0xF0000000

```
...
Bus Regions: (3)
rom                 : Origin: 0x00000000, Size: 0x00020000, Mode: R, Cached: True Linker: False
sram                : Origin: 0x10000000, Size: 0x00002000, Mode: RW, Cached: True Linker: False
csr                 : Origin: 0xf0000000, Size: 0x00010000, Mode: RW, Cached: False Linker: False
...
```

```
INFO:SoC:--------------------------------------------------------------------------------
INFO:SoC:Finalized SoC:
INFO:SoC:--------------------------------------------------------------------------------
...
INFO:SoC:32-bit CSR Bus, 32-bit Aligned, 16.0KiB Address Space, 2048B Paging, ...
CSR Locations: (3)
- ledring        : 0
- ctrl           : 1
- identifier_mem : 2

$litex_cli --regs
0x00000000 : 0x00400000 ledring_color
0x00000800 : 0x00000000 ctrl_reset
0x00000804 : 0x12345678 ctrl_scratch
0x00000808 : 0x00000000 ctrl_bus_errors
```

Step9 – Add / configure / use Litescope

What you'll learn:

- Add Litescope to your design
- Use litescope_cli to configure trigger and dump waveforms

Step9 – Add / configure / use Litescope

- Needs a bridge to the SoC (uartbone, etherbone,…)
- Signals to be observed need to be listed in the source code (add accessible from the top level module)

```
analyzer_signals = [
    # IBus (could also just added as self.cpu.ibus)
    self.cpu.ibus.stb,
    self.cpu.ibus.cyc,
    self.cpu.ibus.adr,
    self.cpu.ibus.we,
    self.cpu.ibus.ack,
    self.cpu.ibus.sel,
    self.cpu.ibus.dat_w,
    self.cpu.ibus.dat_r,
]
```

```
from litescope import LiteScopeAnalyzer
self.submodules.analyzer = LiteScopeAnalyzer(
            analyzer_signals,
            depth        = 512,
            clock_domain ="sys",
            csr_csv      = "analyzer.csv"
)
```

Step9 – Add / configure / use Litescope

- Samples are stored in embedded block rams. Resources are limited !

```
from litescope import LiteScopeAnalyzer
self.submodules.analyzer = LiteScopeAnalyzer(
            analyzer_signals,
            depth        = 512,
            clock_domain ="sys",
            csr_csv      = "analyzer.csv"
)
self.add_csr("analyzer")
```

- ***depth*** configures how many samples are captured
- ***clock_domain*** tells which clock domain is used
- The current configuration is stored in ***analyzer.csv***

Step9 – Add / configure / use Litescope

- **litex_server** needs to be started
- **litescope_cli** is used to control the capture

```
$ litescope_cli --help
usage: litescope_cli [-h] [-r RISING_EDGE] [-f FALLING_EDGE]
                     [-v TRIGGER VALUE] [-l] [--csv CSV]
                     [--csr-csv CSR_CSV] [--group GROUP]
                     [--subsampling SUBSAMPLING] [--offset OFFSET]
                     [--length LENGTH] [--dump DUMP]

LiteScope Client utility

optional arguments:
  -h, --help            show this help message and exit
  -r RISING_EDGE, --rising-edge RISING_EDGE
                        Add rising edge trigger
  -f FALLING_EDGE, --falling-edge FALLING_EDGE
                        Add falling edge trigger
  -v TRIGGER VALUE, --value-trigger TRIGGER VALUE
                        Add conditional trigger with given value
  -l, --list            List signal choices
  --csv CSV             Analyzer CSV file
  --csr-csv CSR_CSV     SoC CSV file
  --group GROUP         Capture Group
  --subsampling SUBSAMPLING
                        Capture Subsampling
  --offset OFFSET       Capture Offset
  --length LENGTH       Capture Length
  --dump DUMP           Capture Filename
```

Step9 – Let’s get to work

- Add a Litescope instance
- Configure Litescope to visualize:
  - bit_count and trst_timer.wait in RingSerialCtrl,
  - Index in RingControl
- Triggers on trst_timer.wait rising edge
- Visualize the result

Step9 – Observation

- Signals that you want to watch must be part of the Module's interface

- Don't forget to add *self.add_csr("analyzer")*

- Several instances of LiteScopeAnalyzer can be used at the same time (e.g several clock domains)

Each litescope_cli needs to read the correct (- -csv) analyzer CSV file

https://github.com/enjoy-digital/litex/wiki/Use-LiteScope-To-Debug-A-SoC

Step10 – Write a baremetal software

What you'll learn:

- Create a baremetal software for your SoC
- Download and run your software using the litex_term
- Embedded your software in ROM

Step10 – Write a baremetal software

- Need to use SoCCore (was SoCMini until now)
- We need some RAM since the code will be upload from the host (in case we don’t replace the BIOS in ROM)

```
--integrated-main-ram-size 0x1000

Bus Regions: (4)
rom                 : Origin: 0x00000000, Size: 0x00020000, Mode: R, Cached: True Linker: False
sram                : Origin: 0x10000000, Size: 0x00002000, Mode: RW, Cached: True Linker: False
main_ram            : Origin: 0x40000000, Size: 0x00001000, Mode: RW, Cached: True Linker: False
csr                 : Origin: 0xf0000000, Size: 0x00010000, Mode: RW, Cached: False Linker: False
```

- Write a makefile that uses the generated variables from the SoC definition
- Provide a linker script

Step10 – Workshop

- Build the SoC with some integrated main ram
- Complete the provided main.c to control the color of the LEDs
- Load and run the program using litex_term
- Build the program to target the ROM
- Initialize the ROM with your program and load the bitstream

Step10 – Workshop

- Booting from ROM require a change in the Linker script

```
--- linker-rom.ld   2022-01-05 15:08:42.751995784 +0100
+++ linker.ld   2022-01-05 14:23:18.786054588 +0100
@@ -27,7 +27,7 @@

         *(.text .stub .text.* .gnu.linkonce.t.*)
         _etext = .;
-    } > rom
+    } > main_ram
```

Step11 – PLL and ClockDomains

What you'll learn:

- What is a ClockDomain
- Use a PLL
- Use ClockDomainsRenamer

Migen – Attributes of Modules: clock_domains

- ***clock_domains*** → clock domains used by this module
- Clock domains object contains:
  - a the name for the clock domain
  - a clock signal
  - an optional reset signal
- Default clock domain is **sys** (implicit)
- A module can have more than one clock domain

```
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
```

Migen – Attributes of Modules: clock_domains

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

        ###

        # Add a clock domain to M1
        self.clock_domains.cd_pix = ClockDomain()
        # Connect pic_clk to cd_pix clock signal
        self.comb += self.cd_pix.clk.eq(self.pix_clk)
        # Add a synchronous assignment in cd_pix clock domain
        self.sync.pix += [
            self.a.eq(self.b),
        ]

        # Add a synchronous assignment in cd_sys clock domain (implicit)
        self.sync += [
            self.leds.eq(self.btn & self.write),
        ]
```

Create a new clock domain and assign a clock signal

Migen – Attributes of Modules: clock_domains

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

        ###

        # Add a clock domain to M1
        self.clock_domains.cd_pix = ClockDomain()
        # Connect pic_clk to cd_pix clock signal
        self.comb += self.cd_pix.clk.eq(self.pix_clk)
        # Add a synchronous assignment in cd_pix clock domain
        self.sync.pix += [
            self.a.eq(self.b),
        ]

        # Add a synchronous assignment in cd_sys clock domain (implicit)
        self.sync += [
            self.leds.eq(self.btn & self.write),
        ]
```

This assignment takes place in the “pix” clock domain

Migen – Attributes of Modules: clock_domains

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

        ###

        # Add a clock domain to M1
        self.clock_domains.cd_pix = ClockDomain()
        # Connect pic_clk to cd_pix clock signal
        self.comb += self.cd_pix.clk.eq(self.pix_clk)
        # Add a synchronous assignment in cd_pix clock domain
        self.sync.pix += [
            self.a.eq(self.b),
        ]

        # Add a synchronous assignment in cd_sys clock domain (implicit)
        self.sync += [
            self.leds.eq(self.btn & self.write),
        ]
```

Assignment to ***cd_sys*** is implicit

Migen – Attributes of Modules: clock_domains

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

        ###

        # Add a clock domain to M1
        self.clock_domains.cd_pix = ClockDomain()
        # Connect pic_clk to cd_pix clock signal
        self.comb += self.cd_pix.clk.eq(self.pix_clk)
        # Add a synchronous assignment in cd_pix clock domai
        self.sync.pix += [
            self.a.eq(self.b),
        ]

        # Add a synchronous assignment in cd_sys clock domai
        self.sync += [
            self.leds.eq(self.btn & self.write),
        ]
```

```
module top(
        output reg leds,
        input btn,
        input write,
        output reg a,
        input b,
        input pix_clk,
        input sys_clk,
        input sys_rst
);

wire pix_clk_1;
reg pix_rst = 1'd0;

assign pix_clk_1 = pix_clk;

always @(posedge pix_clk_1) begin
        a <= b;
        if (pix_rst) begin
                a <= 1'd0;
        end
end

always @(posedge sys_clk) begin
        leds <= (btn & write);
        if (sys_rst) begin
                leds <= 1'd0;
        end
end

endmodule
```

Step11 – ClockDomainsRenamer

- Change the clock domain of a module
- Used while adding a submodule
- Can change several clock domains at the same time

```
self.submodules += ClockDomainsRenamer("new")(MyModule())

read_fifo = ClockDomainsRenamer({"write": "usb", "read": "sys"})(read_fifo)
```

Step11 – ClockDomainsRenamer - Example

```
class FreqMeter(Module, AutoCSR):
    def __init__(self, period, width=6, clk=None):
        self.clk   = Signal() if clk is None else clk
        self.value = CSRStatus(32)

        # # #

        self.clock_domains.cd_fmeter = ClockDomain(reset_less=True)
        self.comb += self.cd_fmeter.clk.eq(self.clk)

        # Period generation
        period_done    = Signal()
        period_counter = Signal(32)
        self.comb += period_done.eq(period_counter == period)
        self.sync += period_counter.eq(period_counter + 1)
        self.sync += If(period_done, period_counter.eq(0))

        # Frequency measurement
        event_counter = ClockDomainsRenamer("fmeter")(GrayCounter(width))
        gray_decoder  = GrayDecoder(width)
        sampler       = _Sampler(width)
        self.submodules += event_counter, gray_decoder, sampler
```

Step11 – ClockDomainsRenamer

```
class Descrambler(Module, AutoCSR):
    def __init__(self, clock_domain):

        self.testmode = CSRStorage()
        ...
        _testmode    = Signal()

        self.specials += MultiReg(self.testmode.storage, _testmode, clock_domain)
        ...
        sync = getattr(self.sync, clock_domain)
        sync += [
            self.source.type.eq(self.sink.type),
            ...
```

Usage:

```
self.submodules.descrambler = Descrambler("gtp0_rx")
```

And:

```
sync = getattr(self.sync, clock_domain)
```

Is equivalent to:

```
self.sync.gtp0_rx += [
                ...
```

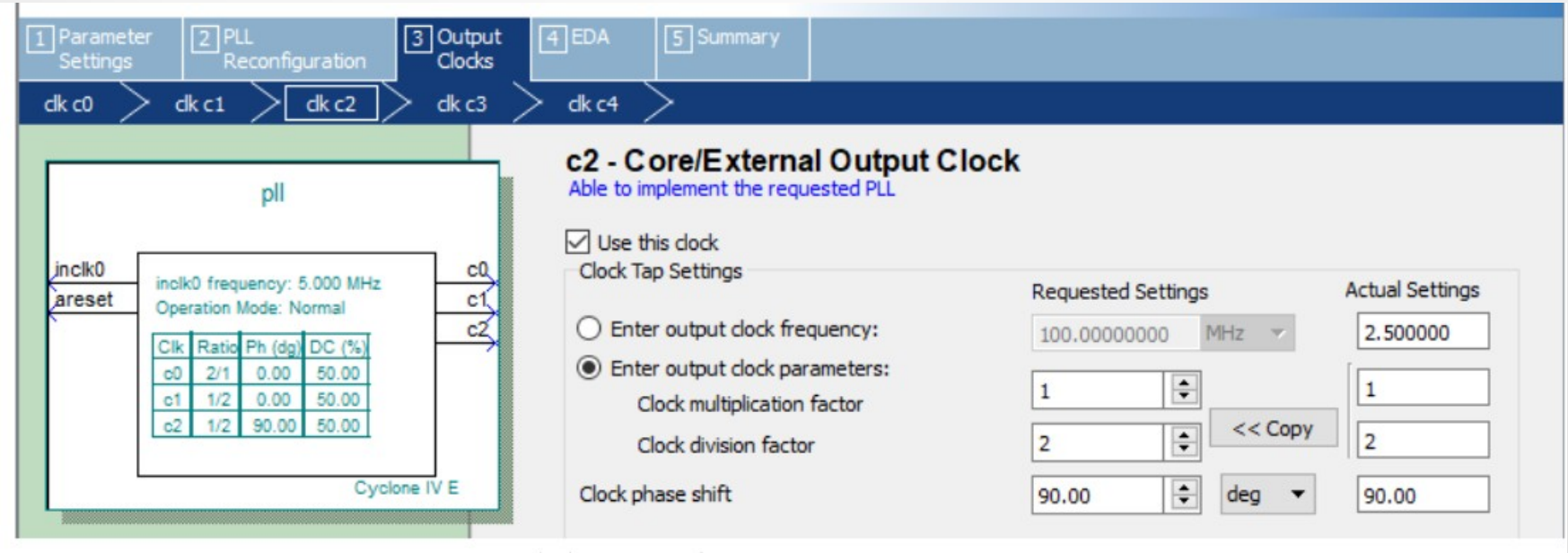
Step11 – ClockDomainsRenamer

```
class Descrambler(Module, AutoCSR):
    def __init__(self, clock_domain):

        self.testmode = CSRStorage()
        ...
        _testmode    = Signal()

        self.specials += MultiReg(self.testmode.storage, _testmode, clock_domain)
        ...
        sync = getattr(self.sync, clock_domain)
        sync += [
            self.source.type.eq(self.sink.type),
            ...
```

- CSR are always in cd_sys

Step11 – PLL

- Phase Locked Loop
- One clock input, several clock output
- Clock multiplication, phase shift



Step11 – PLL

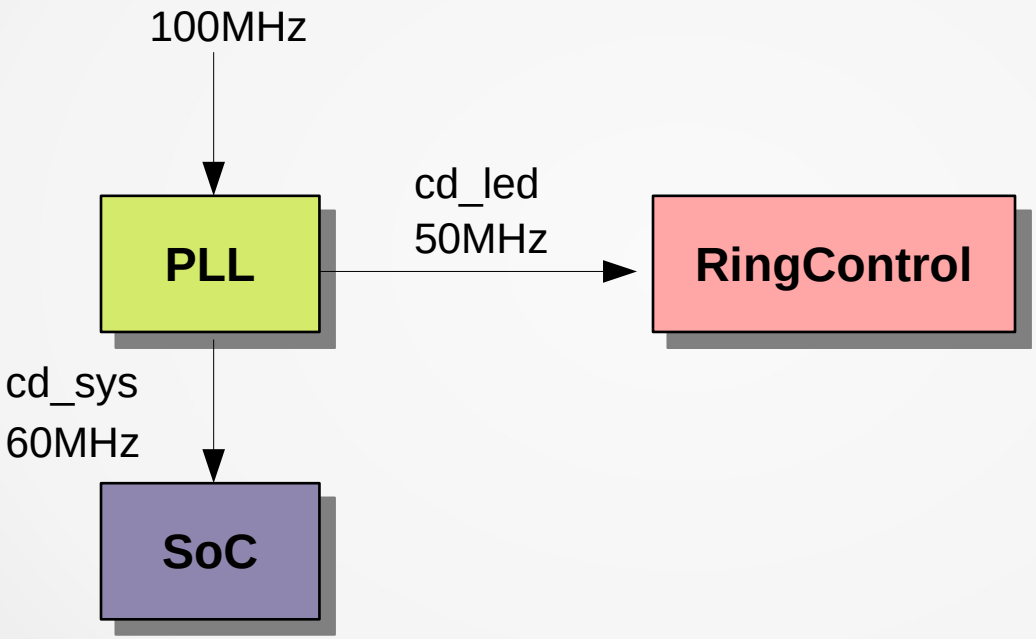
- PLL code is in litex → soc → cores → clock
- Constructor can be slightly different between platforms

```
self.submodules.pll = pll = ECP5PLL()
self.comb += pll.reset.eq(~rst_n | self.rst)
pll.register_clkin(clk, 10e6)
pll.create_clkout(self.cd_sys, 120e6)
```

- You still need to get an idea what your PLL is capable of

Step11 – Workshop

- Add a PLL and clock the design as shown here after



- Change the color of the LEDs using the BIOS (there is no uart_bone anymore)

Step11 – Observations

- Reset signal of clock domains is automatically handled
- CSR are in sys clock domain

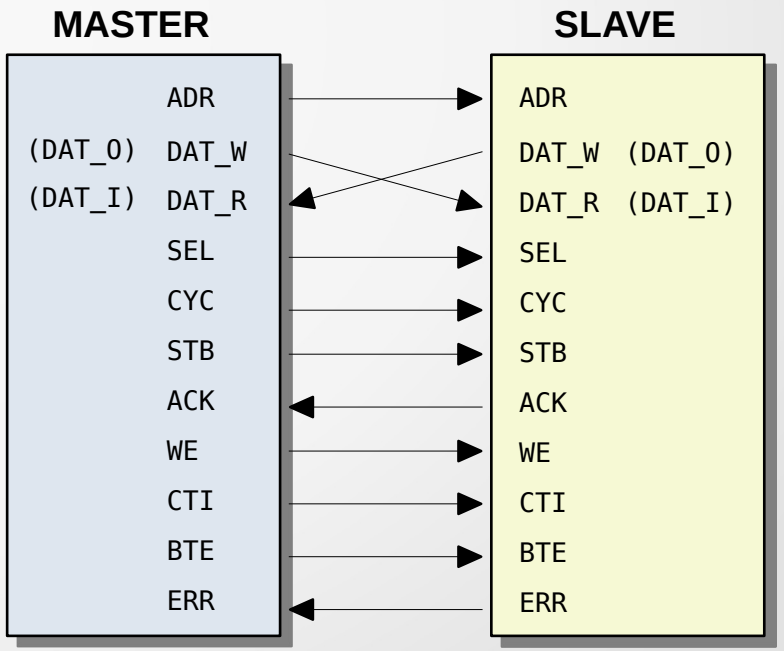
Step12 – Use the wishbone bus

What you'll learn:

- How Wishbone works
- Add and use a wishbone slave
- Add and use a wishbone master

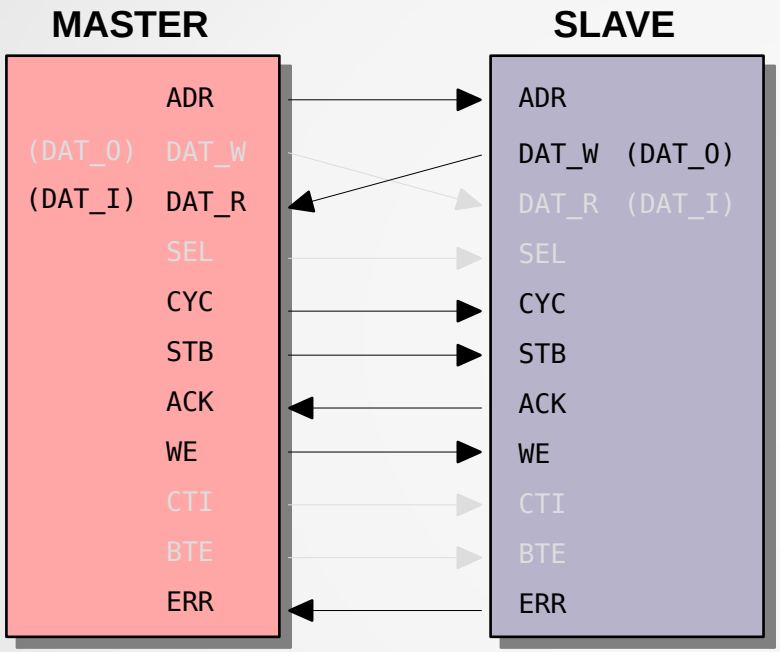
Step12 – Wishbone description

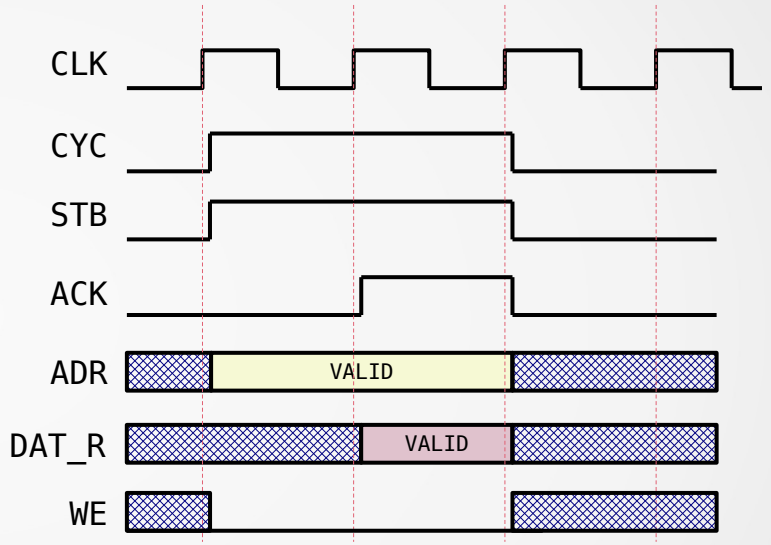
- Open source hardware bus definition
- 8 – 64 bits data bus
- Supports single transfers and bursts
- Two version are used: **B3** and **B4**
- B4 introduces pipelined transfers
- **LiteX uses the Wishbone B3**



https://cdn.opencores.org/downloads/wbspec_b3.pdf

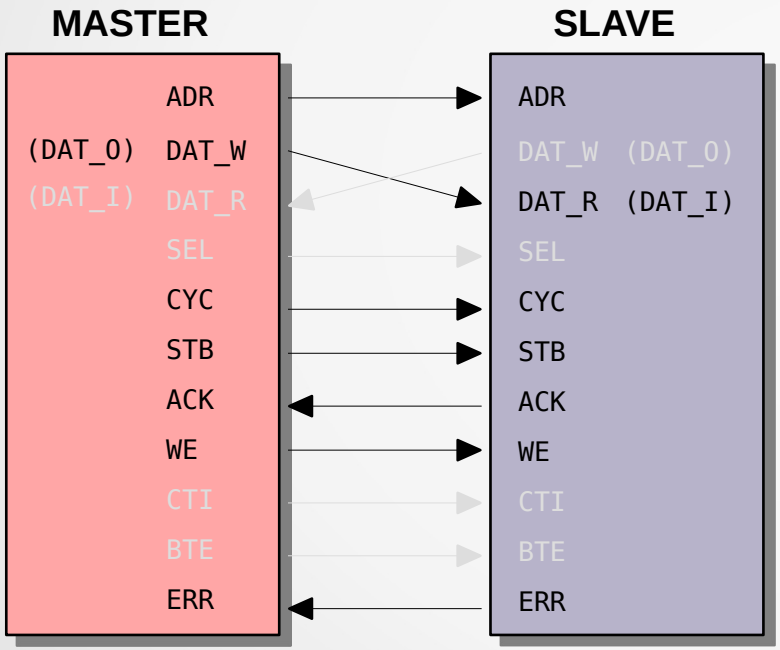
Step12 – Wishbone simple read

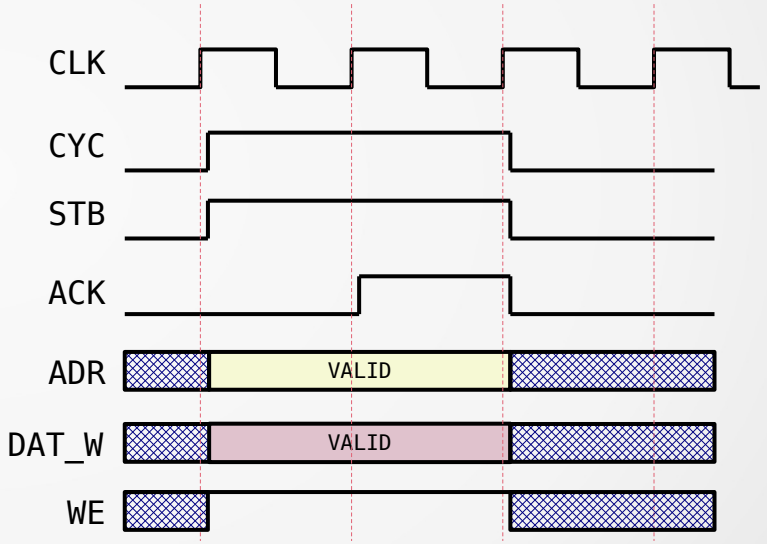




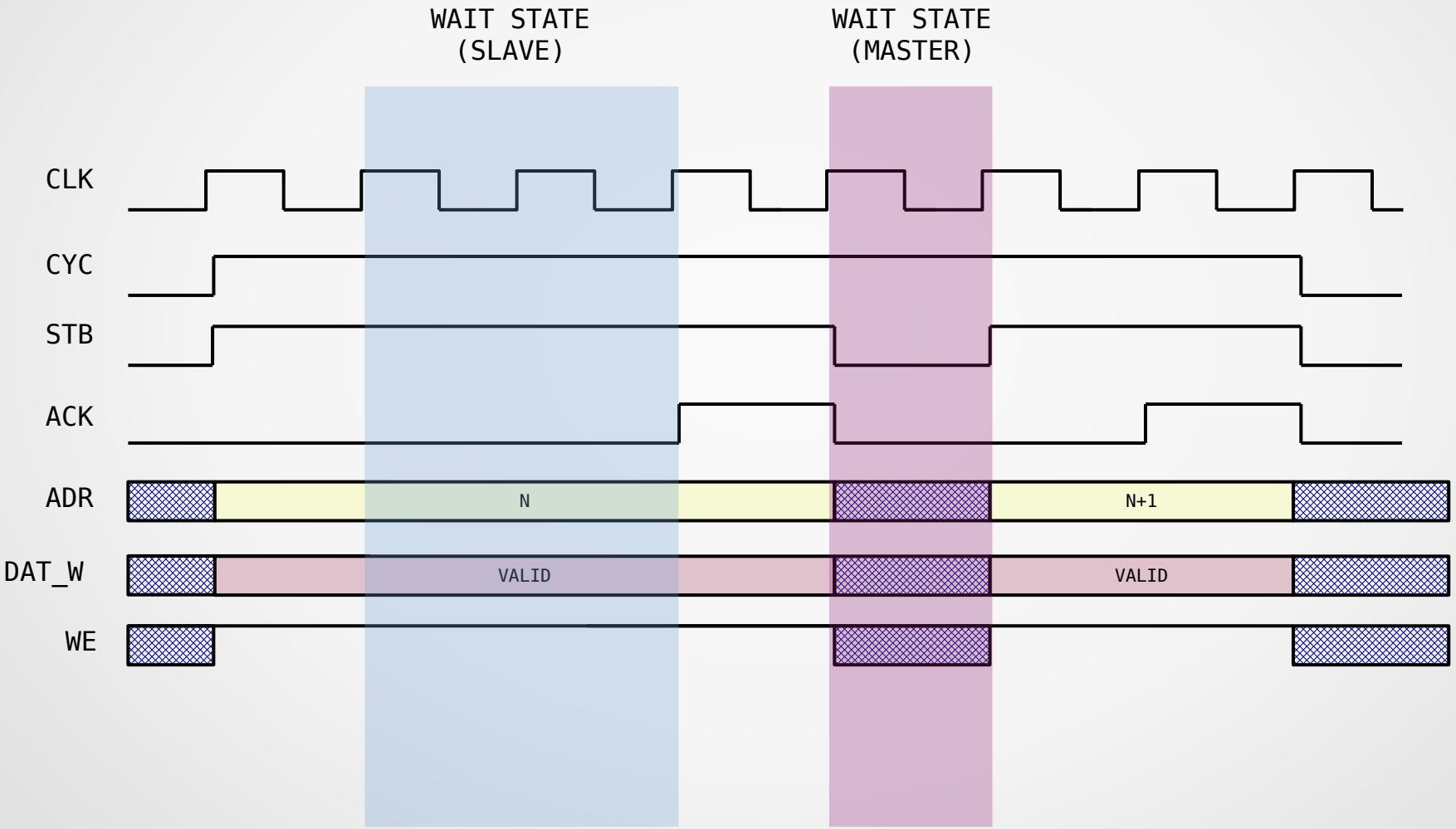
- ERR can finish a cycle (like ACK)

Step12 – Wishbone simple write



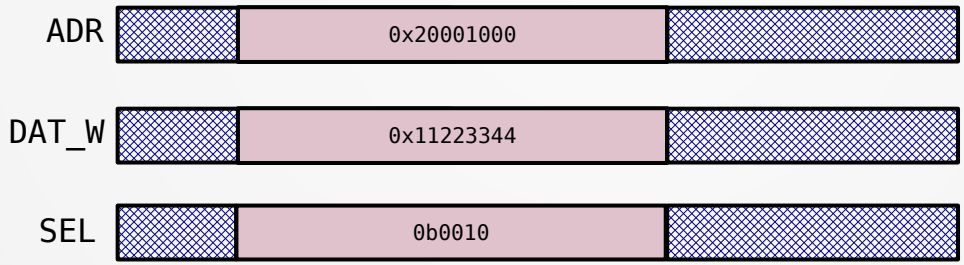


Step12 – Wishbone wait states



Step12 – Wishbone SEL

- Indicates where valid data is on the bus
- Used when a granularity smaller than the bus width is needed (write a 8-bit value on a 32-bit bus)



- In this example, 0x33 is written at address 0x20001001
- Depends on ENDIANNESS

Step12 – Wishbone burst cycles

- Increase bandwidth (1 transfer per cycle)

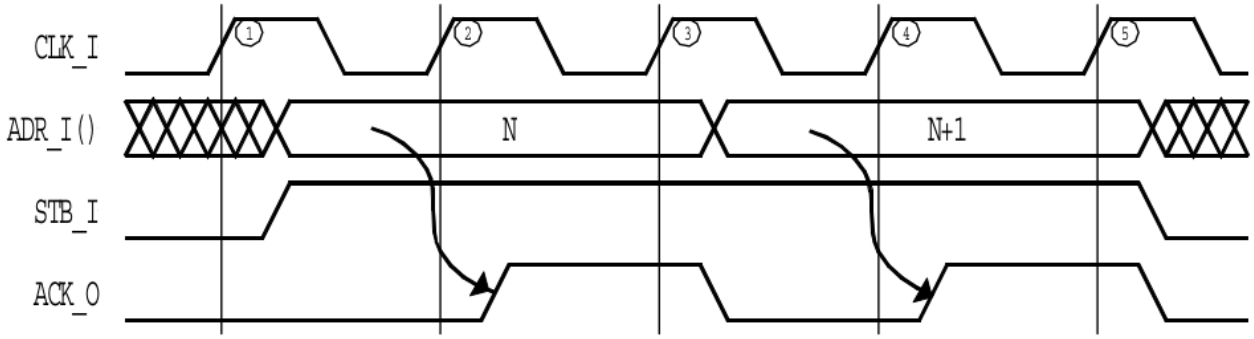


Figure 4-3 WISHBONE Classic synchronous cycle terminated burst

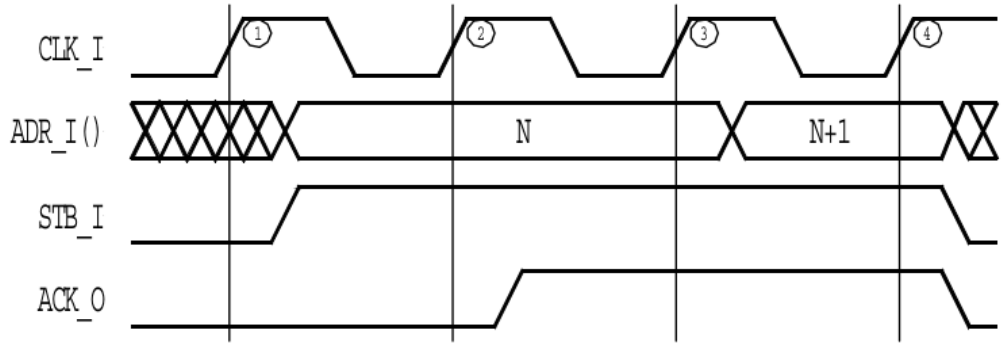


Figure 4-4 Advanced synchronous terminated burst

Step12 – Wishbone burst cycles

- Use CTI (Cycle Type Idenfier)
- Use BTE (Burst Type Extension)

Table 4-2 Cycle Type Identifiers	
CTI_O(2:0)	Description
'000'	Classic cycle.
'001'	Constant address burst cycle
'010'	Incrementing burst cycle
'011'	*Reserved*
'100'	*Reserved*
'101	*Reserved*
'110'	*Reserved*
'111'	End-of-Burst

Table 4-2 Burst Type Extension for Incrementing and Decrementing bursts	
BTE_IO(1:0)	Description
'00'	Linear burst
'01'	4-beat wrap burst
'10'	8-beat wrap burst
'11'	16-beat wrap burst

Table 4-3 Wrap Size address increments			
Starting address' LSBs	Linear	Wrap-4	Wrap-8
000	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
001	1-2-3-4-5-6-7-8	1-2-3-0-5-6-7-4	1-2-3-4-5-6-7-0
010	2-3-4-5-6-7-8-9	2-3-0-1-6-7-4-5	2-3-4-5-6-7-0-1
011	3-4-5-6-7-8-9-A	3-0-1-2-7-4-5-6	3-4-5-6-7-0-1-2
100	4-5-6-7-8-9-A-B	4-5-6-7-8-9-A-B	4-5-6-7-0-1-2-3
101	5-6-7-8-9-A-B-C	5-6-7-4-9-A-B-8	5-6-7-0-1-2-3-4
110	6-7-8-9-A-B-C-D	6-7-4-5-A-B-8-9	6-7-0-1-2-3-4-5
111	7-8-9-A-B-C-D-E	7-4-5-6-B-8-9-A	7-0-1-2-3-4-5-6

Step12 – Wishbone burst cycles

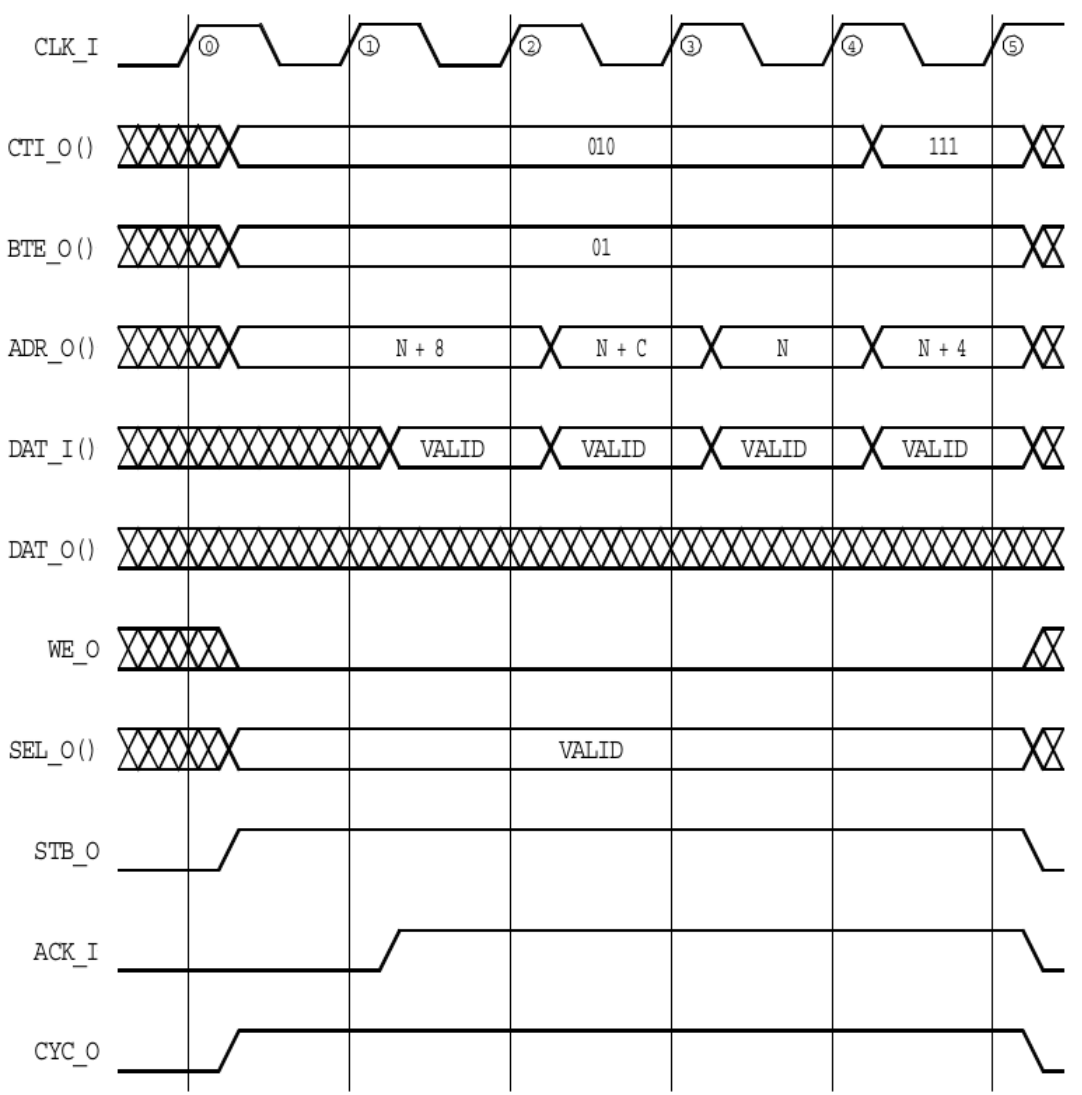


Figure 4-8 4-beat wrapped incrementing burst for a 32bit data array

Step12 – Use Wishbone slave with LiteX

```
# Add mymodule as a Wisbone slave in a non-cacheable region
self.bus.add_slave(name="mymodule", slave=self.mymodule.bus, region=SoCRegion(
     size   = 4,
     cached = False
 ))
```

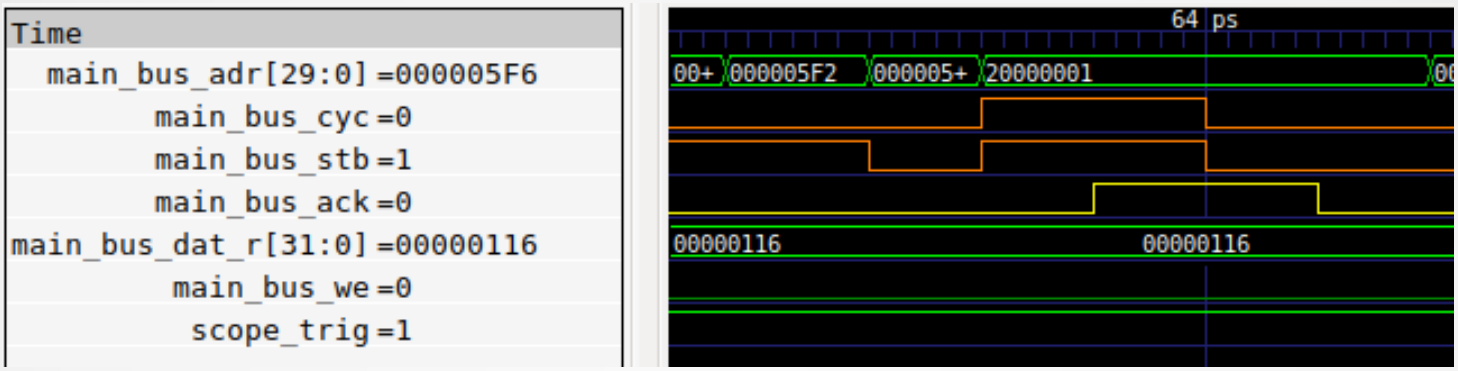
```
INFO:SoC:32-bit wishbone Bus, 4.0GiB Address Space.
IO Regions: (1)
io0                 : Origin: 0x80000000, Size: 0x80000000, Mode: RW, Cached: False Linker: False
Bus Regions: (4)
rom                 : Origin: 0x00000000, Size: 0x00020000, Mode: R, Cached: True Linker: False
sram                : Origin: 0x10000000, Size: 0x00002000, Mode: RW, Cached: True Linker: False
ledring             : Origin: 0x80000000, Size: 0x00000004, Mode: RW, Cached: False Linker: False
csr                 : Origin: 0xf0000000, Size: 0x00010000, Mode: RW, Cached: False Linker: False
```

- **add_slave** method from SoCCore
- IO Regions are non-cacheable
- Origin can be specified
- Address to the module is adr[2:32] and is not relative to the base address

Step12 – Exercise

- Add a wishbone interface slave to RingControl
- Use this bus to control the ring’s color
- Use this bus to read a version number
- Read and write values from the BIOS

Step12 – Observation



- The address is expressed in 4 bytes words
- The address is not relative to the base address of the module

Step12bis – Use Wishbone master with LiteX

```
self.bus.add_master(name="mymodule", master=self.mymodule.bus)
```

```
INFO:SoC:32-bit wishbone Bus, 4.0GiB Address Space.
IO Regions: (1)
io0                 : Origin: 0x80000000, Size: 0x80000000, Mode: RW, Cached: False Linker: False
Bus Regions: (4)
rom                 : Origin: 0x00000000, Size: 0x00020000, Mode: R, Cached: True Linker: False
sram                : Origin: 0x10000000, Size: 0x00002000, Mode: RW, Cached: True Linker: False
main_ram            : Origin: 0x40000000, Size: 0x10000000, Mode: RW, Cached: True Linker: False
csr                 : Origin: 0xf0000000, Size: 0x00010000, Mode: RW, Cached: False Linker: False
Bus Masters: (3)
- cpu_bus0
- cpu_bus1
- ledring
```

- **add_master** method from SoCCore
- Address from the module is adr[2:32]

Step12bis – Exercise

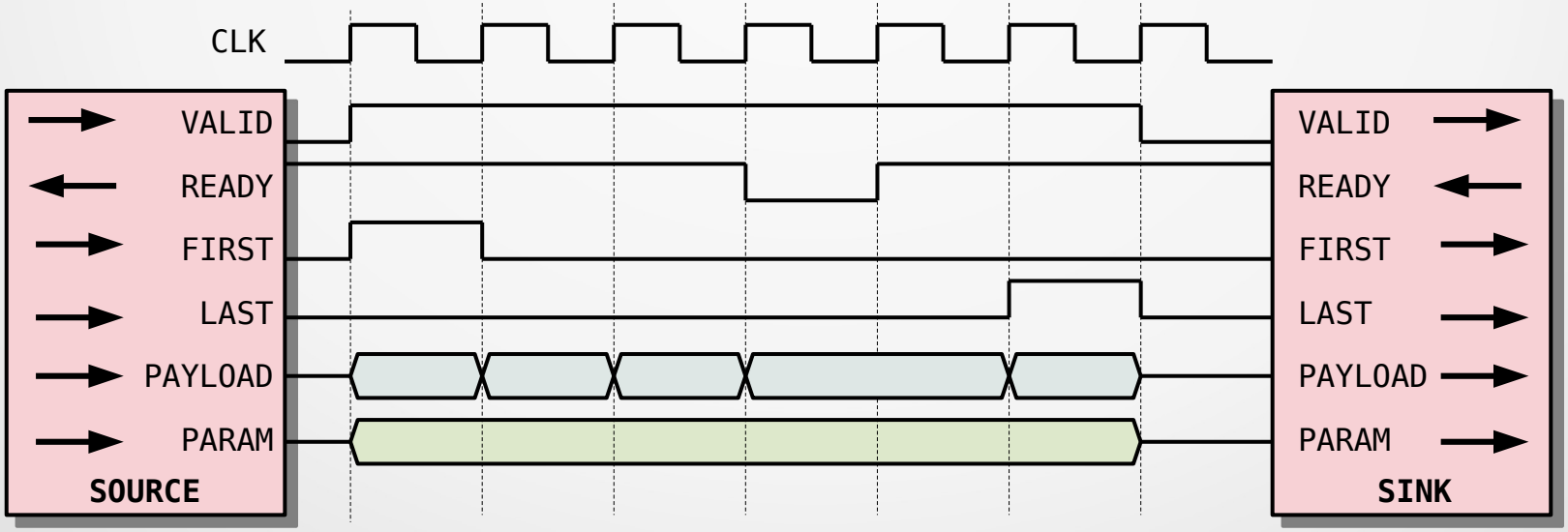
- A DDR3 controller has been added as main_ram mapped at address 0x40000000
- Add a wishbone master to RingControl
- Read LEDs color from the DRAM using the wishbone master interface
- Color will be written to memory from the BIOS

Agenda

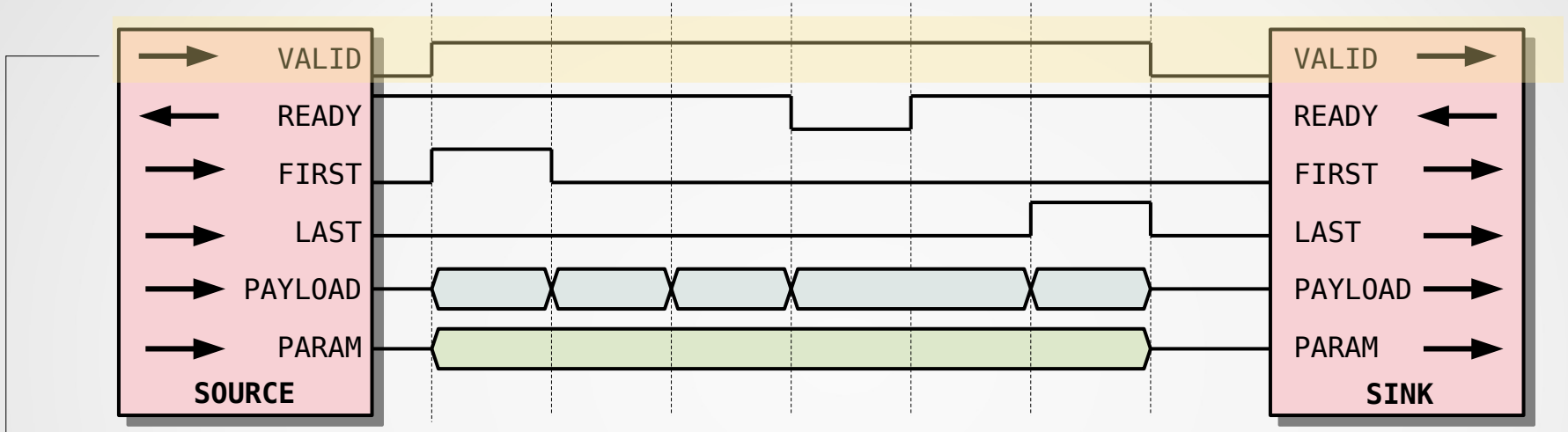
- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
- LiteX: advanced topics
  - Streams / workshop
  - Usage of Verilog/VHDL modules in LiteX / workshop
  - Verilator / workshop

LiteX – Streams

- Streams are groups of signals (Migen's record) used to exchange data between Modules
- There is no "addresses" on this "bus"
- Transfers are from the **Source** to the **Sink**
- Stream nodes are called **Endpoints**

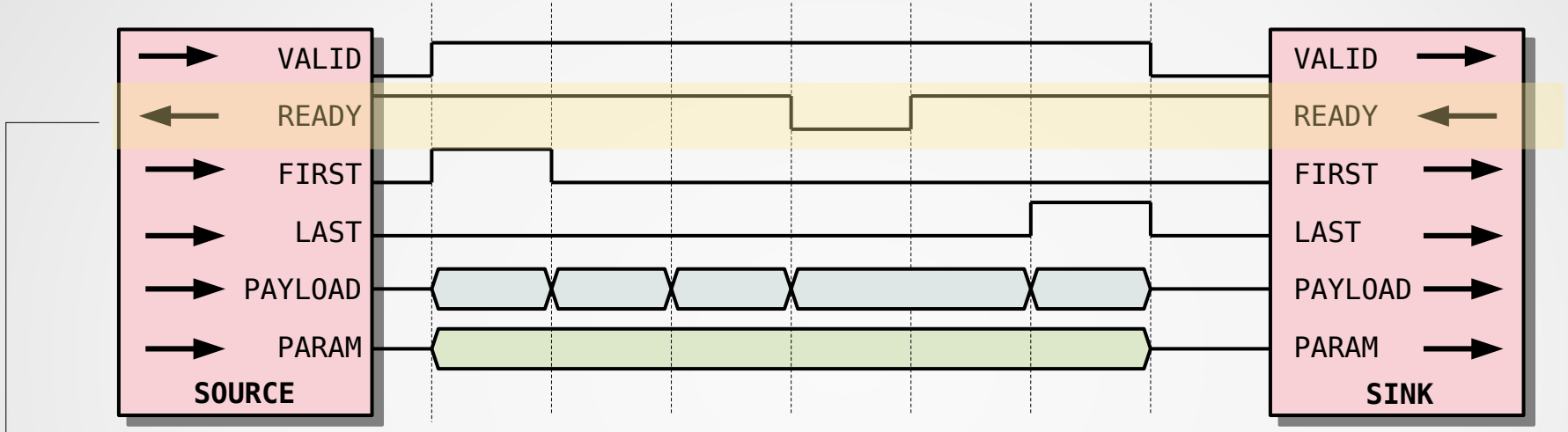


LiteX – Streams



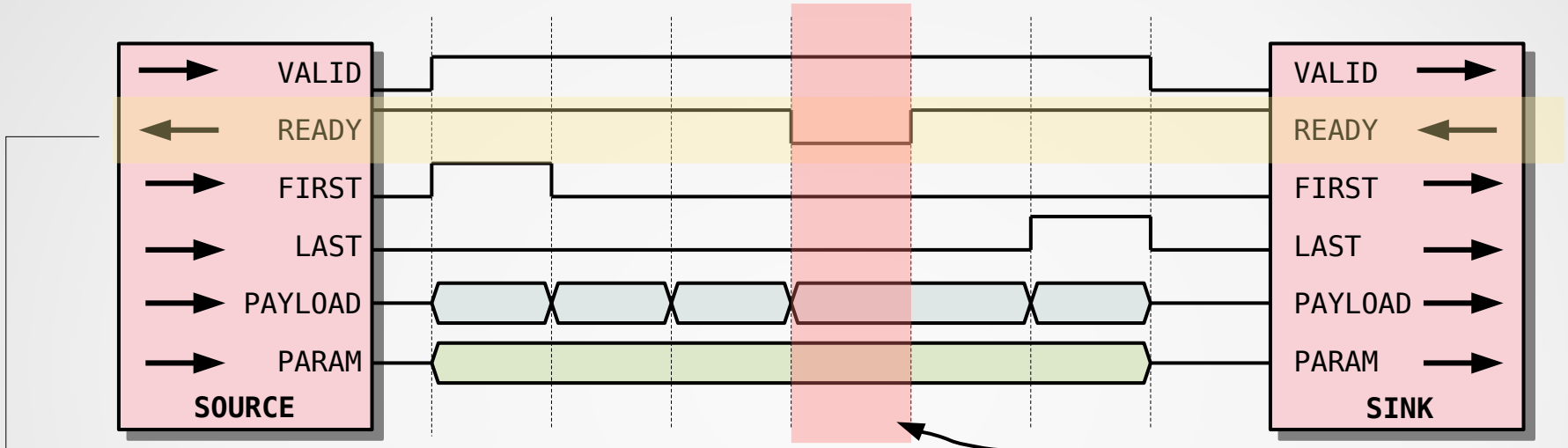
- ***valid*** indicates data from source are valid

LiteX – Streams



- ***valid*** indicates data from source are valid
- ***ready*** is high when sink is ready to receive

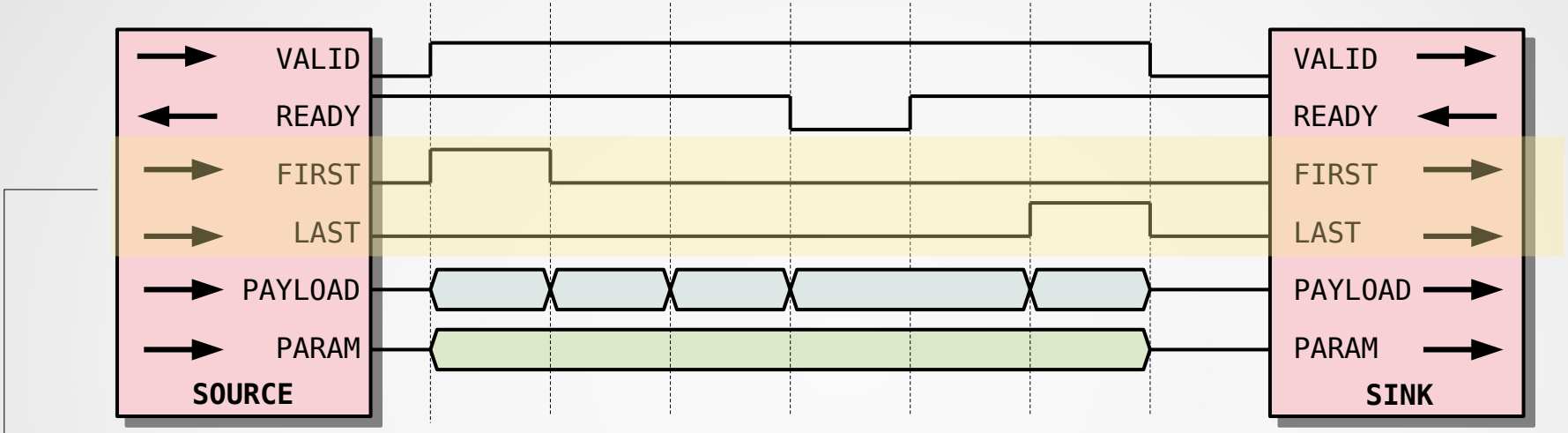
LiteX – Streams



- ***valid*** indicates data from source are valid
- ***ready*** is high when sink is ready to receive

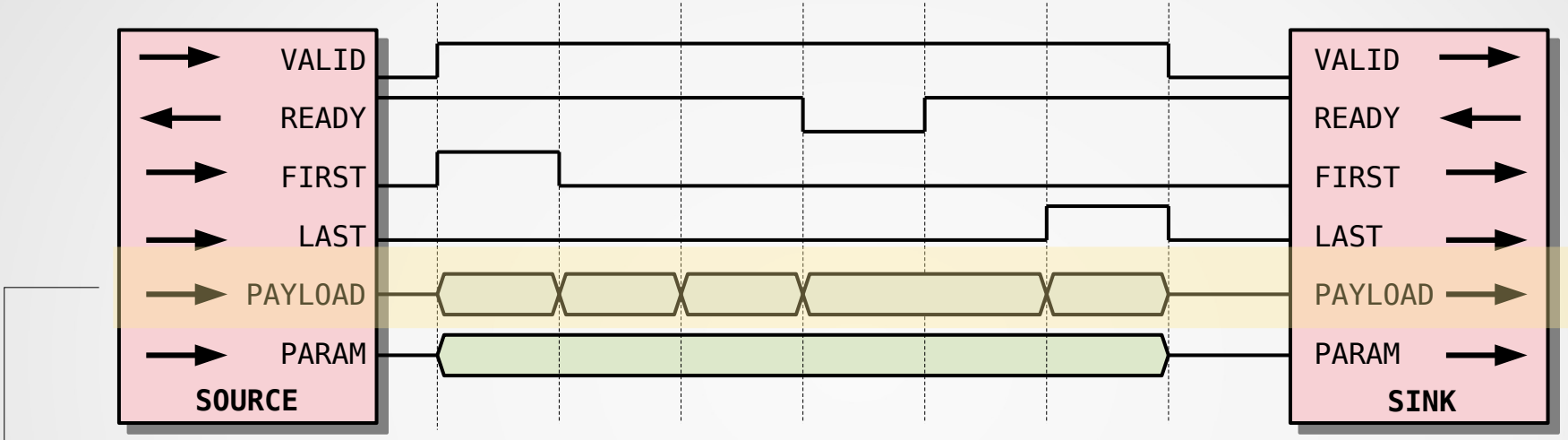
Sink is not ready.
The Source keeps the current Payload
until Sink is ready again

LiteX – Streams



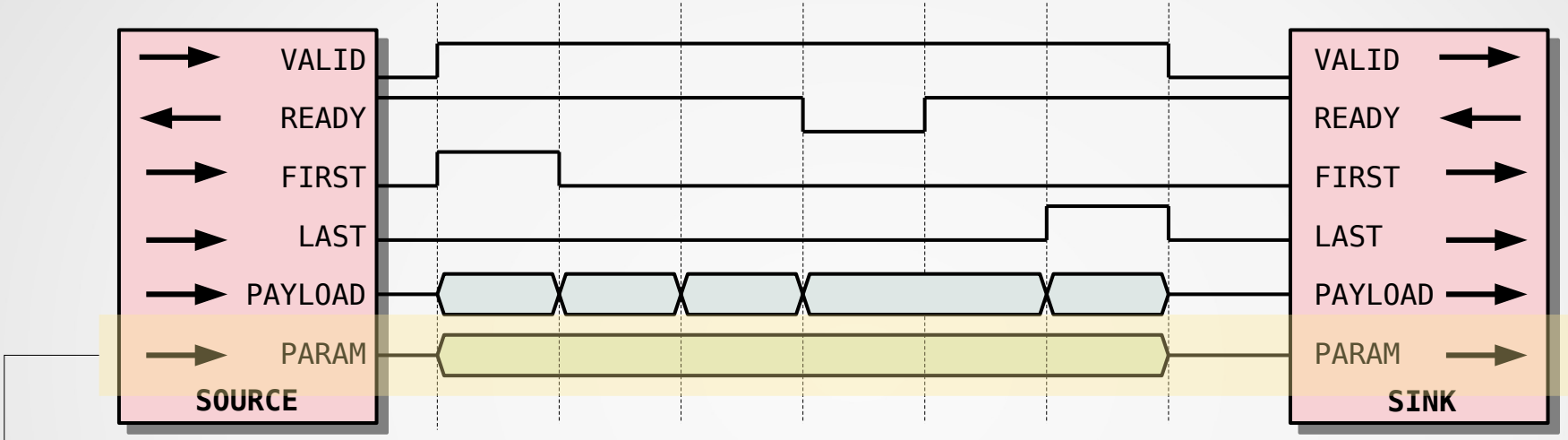
- ***valid*** indicates data from source are valid
- ***ready*** is high when sink is ready to receive
- ***first*** and ***last*** mark ***packets*** boundaries

LiteX – Streams



- ***valid*** indicates data from source are valid
- ***ready*** is high when sink is ready to receive
- ***first*** and ***last*** mark ***packets*** boundaries
- ***payload*** is a Record with its own layout, it can change on every valid/ready transaction

LiteX – Streams



- ***valid*** indicates data from source are valid
- ***ready*** is high when sink is ready to receive
- ***first*** and ***last*** mark ***packets*** boundaries
- ***payload*** is a Record with its own layout, it can change on every valid/ready transaction
- ***param*** is a Record with its own layout, it can evolve at each start of packet

LiteX – Streams usage

- Streams are ***Endpoint()*** classes
- Defined from a ***layout***
- ***param_layout*** is optional
- valid, ready, first, last are added automatically

```
descrambler_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4)
]

param_layout = [
    ("config", 4),
    ("version", 2)
]

class DetectOrderedSets(Module):
    def __init__(self):
        self.source = source = stream.Endpoint(descrambler_layout, param_layout)
        self.sink   = sink   = stream.Endpoint([("data", 16), ("ctrl", 2)])
```

LiteX – Streams example

```
filter_fifo_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4),
    ("ts"   , 32),
    ("error", 1),
]
```

```
descrambler_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4)
]
```

Input and output Streams are added to the Module

```
class Filter(Module, AutoCSR):
    def __init__(self):
        ...
        self.source       = source = stream.Endpoint(descrambler_layout)
        self.sink         = sink   = stream.Endpoint(descrambler_layout)
        ...
        fifo = ResetInserter()(stream.SyncFIFO(filter_fifo_layout, fifo_size))
        self.submodules.fifo   = ClockDomainsRenamer(clock_domain)(fifo)
        ...
        self.source.connect(fifo.sink, omit={"valid", "ts", "error"}),
        ...

        fsmWriter.act("NO_FILTER",
            NextValue(fifo.sink.valid, 0),
            ...
```

LiteX – Streams example

```
filter_fifo_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4),
    ("ts"   , 32),
    ("error", 1),
]

descrambler_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4)
]

class Filter(Module, AutoCSR):
    def __init__(self):
        ...
        self.source       = source = stream.Endpoint(descrambler_layout)
        self.sink         = sink   = stream.Endpoint(descrambler_layout)
        ...
        fifo = ResetInserter()(stream.SyncFIFO(filter_fifo_layout, fifo_size))
        self.submodules.fifo   = ClockDomainsRenamer(clock_domain)(fifo)
        ...
        self.source.connect(fifo.sink, omit={"valid", "ts", "error"}),
        ...

        fsmWriter.act("NO_FILTER",
            NextValue(fifo.sink.valid, 0),
            ...
```

Add a stream FIFO with its own layout

LiteX – Streams example

```
filter_fifo_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4),
    ("ts"   , 32),
    ("error", 1),
]

descrambler_layout = [
    ("data" , 16),
    ("ctrl" , 2),
    ("osets", 2),
    ("type" , 4)
]

class Filter(Module, AutoCSR):
    def __init__(self):
        ...
        self.source        = source = stream.Endpoint(descrambler_layout)
        self.sink          = sink   = stream.Endpoint(descrambler_layout)
        ...
        fifo = ResetInserter()(stream.SyncFIFO(filter_fifo_layout, fifo_size))
        self.submodules.fifo   = ClockDomainsRenamer(clock_domain)(fifo)
        ...
        self.source.connect(fifo.sink, omit={"valid", "ts", "error"}),
        ...

        fsmWriter.act("NO_FILTER",
            NextValue(fifo.sink.valid, 0),
            ...
```

connect() is used to connect a sink to a source. Always use ***source.connect(sink)***

Connect ***self.source*** to ***fifo.sink*** but don't connect ***valid***, ***ts*** and ***error*** (omit).They will be controlled in the module.

LiteX – Streams components

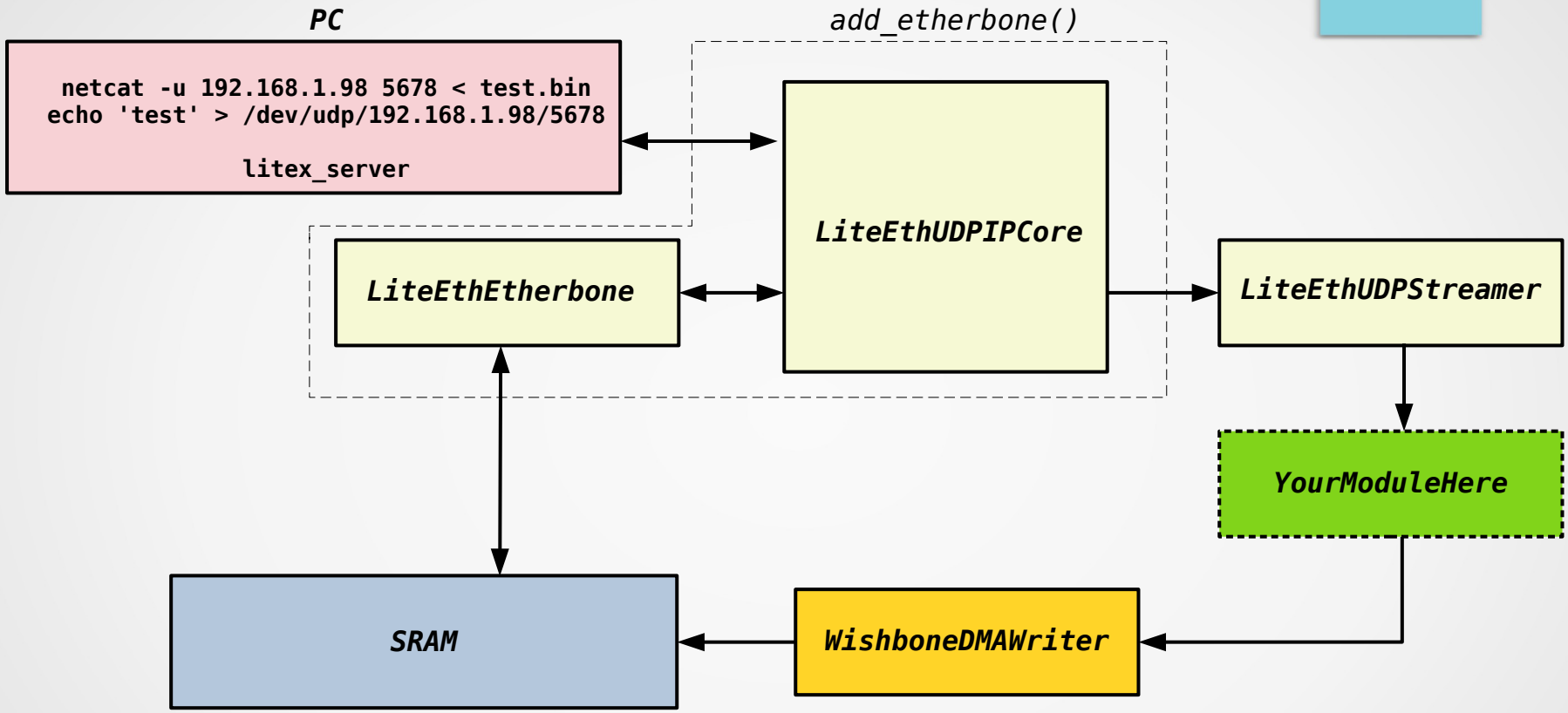
- stream.SyncFIFO
- stream.AsyncFIFO
- stream.ClockDomainCrossing
- stream.Multiplexer
- stream.Demultiplexer
- stream.StrideConverter
- stream.Pipeline
- ...

step13 – Streams workshop

What you will learn:

- Connect and control sinks / sources
- Use Ethernet UDP streamer
- Use WishboneDMAWriter

step13 – Streams workshop



- Receive UDP payload and write it in DRAM
- Check received payload using the BIOS

step13 – Streams workshop

- The SoC has an etherbone and an LiteEthUDPStreamer
- An SRAM memory (sram_udp) is present at 0x20000000
- LiteEthUDPStreamer provides a stream from UDP received frames
- WishboneDMAWriter takes a stream (address, data) and converts it to Wishbone transfers

- Write a module to prepare the stream from LiteEthUDPStreamer to WishboneDMAWriter
- See further instructions in the code

step13 – Obervations

- **Always use xxx.from.connect(yyy.to)**

Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
- LiteX: advanced topics
  - Streams / workshop
  - Usage of Verilog/VHDL modules in LiteX / workshop
  - Verilator / workshop

LiteX – Reuse Verilog/VHDL modules

- ***Verilog/VHDL*** cores can be integrated to Migen/LiteX
- Other description languages (***Spinal-HDL, nMigen***) can be reused through ***Verilog***
- Migen's ***Instance()*** is used to instantiate the core

```
din    = Signal(32)
dout   = Signal(32)
dinout = Signal(32)
self.specials += Instance("custom_core",
    p_DATA_WIDTH = 32,
    i_din     = din,
    o_dout    = dout,
    io_dinout = dinout
)
```

```
platform.add_source("core.v")   # Will automatically add the core as Verilog source.
platform.add_source("core.sv")  # Will automatically add the core as System-Verilog source.
platform.add_source("core.vhd") # Will automatically add the core as VHDL source.
```

LiteX – Reuse Verilog/VHDL modules

```
din    = Signal(32)
dout   = Signal(32)
dinout = Signal(32)
self.specials += Instance("custom_core",
    p_DATA_WIDTH = 32,
    i_din     = din,
    o_dout    = dout,
    io_dinout = dinout
)
```

- Prefixes are used to specify the type of interface

```
p_   for a Parameter           (Python's str or int or Migen's Const).
i_   for an Input port         (Python's int or Migen's Signal, Cat, Slice).
o_   for an Output port        (Python's int or Migen's Signal, Cat, Slice).
io_  for a Bi-Directional port (Migen's Signal, Cat, Slice).
```

LiteX – Reuse Verilog/VHDL modules

- LiteX automatically determines the language based on the file extension

```
platform.add_source("core.v")   # Will automatically add the core as Verilog source.
platform.add_source("core.sv")  # Will automatically add the core as System-Verilog source.
platform.add_source("core.vhd") # Will automatically add the core as VHDL source.
```

- It is possible to pass multiple sources at once

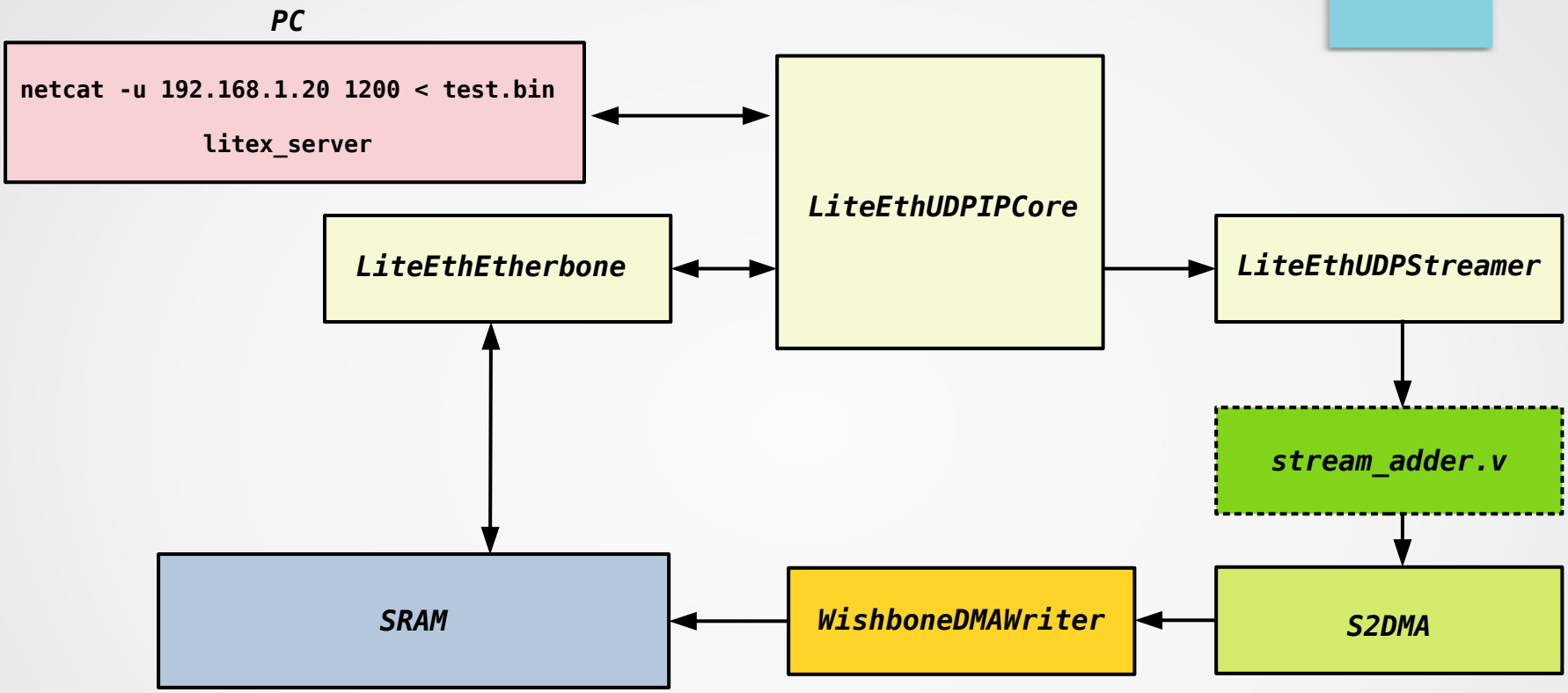
```
platform.add_sources(path="./",
  "core0.v",
  "core1.vhd",
  "core2.sv"
)
```

step14 – Reuse Verilog/VHDL modules

What you will learn:

- Use an external verilog core
- Use litex_read_verilog

step14 – Reuse Verilog/VHDL modules



- Create ***StreamAddOne*** module from ***stream_adder.v*** and add it to the system
- Check received payload using the BIOS

step14 – Reuse Verilog/VHDL modules

- Use litex_read_verilog to generate a Migen class from the verilog file

- Create a StreamAddOne module with a sink and a source stream port and connect your stream_adder inside this module

- Insert StreamAddOne between the udp_streamer and S2DMA

Agenda

- Description of FPGAs
- Digital design basics
- Migen: introduction and workshops
- LiteX: introduction and workshops
- LiteX: advanced topics
  - Streams / workshop
  - Usage of Verilog/VHDL modules in LiteX / workshop
  - Verilator / workshop

LiteX – What is Verilator ?

- Verilog / SystemVerilog simulator
- Accept only synthesizable structures
- Converts Verilog into multithreaded C++ or SystemC model
- Generates a .cpp and .h file, the ***Verilated*** code
- Write a test bench with an instance of the Verilated model
- Get an executable that runs the simulation
- **Very fast**

LiteX – Verilator infrastructure

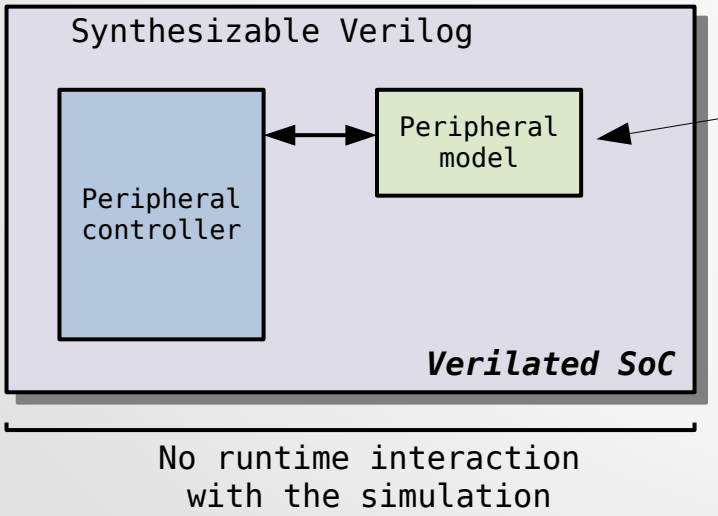
- LiteX provides a Verilator simulation framework
- Verilator models for **DRAM, SPI Flash, SD-Card**
- Verilator models for **Ethernet** and **serial** (interactive)
- Modular conception. Modules can easily be added
- **litex_sim** is a ready to use simulated SoC (with all available simulated peripherals)

LiteX – Simulation model

- System simulation needs model for external interfaces

Two ways:

- Write a synthesizable model



The simulated peripheral is written in Migen like any other Module.

Only build time configuration

No user interaction

LiteX – Simulation model

- System simulation needs model for external interfaces

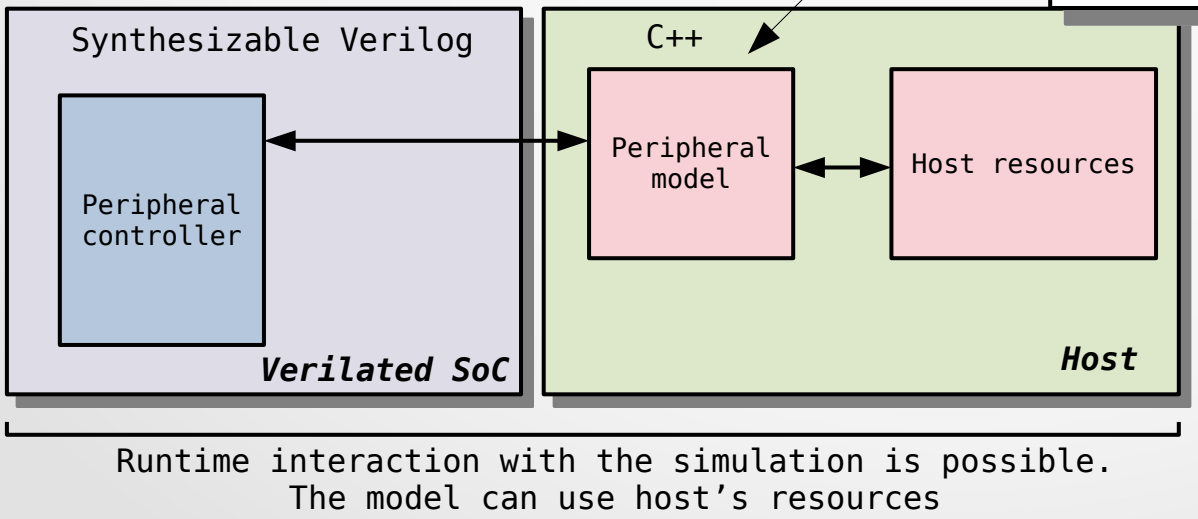
Two ways:

- Write a synthesizable model
- Write a C++ model

The simulated peripheral is written in C++ and it will not be part of the Verilated code

Can use host's resources

Signals must be present on the top level of your SoC



LiteX – Writing a model, synthesizable

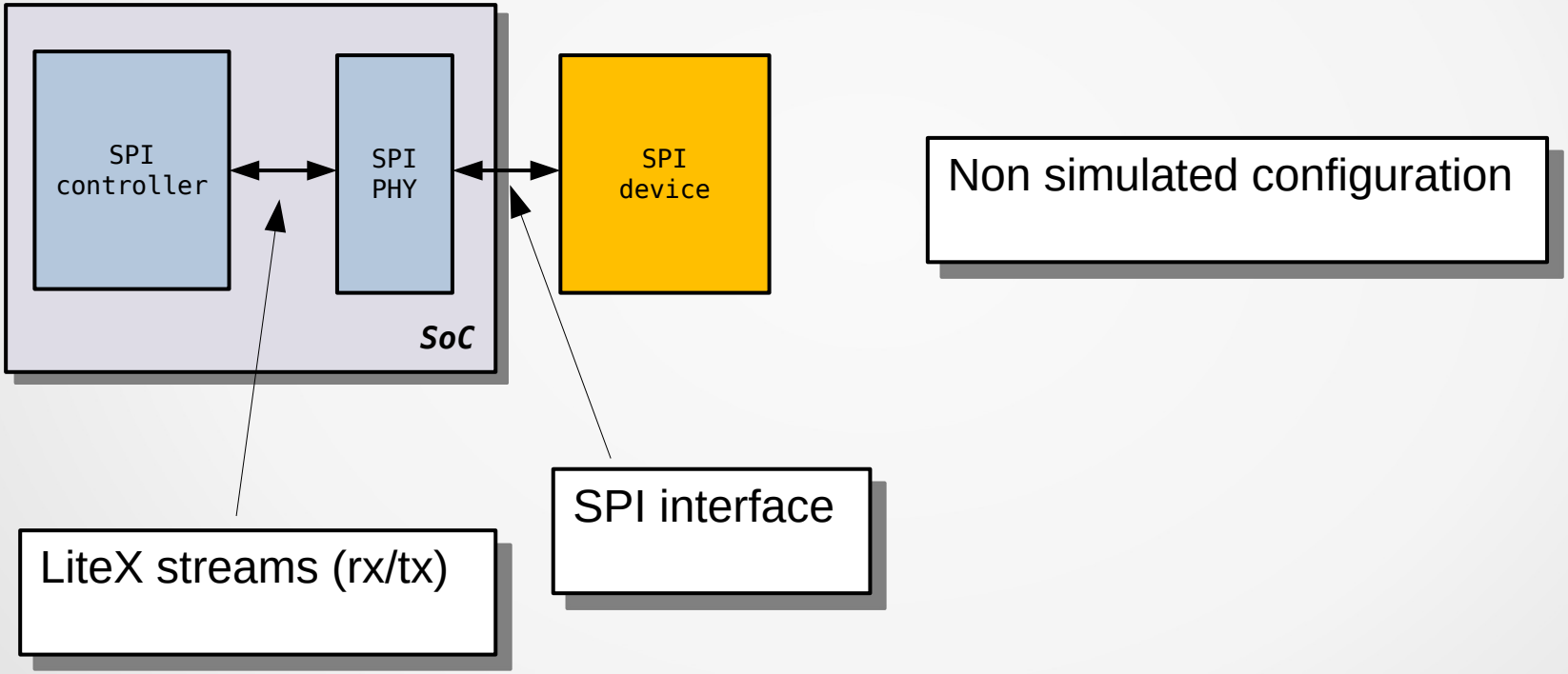
- Writing a model with Migen code is not specific to simulation
- The model is synthesizable but resources are not important
- In general, fully equivalent to the real interface
- See `LiteSPIPHYModel` in `litespi/litespi/phy/model.py`

```
self.submodules.spiflash_phy = LiteSPIPHYModel(spiflash_module, ...
self.add_spi_flash(phy=self.spiflash_phy, ...
```

```
def add_spi_flash(self, name="spiflash", ..., phy=None...
    ...
        spiflash_phy = phy
        if spiflash_phy is None:
            ...
            spiflash_phy = LiteSPIPHY(spiflash_pads, ...
            setattr(self.submodules, name + "_phy",  spiflash_phy)
```

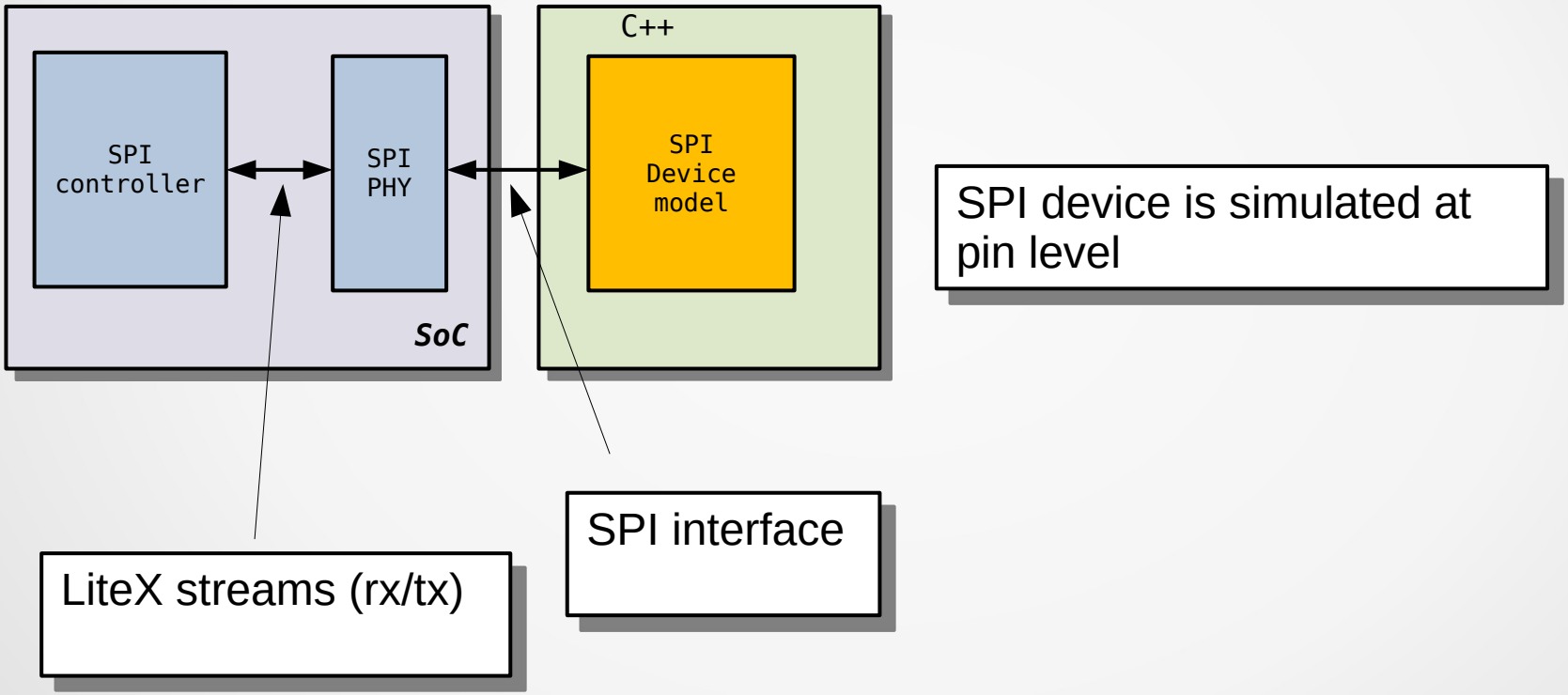
LiteX – Writing a C++ model

- Simulation can be at pins level or interface level



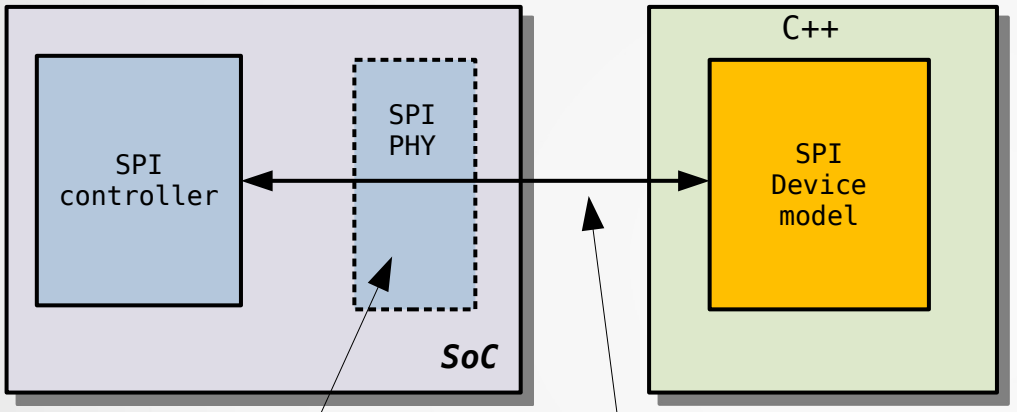
LiteX – Writing a C++ model

- Simulation can be at pins level or interface level



LiteX – Writing a C++ model

- Simulation can be at pins level or interface level



SPI device is simulated at interface level

Easier, faster

LiteX streams (rx/tx)

SPI PHY is just a pass-through module to expose the LiteX streams to the SoC interface

LiteX – Writing a C++ model, structure

- New modules must be declared during build:

```
builder.build(
    extra_mods = ["ledring"],
    extra_mods_path = os.path.abspath(os.getcwd()) + "/modules",
    sim_config=sim_config
)
```

- Must provides and register a **struct ext_module_s**

```
struct ext_module_s {
    char *name;
    int (*start)(void *);
    int (*new_sess)(void **, char *);
    int (*add_pads)(void *, struct pad_list_s *);
    int (*close)(void*);
    int (*tick)(void*, uint64_t);
};
```

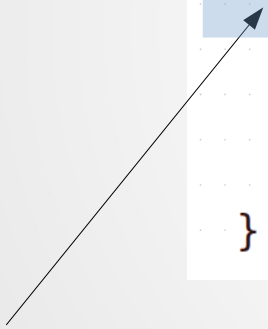
Name of this module.

Will be used to add it to the simulation.

LiteX – Writing a C++ model, structure

- Must provides and register a **struct ext_module_s**

```
struct ext_module_s {
  char *name;
  int (*start)(void *);
  int (*new_sess)(void **, char *);
  int (*add_pads)(void *, struct pad_list_s *);
  int (*close)(void*);
  int (*tick)(void*, uint64_t);
};
```



Called once during startup

LiteX – Writing a C++ model, structure

- Must provides and register a **struct ext_module_s**

```
struct ext_module_s {
  char *name;
  int (*start)(void *);
  int (*new_sess)(void **, char *);
  int (*add_pads)(void *, struct pad_list_s *);
  int (*close)(void*);
  int (*tick)(void*, uint64_t);
};
```

Must provides a user's defined session information.

This will be available in other callbacks.

LiteX – Writing a C++ model, structure

- Must provides and register a **struct ext_module_s**

```
struct ext_module_s {
  char *name;
  int (*start)(void *);
  int (*new_sess)(void **, char *);
  int (*add_pads)(void *, struct pad_list_s *);
  int (*close)(void*);
  int (*tick)(void*, uint64_t);
};
```

This is where you get and save pointers to your pads

LiteX – Writing a C++ model, structure

- Must provides and register a **struct ext_module_s**

```
struct ext_module_s {
  char *name;
  int (*start)(void *);
  int (*new_sess)(void **, char *);
  int (*add_pads)(void *, struct pad_list_s *);
  int (*close)(void*);
  int (*tick)(void*, uint64_t);
};
```

Called once during the end of simulation

LiteX – Writing a C++ model, structure

- Must provides and register a **struct ext_module_s**

```
struct ext_module_s {
  char *name;
  int (*start)(void *);
  int (*new_sess)(void **, char *);
  int (*add_pads)(void *, struct pad_list_s *);
  int (*close)(void*);
  int (*tick)(void*, uint64_t);
};
```

Called every clock cycle

LiteX – Writing a C++ model, example

- **serial2console** is a terminal emulator
- Gets input/output from UART to your console
- `litex/build/sim/core/modules/serial2console/serial2console.c`

```
static struct ext_module_s ext_mod = {
  "serial2console",
  serial2console_start,
  serial2console_new,
  serial2console_add_pads,
  NULL,
  serial2console_tick
};

int litex_sim_ext_module_init(int (*register_module) (struct ext_module_s *))
{
  int ret = RC_OK;
  ret = register_module(&ext_mod);
  return ret;
}
```

Definition of the Module

Called by LiteX infrastructure

Add the module to the list

LiteX – Writing a C++ model, example

```
_io = [
    ...
    ("serial", 0,
        Subsignal("source_valid", Pins(1)),
        Subsignal("source_ready", Pins(1)),
        Subsignal("source_data",  Pins(8)),

        Subsignal("sink_valid",   Pins(1)),
        Subsignal("sink_ready",   Pins(1)),
        Subsignal("sink_data",    Pins(8)),
    ),
    ...
```

UART pads from the platform definition.

This UART model use streams, not UART pins.

```
static struct ext_module_s ext_mod = {
  "serial2console",
  serial2console_start,
  serial2console_new,
  serial2console_add_pads,
  NULL,
  serial2console_tick
};

int litex_sim_ext_module_init(int (*register_module) (struct ext_module_s *))
{
  int ret = RC_OK;
  ret = register_module(&ext_mod);
  return ret;
}
```

Module added to the simulation

```
sim_config = SimConfig()
sim_config.add_module("serial2console", "serial")
```

pads

LiteX – Writing a C++ model, example

```
_io = [
    ...
    ("serial", 0,
        Subsignal("source_valid", Pins(1)),
        Subsignal("source_ready", Pins(1)),
        Subsignal("source_data",  Pins(8)),

        Subsignal("sink_valid",   Pins(1)),
        Subsignal("sink_ready",   Pins(1)),
        Subsignal("sink_data",    Pins(8)),
    ),
    ...
```

```
static struct ext_module_s ext_mod = {
  "serial2console",
  serial2console_start,
  serial2console_new,
  serial2console_add_pads,
  NULL,
  serial2console_tick
};

int litex_sim_ext_module_init(int (*register_module) (struct ext_module_s *))
{
  int ret = RC_OK;
  ret = register_module(&ext_mod);
  return ret;
}
```

```
sim_config = SimConfig()
sim_config.add_module("serial2console", "serial")
```

Configuration of the terminal

Allocation of session's structure

Get pads from Verilated code

Close callback is not used

Handle data transfers

LiteX – Writing a C++ model, example

```
_io = [
    ...
    ("serial", 0,
        Subsignal("source_valid", Pins(1)),
        Subsignal("source_ready", Pins(1)),
        Subsignal("source_data",  Pins(8)),

        Subsignal("sink_valid",   Pins(1)),
        Subsignal("sink_ready",   Pins(1)),
        Subsignal("sink_data",    Pins(8)),
    ),
    ...
```

```
static struct ext_module_s ext_mod = {
  "serial2console",
  serial2console_start,
  serial2console_new,
  serial2console_add_pads,
  NULL,
  serial2console_tick
};

int litex_sim_ext_module_init(int (*register_module) (struct ext_module_s *))
{
  int ret = RC_OK;
  ret = register_module(&ext_mod);
  return ret;
}
```

Get pads from Verilated code

```
sim_config = SimConfig()
sim_config.add_module("serial2console", "serial")
```

LiteX – Writing a C++ model, example

```
struct session_s {
  char *tx;
  char *tx_valid;
  char *tx_ready;
  ...
};
```

session's data

Called for every interface passed in ***add_module*** + clocks

```
static int serial2console_add_pads(void *sess, struct pad_list_s *plist)
{
  ...
  struct session_s *s = (struct session_s*) sess;
  struct pad_s *pads;
  ...
  pads = plist->pads;
  if(!strcmp(plist->name, "serial")) {
    litex_sim_module_pads_get(pads, "sink_data", (void**)&s->rx);
    litex_sim_module_pads_get(pads, "sink_valid", (void**)&s->rx_valid);
    litex_sim_module_pads_get(pads, "sink_ready", (void**)&s->rx_ready);
    litex_sim_module_pads_get(pads, "source_data", (void**)&s->tx);
    litex_sim_module_pads_get(pads, "source_valid", (void**)&s->tx_valid);
    litex_sim_module_pads_get(pads, "source_ready", (void**)&s->tx_ready);
  }

  if(!strcmp(plist->name, "sys_clk"))
    litex_sim_module_pads_get(pads, "sys_clk", (void**) &s->sys_clk);
  ...
}
```

session's data has now a pointer to control or read each pad

LiteX – Writing a C++ model, example

```
_io = [
    ...
    ("serial", 0,
        Subsignal("source_valid", Pins(1)),
        Subsignal("source_ready", Pins(1)),
        Subsignal("source_data",  Pins(8)),

        Subsignal("sink_valid",   Pins(1)),
        Subsignal("sink_ready",   Pins(1)),
        Subsignal("sink_data",    Pins(8)),
    ),
    ...
```

```
static struct ext_module_s ext_mod = {
  "serial2console",
  serial2console_start,
  serial2console_new,
  serial2console_add_pads,
  NULL,
  serial2console_tick
};

int litex_sim_ext_module_init(int (*register_module) (struct ext_module_s *))
{
  int ret = RC_OK;
  ret = register_module(&ext_mod);
  return ret;
}
```

Execute on every simulation cycle

```
sim_config = SimConfig()
sim_config.add_module("serial2console", "serial")
```

LiteX – Writing a C++ model, example

```
static int serial2console_tick(void *sess, uint64_t time_ps) {
  static clk_edge_state_t edge;
  struct session_s *s = (struct session_s*)sess;

  if(!clk_pos_edge(&edge, *s->sys_clk)) {
    return RC_OK;
  }

  *s->tx_ready = 1;
  if(*s->tx_valid) {
    printf("%c", *s->tx);
    fflush(stdout);
  }

  *s->rx_valid = 0;
  if(s->datalen) {
    *s->rx = s->databuf[s->data_start];
    s->data_start = (s->data_start + 1) % 2048;
    s->datalen--;
    *s->rx_valid = 1;
  }

  return RC_OK;
}
```

Check if we are in a clock's rising edge

LiteX – Writing a C++ model, example

```
static int serial2console_tick(void *sess, uint64_t time_ps) {
  static clk_edge_state_t edge;
  struct session_s *s = (struct session_s*)sess;

  if(!clk_pos_edge(&edge, *s->sys_clk)) {
    return RC_OK;
  }

  *s->tx_ready = 1;
  if(*s->tx_valid) {
    printf("%c", *s->tx);
    fflush(stdout);
  }

  *s->rx_valid = 0;
  if(s->datalen) {
    *s->rx = s->databuf[s->data_start];
    s->data_start = (s->data_start + 1) % 2048;
    s->datalen--;
    *s->rx_valid = 1;
  }

  return RC_OK;
}
```

We are always ready to receive characters

Print a valid data

LiteX – Writing a C++ model, example

```
static int serial2console_tick(void *sess, uint64_t time_ps) {
  static clk_edge_state_t edge;
  struct session_s *s = (struct session_s*)sess;

  if(!clk_pos_edge(&edge, *s->sys_clk)) {
    return RC_OK;
  }

  *s->tx_ready = 1;
  if(*s->tx_valid) {
    printf("%c", *s->tx);
    fflush(stdout);
  }

  *s->rx_valid = 0;
  if(s->datalen) {
    *s->rx = s->databuf[s->data_start];
    s->data_start = (s->data_start + 1) % 2048;
    s->datalen--;
    *s->rx_valid = 1;
  }

  return RC_OK;
}
```

By default, no character is sent

Send any available character

LiteX – Minimal Verilator simulation

```
...
from litex.build.sim import SimPlatform
from litex.build.sim.config import SimConfig
...

_io = [
    ("sys_clk", 0, Pins(1)),
    ...
]

class Platform(SimPlatform):
    def __init__(self):
        SimPlatform.__init__(self, "SIM", _io)

class BenchSoC(SoCCore):
        ...
        SoCMini.__init__(self, platform, clk_freq=sys_clk_freq,
            ident          = "LiteEth bench Simulation",
            ident_version  = True
        )
        ...
        self.submodules.crg = CRG(platform.request("sys_clk"))

def main():
    ...
    sim_config = SimConfig()
    sim_config.add_clocker("sys_clk", freq_hz=1e6)

    soc     = BenchSoC()
    builder = Builder(soc, csr_csv="csr.csv")
    builder.build(sim_config=sim_config)
```

Use SimPlatform

LiteX – Minimal Verilator simulation

```
...
from litex.build.sim import SimPlatform
from litex.build.sim.config import SimConfig
...

_io = [
    ("sys_clk", 0, Pins(1)),
    ...
]

class Platform(SimPlatform):
    def __init__(self):
        SimPlatform.__init__(self, "SIM", _io)

class BenchSoC(SoCCore):
        ...
        SoCMini.__init__(self, platform, clk_freq=sys_clk_freq,
            ident          = "LiteEth bench Simulation",
            ident_version  = True
        )
        ...
        self.submodules.crg = CRG(platform.request("sys_clk"))

def main():
    ...
    sim_config = SimConfig()
    sim_config.add_clocker("sys_clk", freq_hz=1e6)

    soc     = BenchSoC()
    builder = Builder(soc, csr_csv="csr.csv")
    builder.build(sim_config=sim_config)
```

Use your SoC as usual

LiteX – Minimal Verilator simulation

```
...
from litex.build.sim import SimPlatform
from litex.build.sim.config import SimConfig
...

_io = [
    ("sys_clk", 0, Pins(1)),
    ...
]

class Platform(SimPlatform):
    def __init__(self):
        SimPlatform.__init__(self, "SIM", _io)

class BenchSoC(SoCCore):
        ...
        SoCMini.__init__(self, platform, clk_freq=sys_clk_freq,
            ident          = "LiteEth bench Simulation",
            ident_version  = True
        )
        ...
        self.submodules.crg = CRG(platform.request("sys_clk"))

def main():
    ...
    sim_config = SimConfig()
    sim_config.add_clocker("sys_clk", freq_hz=1e6)

    soc     = BenchSoC()
    builder = Builder(soc, csr_csv="csr.csv")
    builder.build(sim_config=sim_config)
```

Add a ***clocker*** module to generate the clock from the C++ test bench.

```
self.add_module("clocker",...
```

LiteX – Minimal Verilator simulation

```
...
from litex.build.sim import SimPlatform
from litex.build.sim.config import SimConfig
...

_io = [
    ("sys_clk", 0, Pins(1)),
    ...
]

class Platform(SimPlatform):
    def __init__(self):
        SimPlatform.__init__(self, "SIM", _io)

class BenchSoC(SoCCore):
        ...
        SoCMini.__init__(self, platform, clk_freq=sys_clk_freq,
            ident          = "LiteEth bench Simulation",
            ident_version  = True
        )
        ...
        self.submodules.crg = CRG(platform.request("sys_clk"))

def main():
    ...
    sim_config = SimConfig()
    sim_config.add_clocker("sys_clk", freq_hz=1e6)

    soc     = BenchSoC()
    builder = Builder(soc, csr_csv="csr.csv")
    builder.build(sim_config=sim_config)
```

Run the simulation with given parameters

step15 – Verilator simulation

What you will learn:

- Build a Verilator simulation of the Ring Controller
- Use litex_server and every tools on the simulated system