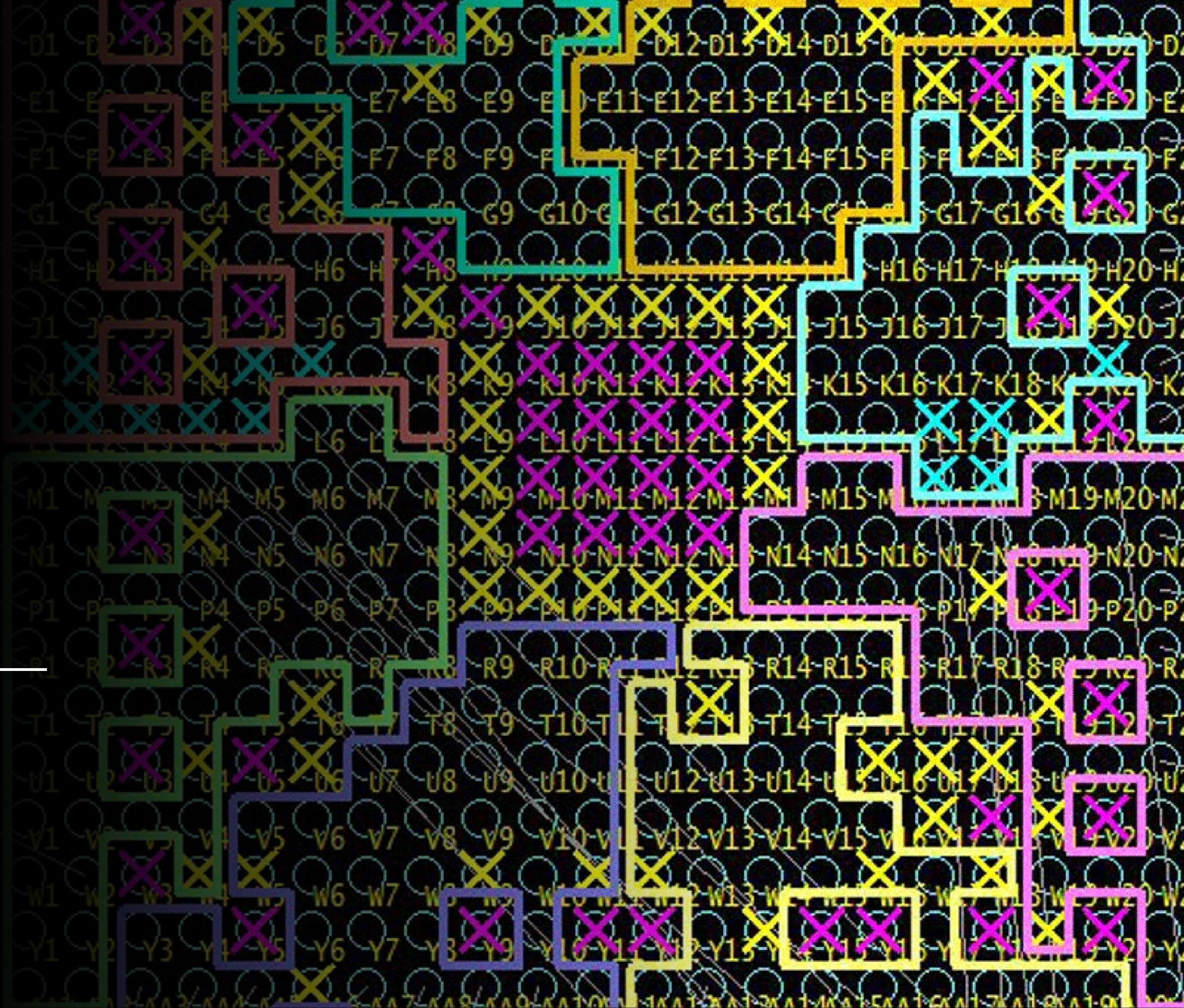
SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078
2022 Introduction to FPGAs Lecture

UNIVERSITY OF TURKU

FPGAs



FPGA Facts

A type of integrated circuit (IC) that can be programmed

Up to two million logic cells that can be configured

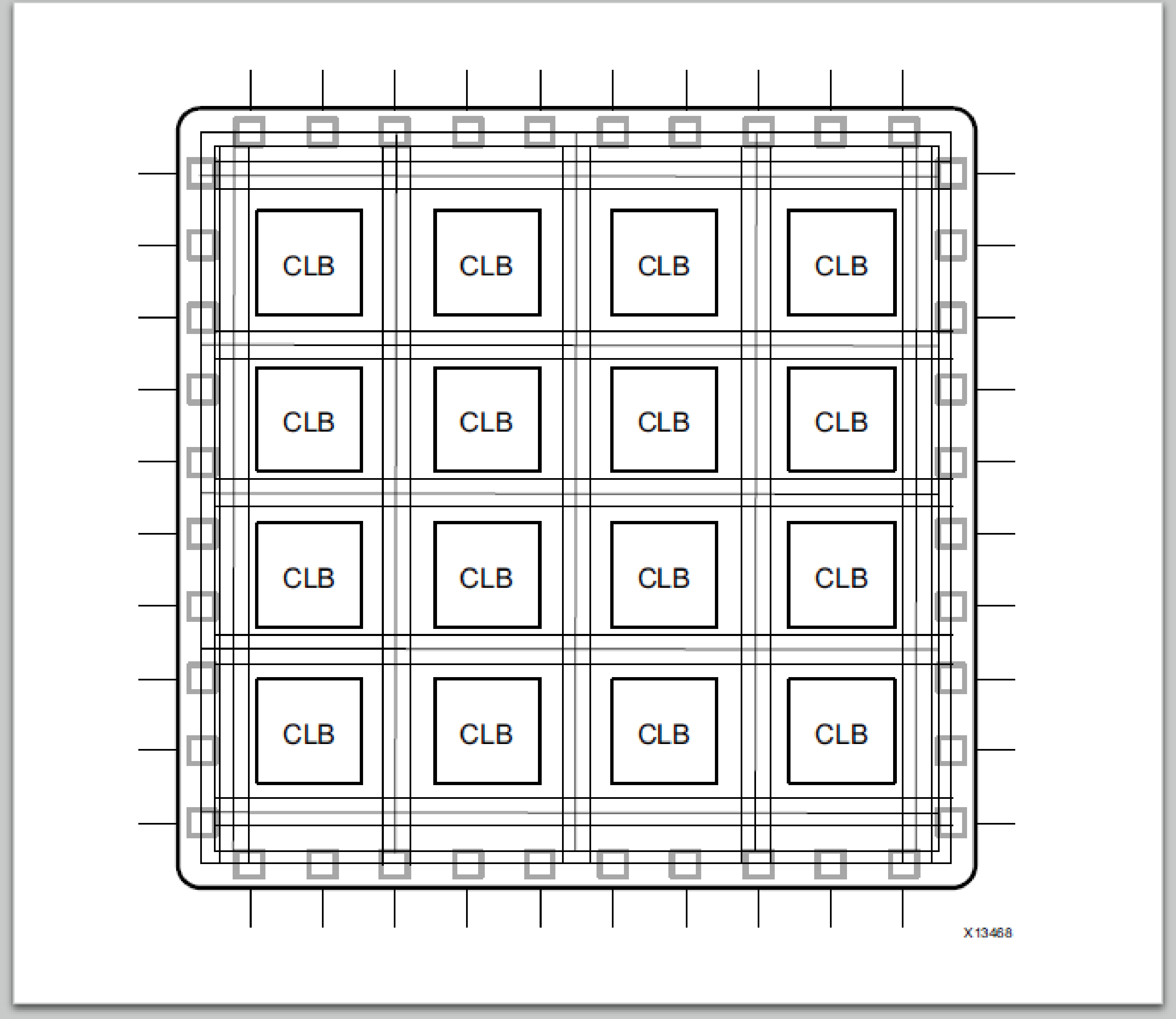
Ability to be dynamically reconfigured

Basic FPGA Architecture

- Look-up table (LUT)
- Flip-Flop (FF)
- Wires
- Input/Output (I/O) pads

Basic FPGA Architecture

FPGA Architecture with Reconfigurable Logic Blocks

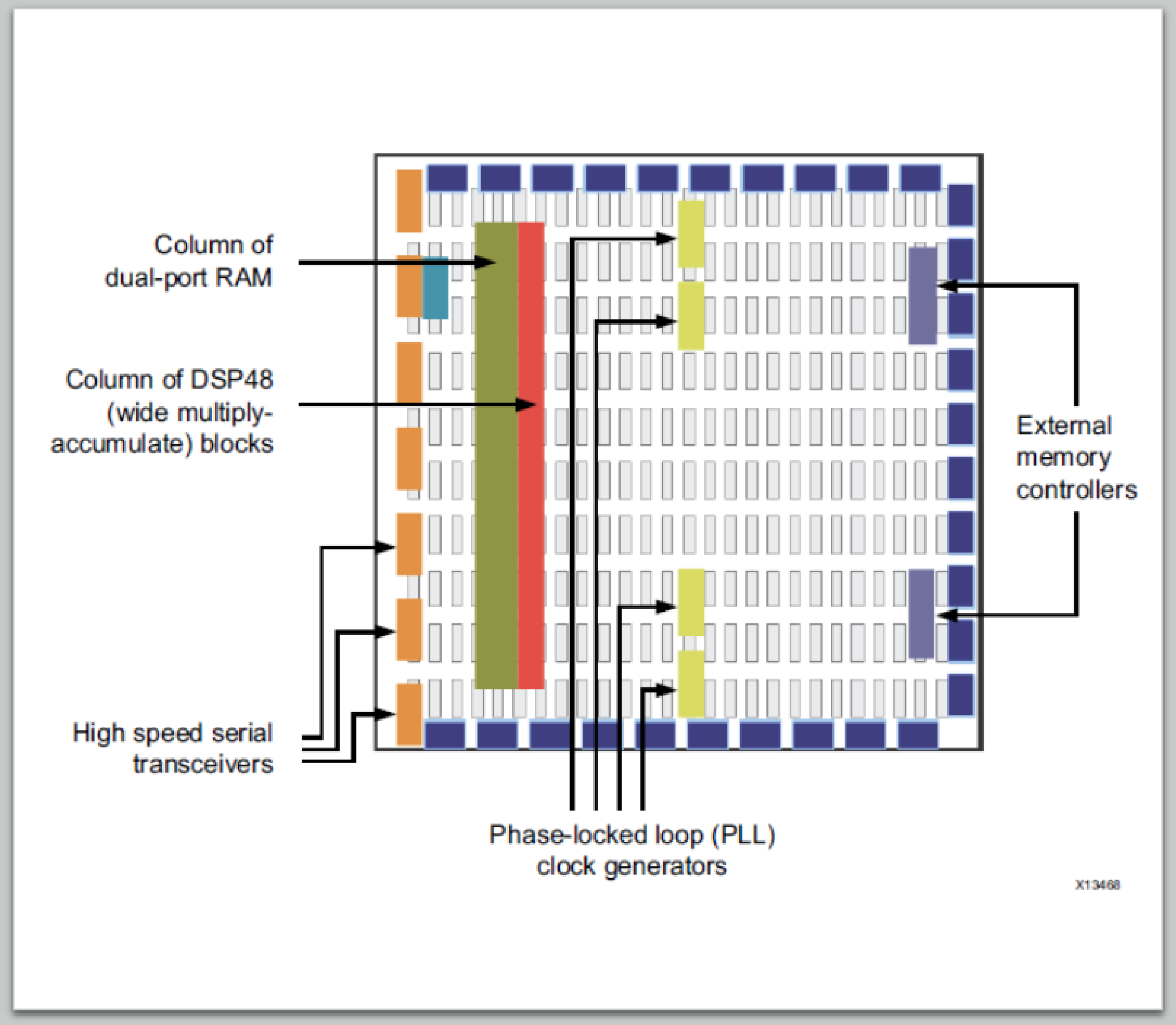


Modern FPGA Architecture

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

Modern FPGA Architecture

FPGA Architecture with CLBs and embedded components



Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL) clock generators
X13468

Building Blocks

(1) LUT

Most basic building block

Implements any logic function with a truth table

An N-input LUT can access up to 2^N memory locations

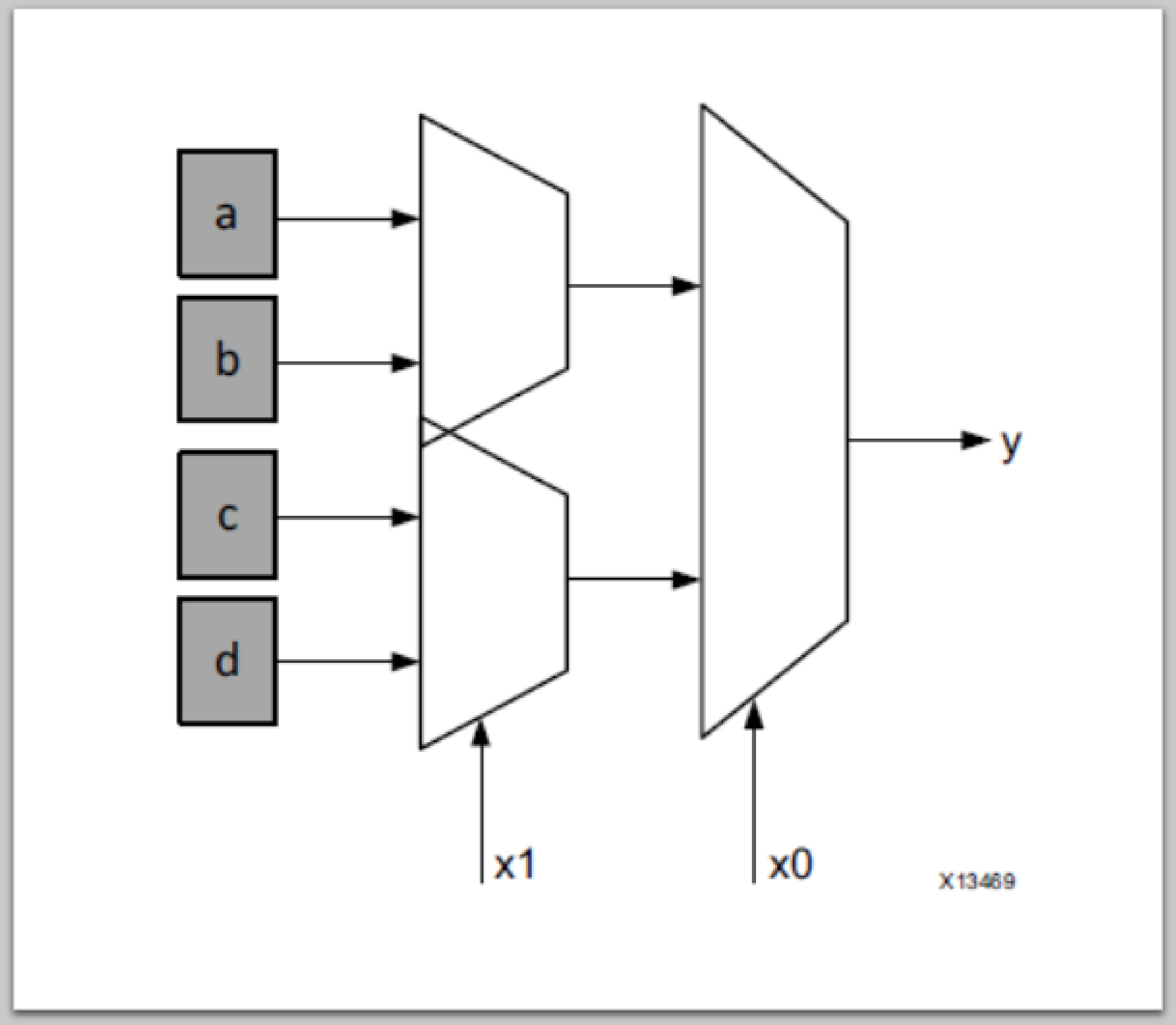
This gives a total of 2^{N^N} different functions that a LUT can implement

- A typical value in Xilinx is N=6

UNIVERSITY OF TURKU

(1) LUT

Functional Representation of a LUT as a Collection of Memory Cells



(2) FLIP-FLOP

Basic storage unit

Always paired with a LUT

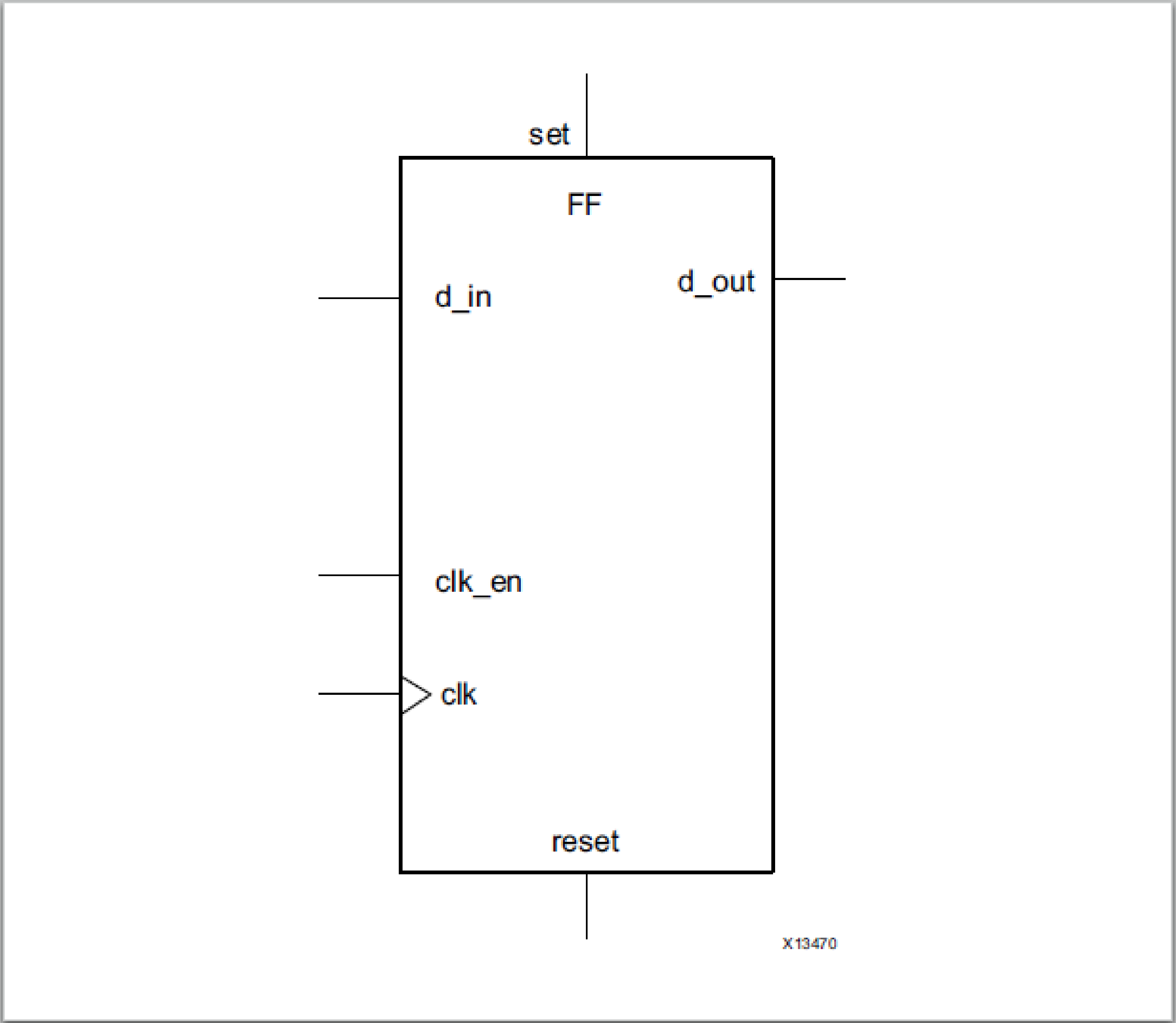
Latches data and synchronizes the output with the clock cycle

Includes: data input, clock input, clock enable, reset and data output

UNIVERSITY OF TURKU

(2) FLIP-FLOP

Structure of a flip-flop with key inputs/outputs



(3) DSP Blocks

Most complex computational unit

It is an arithmetic logic unit (ALU)

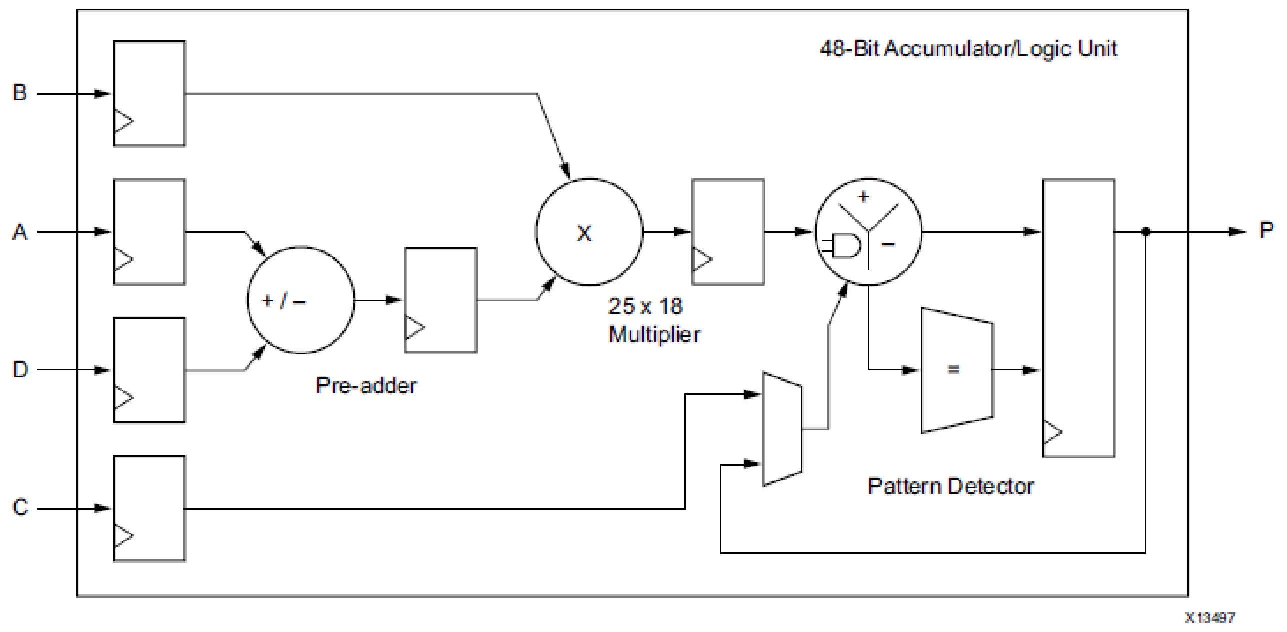
Composed of three blocks: add/subtract + multiply + add/subtract/accumulate

Can implement	r = a(b+c) - d
	r += a(b+c)
Cannot implement	r += a(b+c) - d

UNIVERSITY OF TURKU

(3) DSP Blocks

Structure of a DSP Block with available operations (order matters!)



(4) Storage Elements

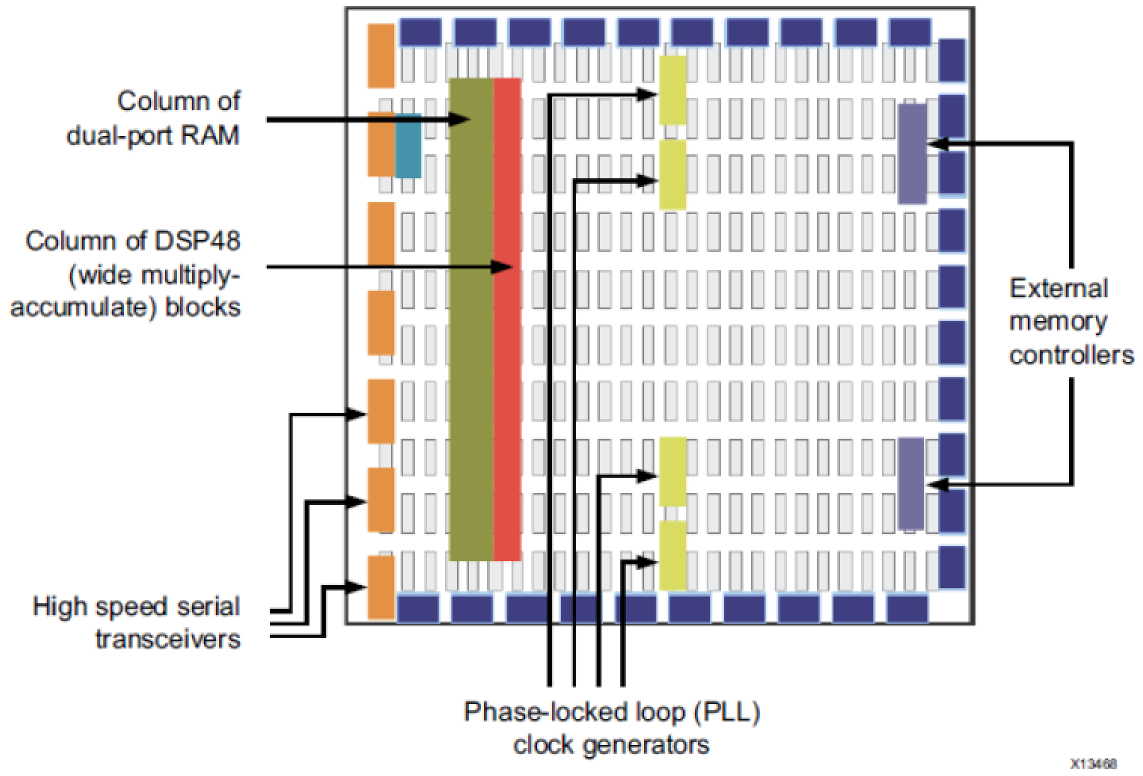
Random Access Memory (RAM), Read-Only Memory (ROM), Shift Registers (SR)

The elements in the FPGA are block RAMs (BRAM), UltraRAM blocks (URAMS), LUTs and SRLs	BRAM is a dual-port RAM, can implement RAM or ROM
	URAMs provide 8x storage capacity of BRAMs and are available in Xilinx Ultrascale+ devices
LUTs are the fastest memory on FPGAs	Flexible location
	64-bit memories (distributed memories)
Shift registers are mainly used for chain operations	For example, a filter represented by a chain of multipliers

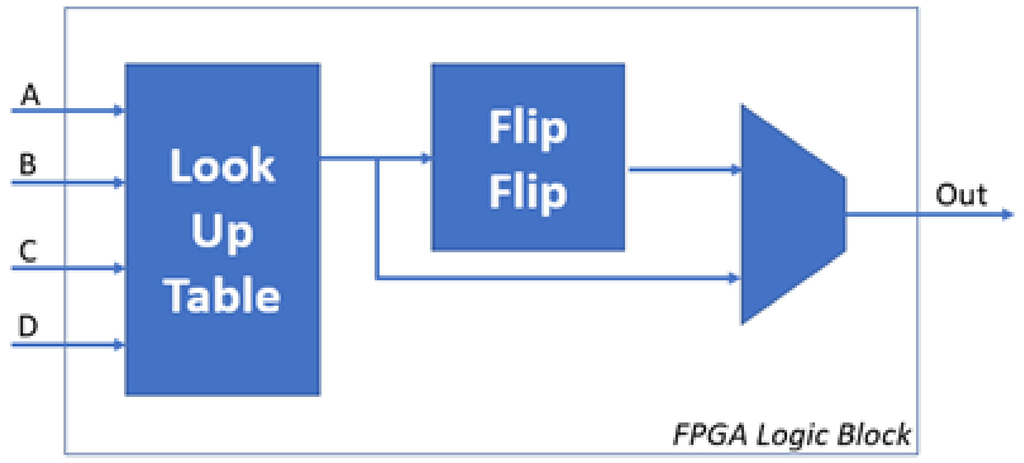
UNIVERSITY OF TURKU

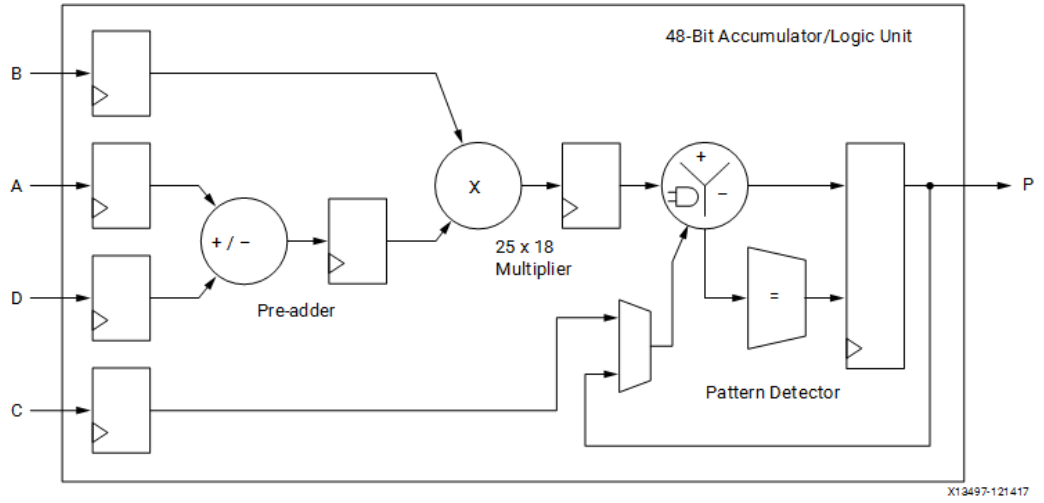
(3) Storage Elements

Storage elements can be found in different places around the FPGA fabric.

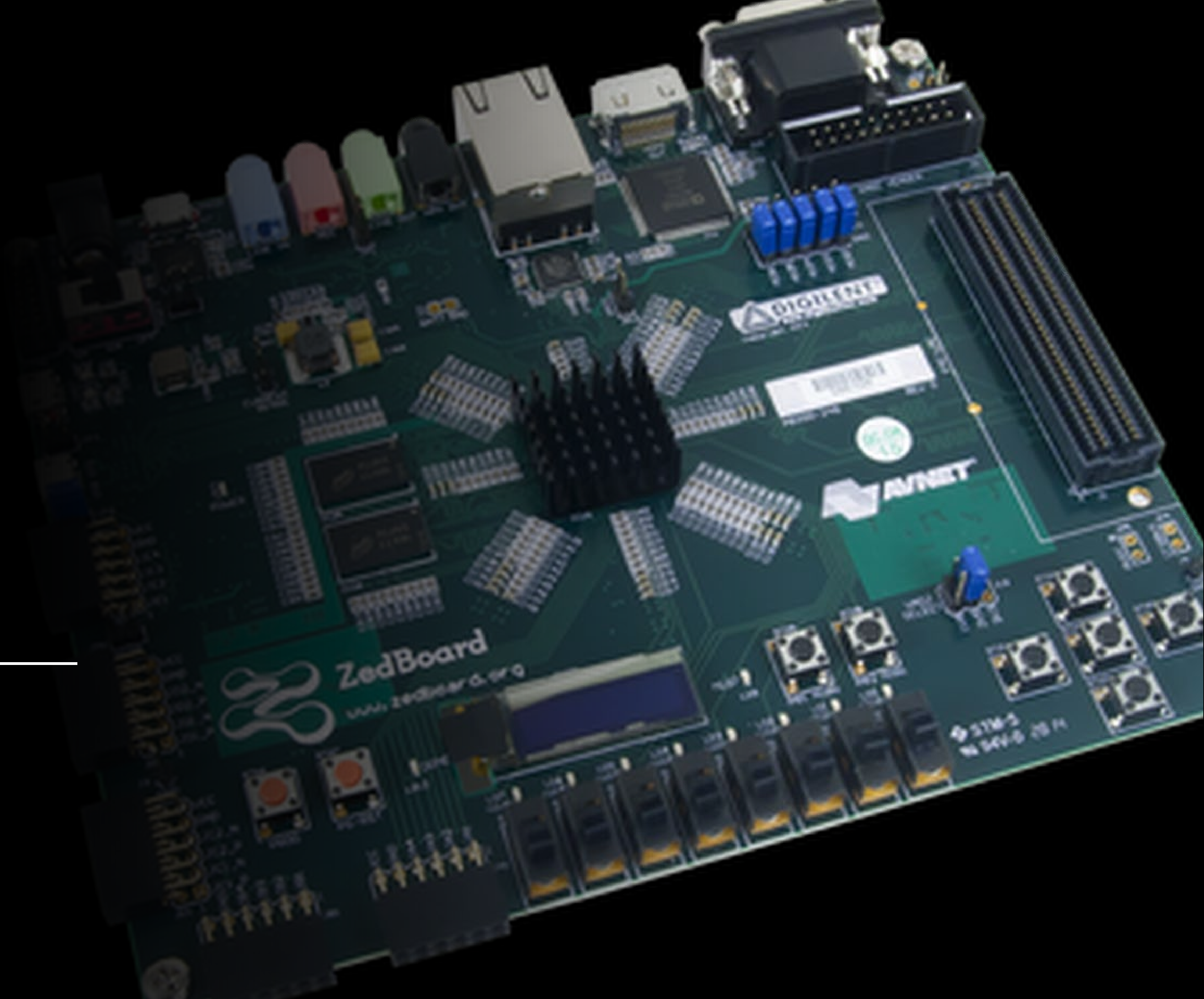

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL) clock generators
X13468

Summary of Components



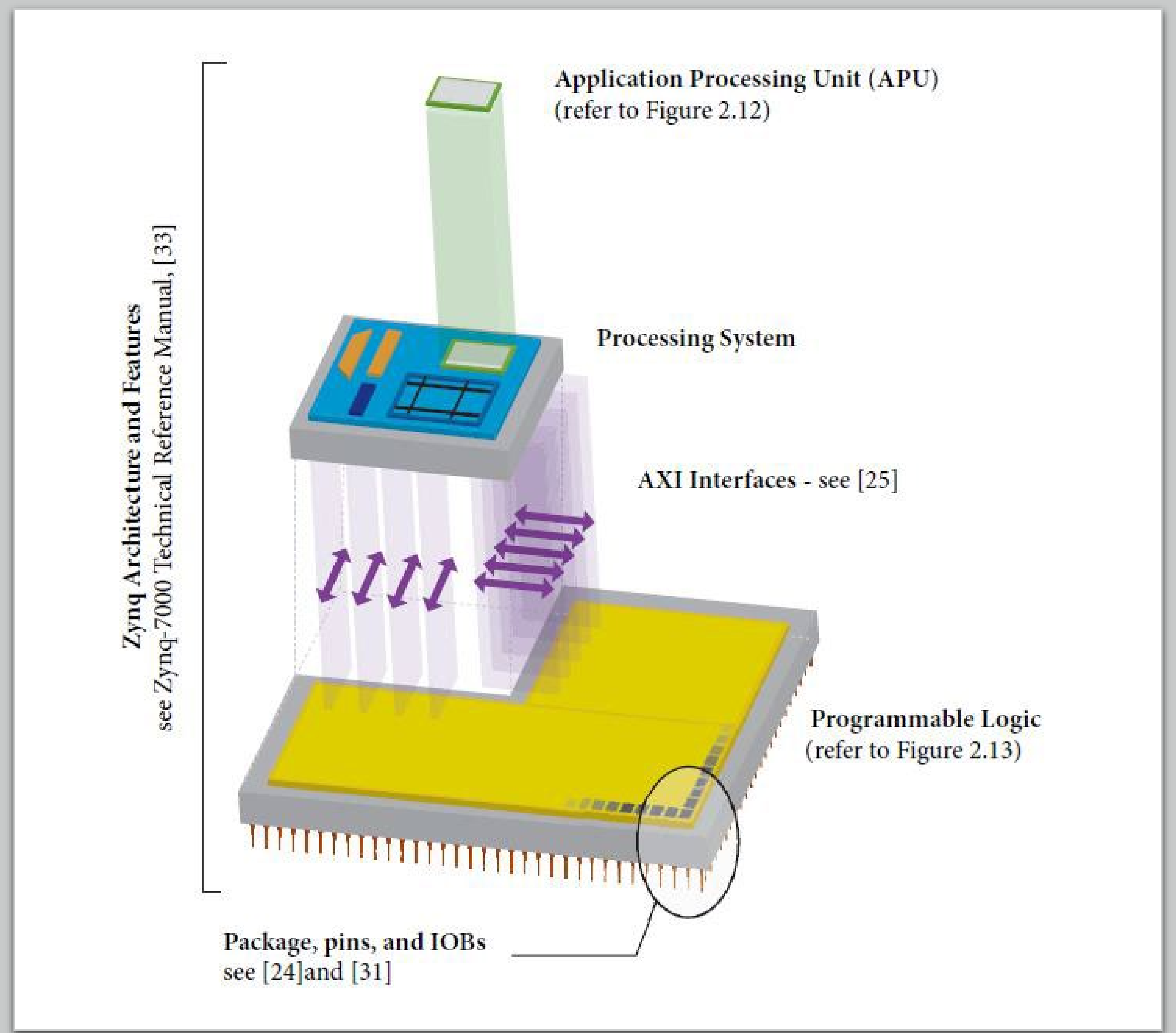


The Zedboard



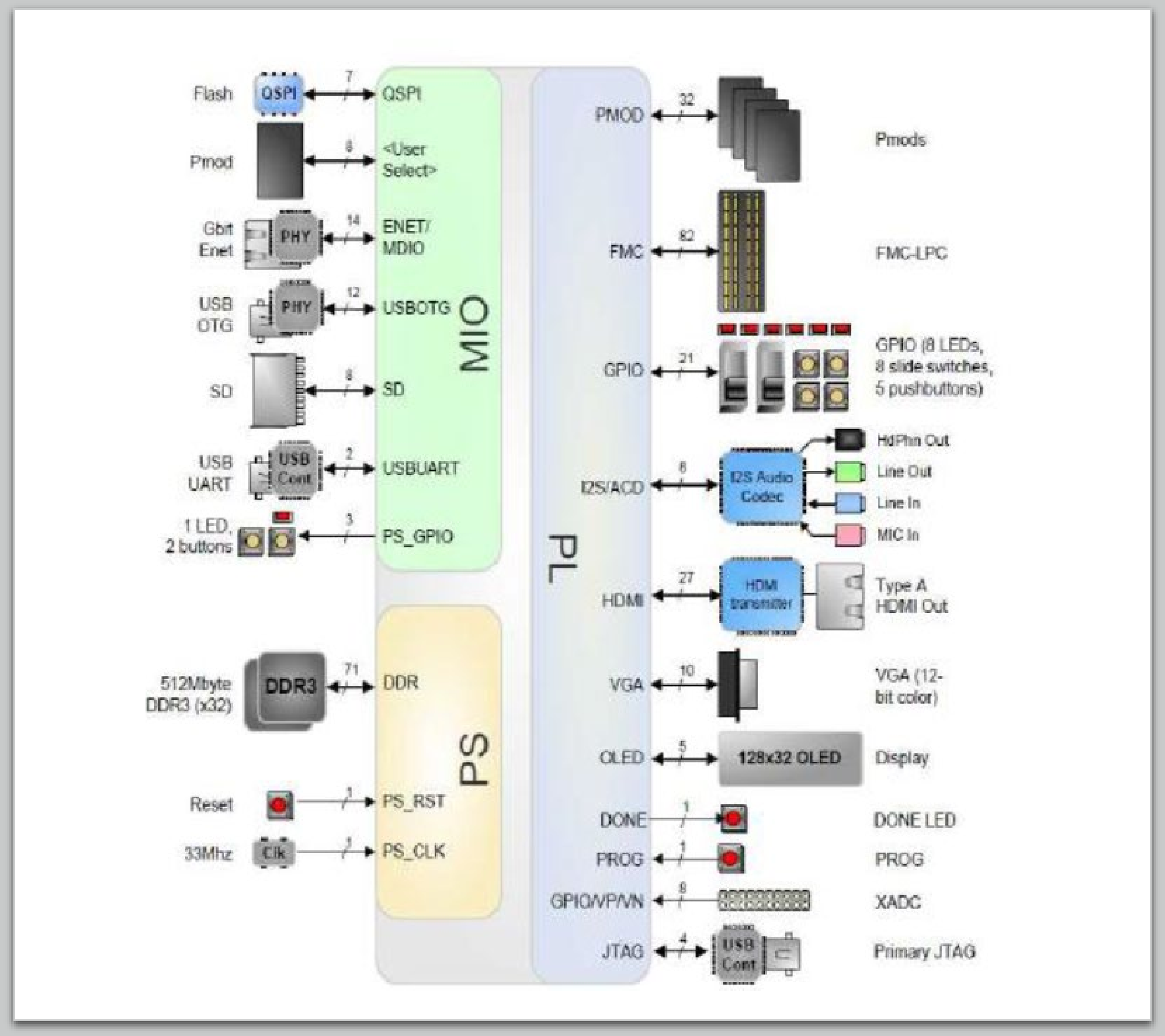
PS and PL

Processing System
Programmable Logic


Zynq Architecture and Features
see Zynq-7000 Technical Reference Manual, [33]
Application Processing Unit (APU)
(refer to Figure 2.12)
Processing System
AXI Interfaces - see [25]
Programmable Logic
(refer to Figure 2.13)
Package, pins, and IOBs
see [24]and [31]

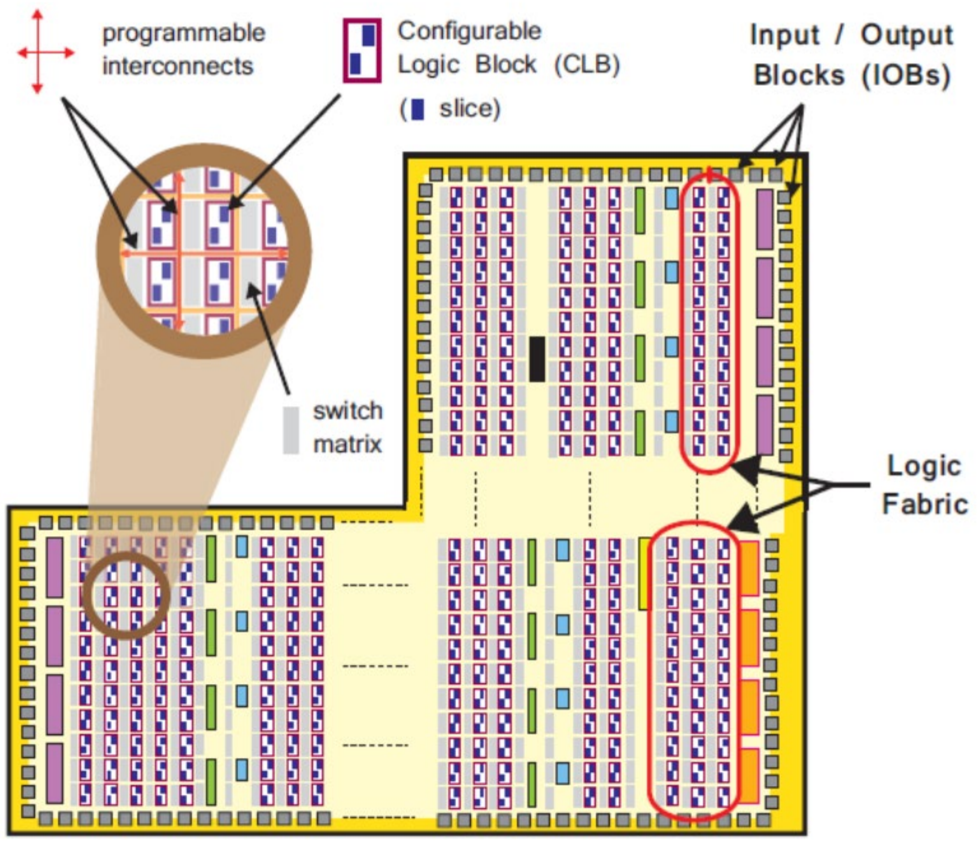
AXI Interface

PS - PL - I/O

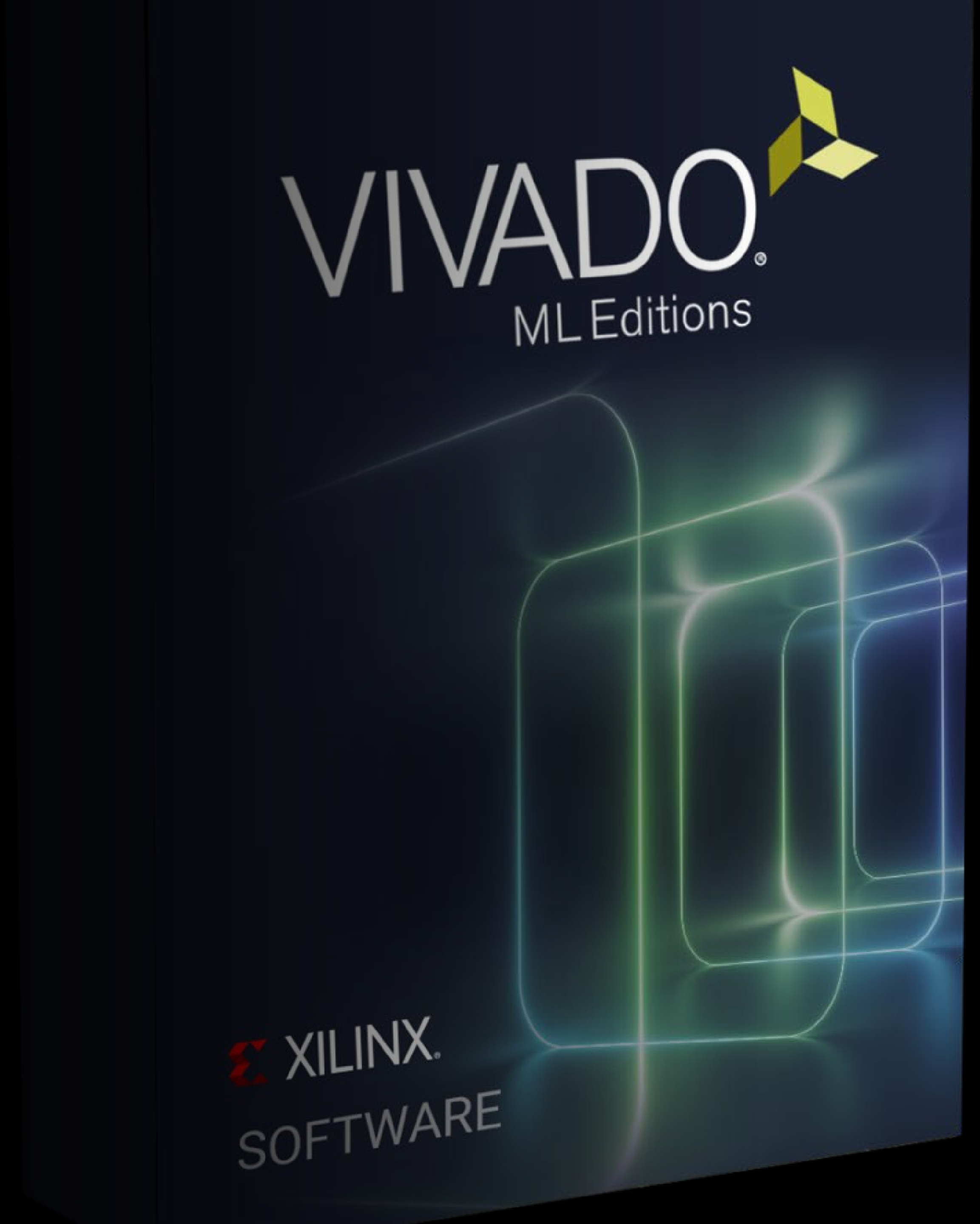


PL

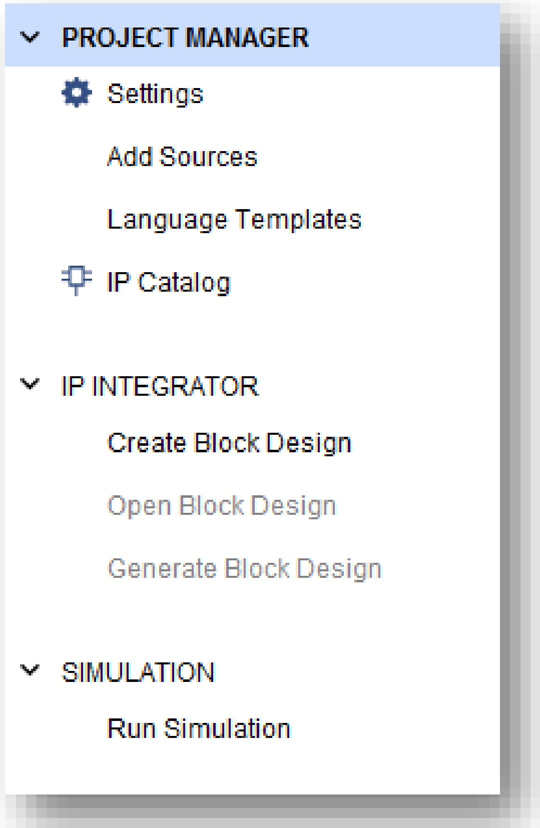
The FPGA fabric


programmable interconnects
Configurable Logic Block (CLB)
(slice)
Input / Output Blocks (IOBs)
switch matrix
Logic Fabric

Vivado Workflow

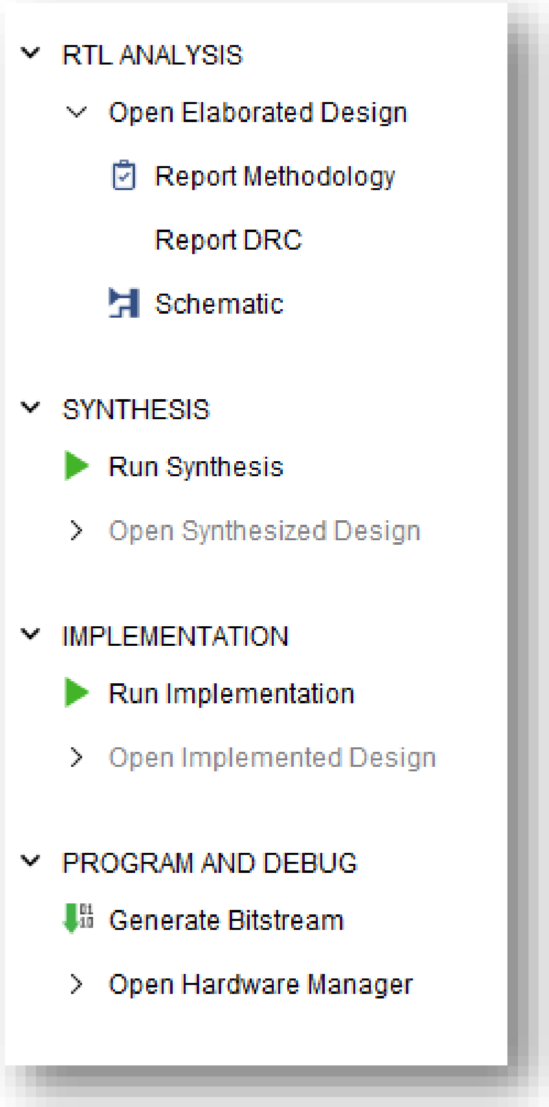


(1) Vivado Workflow


PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog

IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design

SIMULATION
Run Simulation

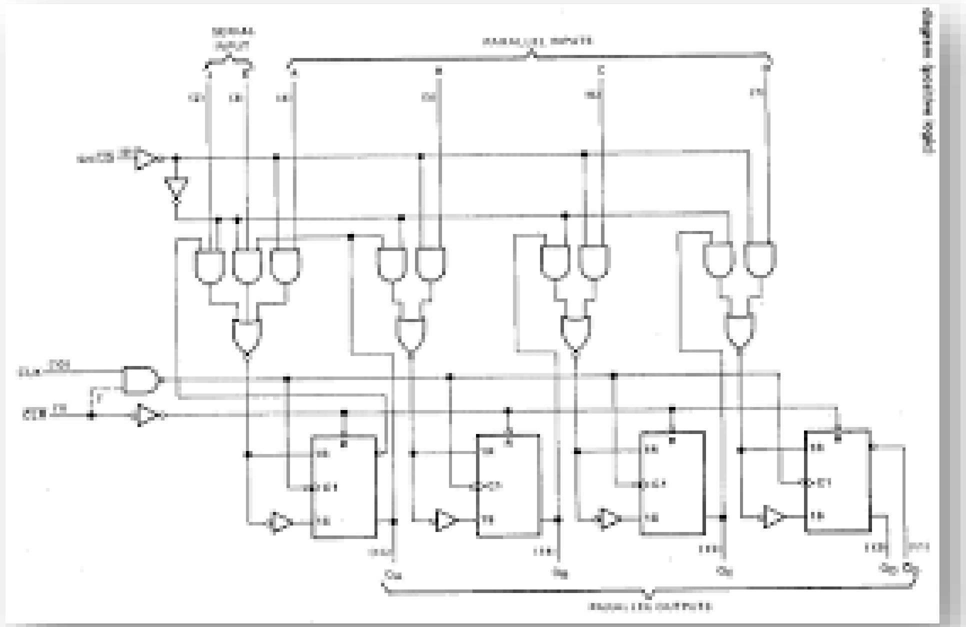

RTL ANALYSIS
Open Elaborated Design
Report Methodology
Report DRC
Schematic

SYNTHESIS
Run Synthesis
Open Synthesized Design

IMPLEMENTATION
Run Implementation
Open Implemented Design

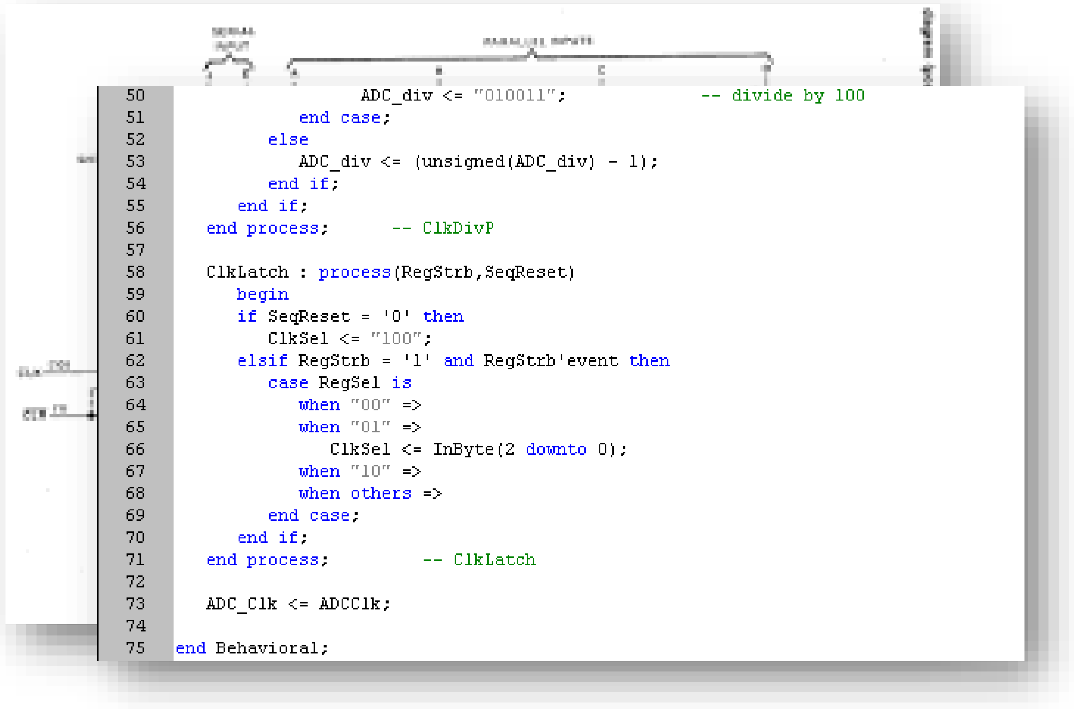
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager

(1) Design Entry



UNIVERSITY OF TURKU

(1) Design Entry



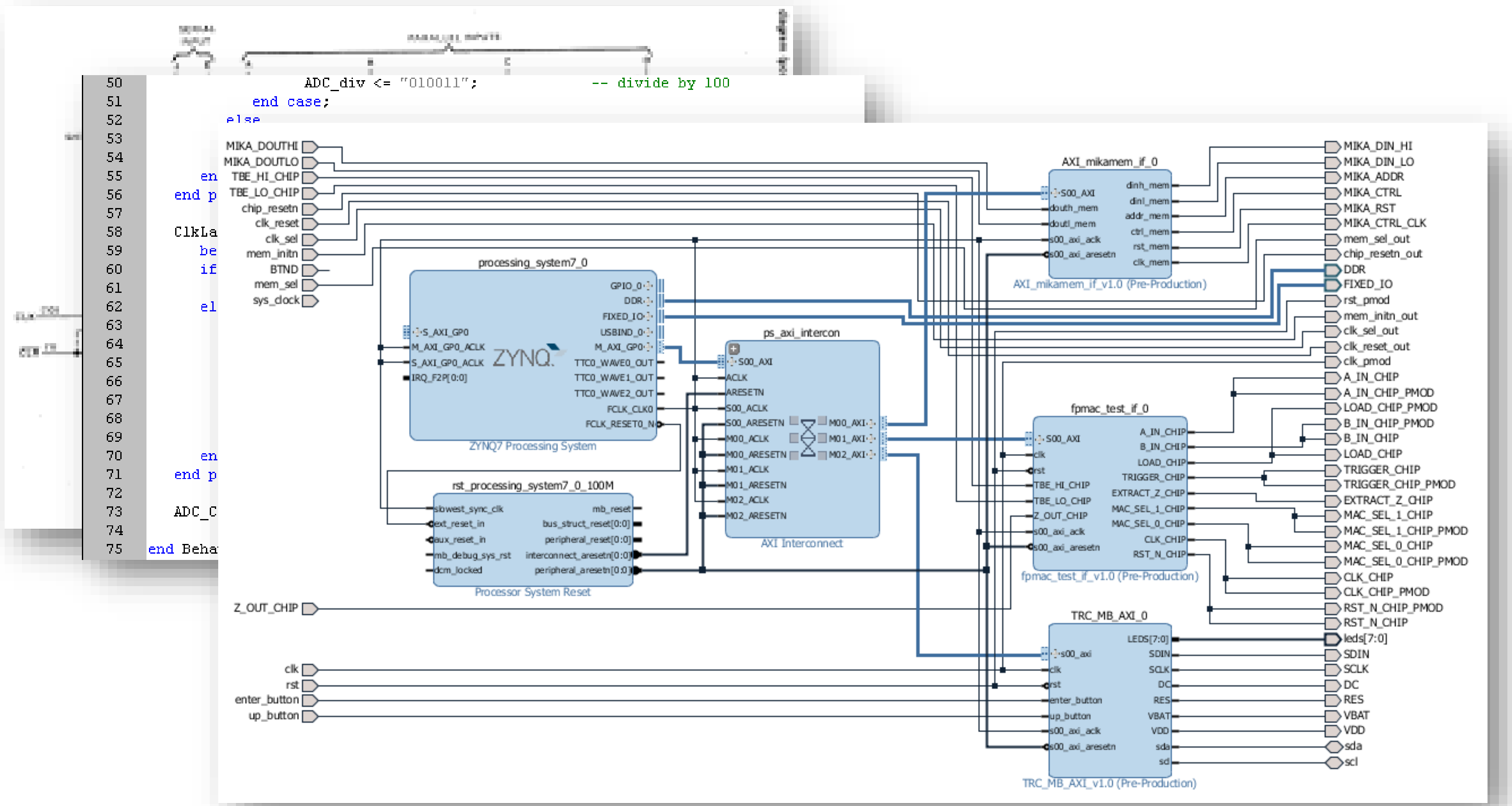
```
                  ADC_div <= "010011";             -- divide by 100
               end case;
            else
               ADC_div <= (unsigned(ADC_div) - 1);
            end if;
         end if;
   end process;      -- ClkDivP

   ClkLatch : process(RegStrb,SeqReset)
      begin
      if SeqReset = '0' then
         ClkSel <= "100";
      elsif RegStrb = '1' and RegStrb'event then
         case RegSel is
            when "00" =>
            when "01" =>
               ClkSel <= InByte(2 downto 0);
            when "10" =>
            when others =>
         end case;
      end if;
   end process;         -- ClkLatch

   ADC_Clk <= ADCClk;

end Behavioral;
```

(1) Design Entry



```
                ADC_div <= "010011";            -- divide by 100
            end case;
        else


        en
    end p

    ClkLa
        be
        if

        el







        en
    end p

    ADC_C

end Beha
```

https://tiers.utu.fi/

UNIVERSITY OF TURKU

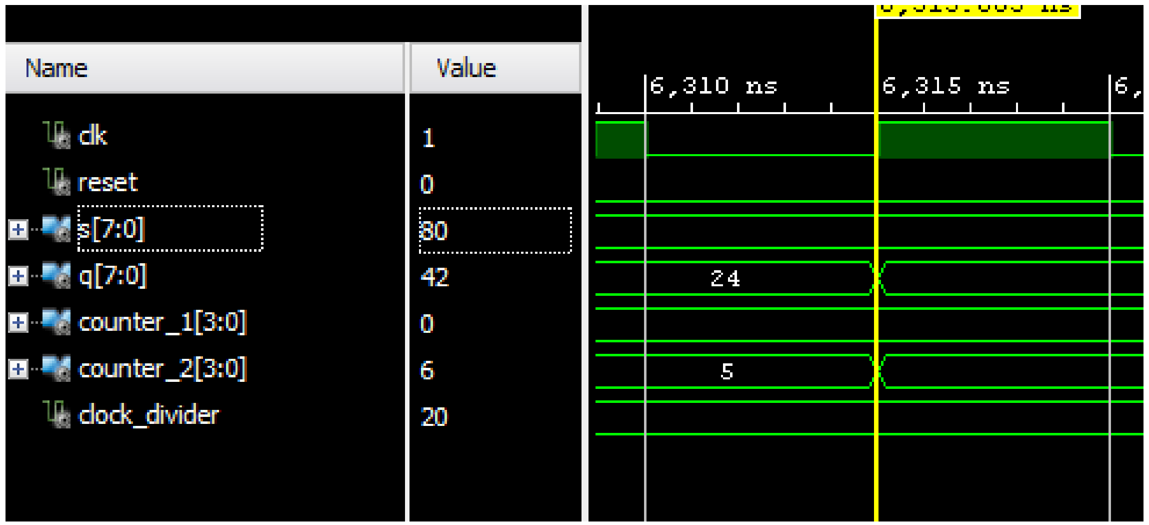
(2) RTL Analysis

- Syntax checks
- Mapping design to functional logic blocks (gates, adders, registers, muxes…)
- You can see the functional block design in “Open Elaborated Design”
- You can also look at the “schematic”

(3) Behavioral Simulation

- Fastest simulation
- Assumes zero delays between blocks
- You need to create a testbench
- Useful to confirm functionality of the design

(3) Behavioral Simulation



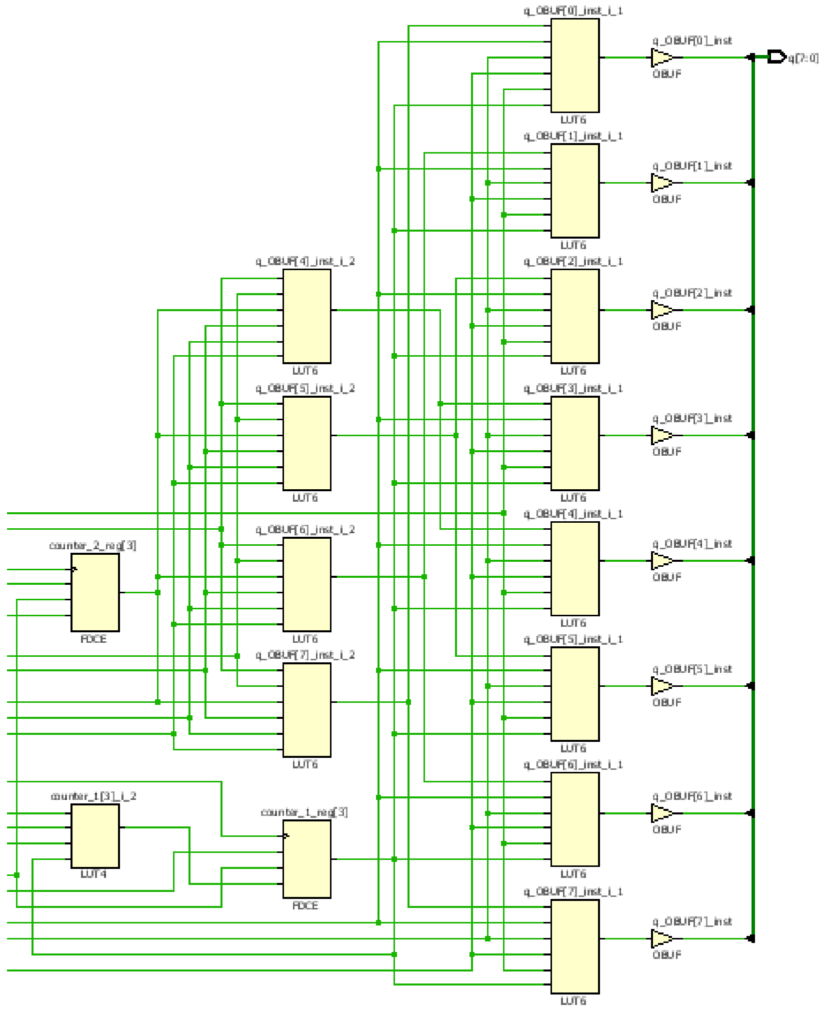
Name	Value
clk	1
reset	0
s[7:0]	80
q[7:0]	42
counter_1[3:0]	0
counter_2[3:0]	6
clock_divider	20

UNIVERSITY OF TURKU

(4) Synthesis

- Map functional blocks to target architecture (Xilinx CLBs)
- Each CLB has 4 6-input LUTs, 8 FF
- Synthesis tries to find most optimal mapping
- Schematic does not show functionality (LUTs are “black boxes”)

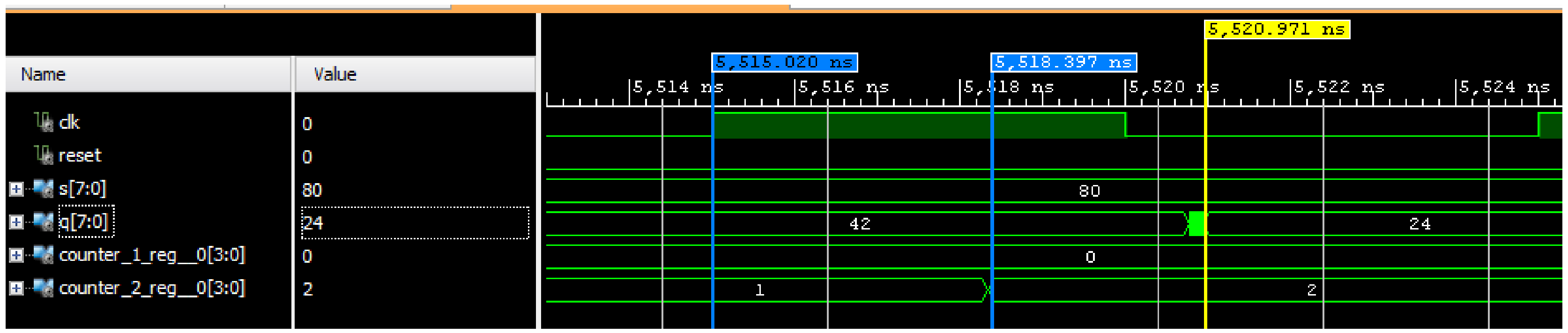
(4) Synthesis


q_OBUF[0]_inst_i_1
q_OBUF[1]_inst_i_1
q_OBUF[2]_inst_i_1
q_OBUF[3]_inst_i_1
q_OBUF[4]_inst_i_1
q_OBUF[5]_inst_i_1
q_OBUF[6]_inst_i_1
q_OBUF[7]_inst_i_1
q_OBUF[4]_inst_i_2
q_OBUF[5]_inst_i_2
q_OBUF[6]_inst_i_2
q_OBUF[7]_inst_i_2
LUT6
q_OBUF[0]_inst
q_OBUF[1]_inst
q_OBUF[2]_inst
q_OBUF[3]_inst
q_OBUF[4]_inst
q_OBUF[5]_inst
q_OBUF[6]_inst
q_OBUF[7]_inst
OBUF
q[7:0]
counter_2_reg[3]
FDCE
counter_1[3]_i_2
LUT4
counter_1_reg[3]

(5) Post-synthesis Simulation

Functional simulation: check that no erroneous behavior after synthesis

Introduces logic block timing but no routing timing



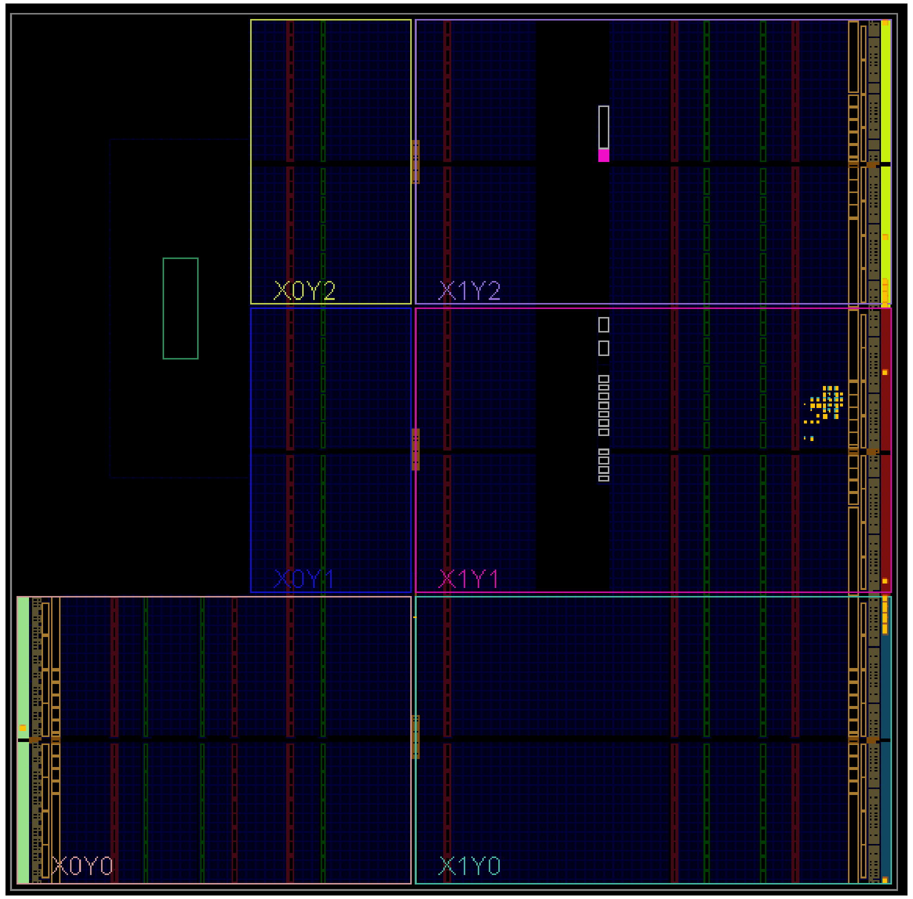
Name	Value
clk	0
reset	0
s[7:0]	80
q[7:0]	24
counter_1_reg__0[3:0]	0
counter_2_reg__0[3:0]	2

UNIVERSITY OF TURKU

(6) Implementation

- Floorplanning (large units)
- Placement (CLBs)
- Routing
- Takes into account timing, area and power constraints

(6) Implementation



(7) Post-implementation Simulation

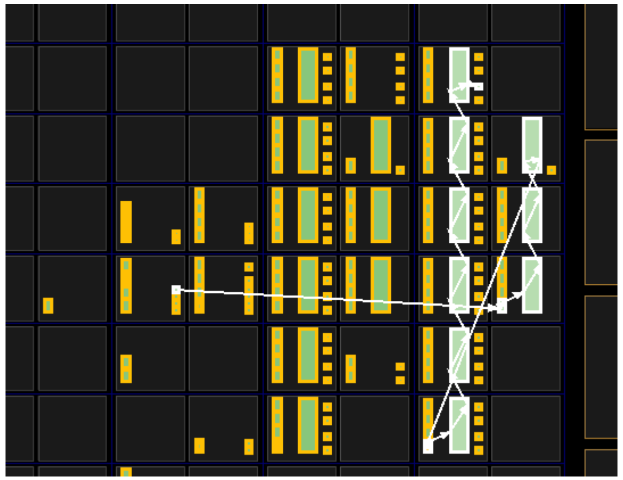
- After placing and routing
- Timing takes into account the path delays
- Also output load delays

(8) Timing analysis

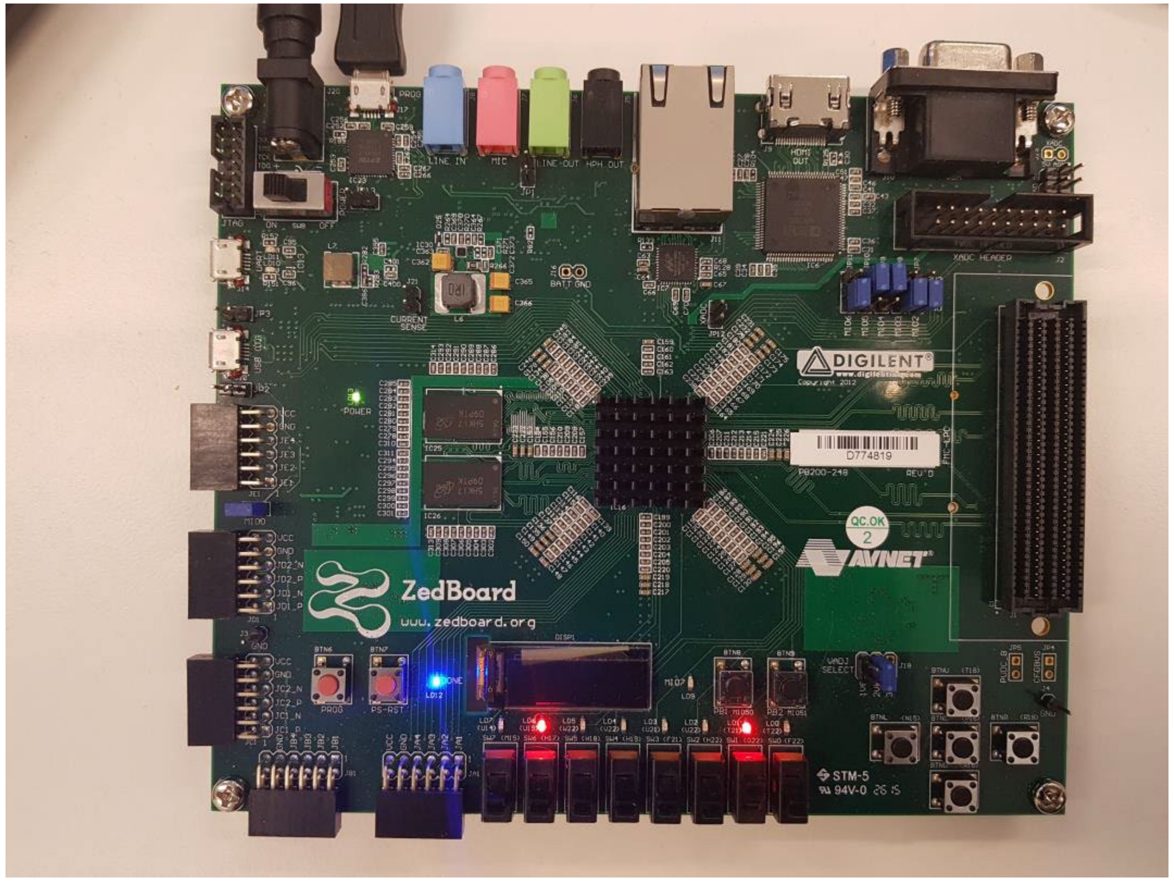
Detect longest paths

Check worst slack

You can select paths in the implemented design to find the logic involved

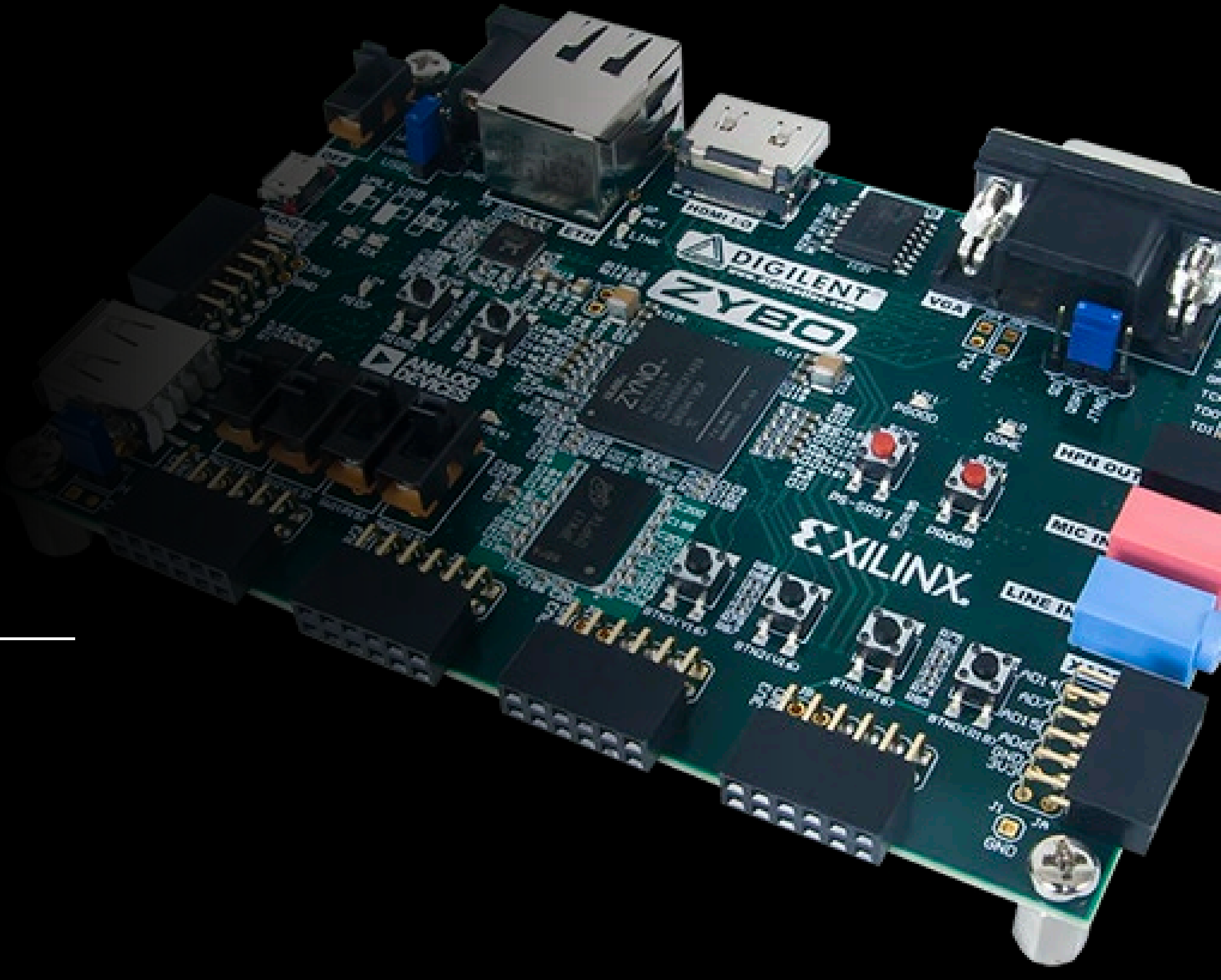


(9) Bitstream Generation and Programming



UNIVERSITY OF TURKU

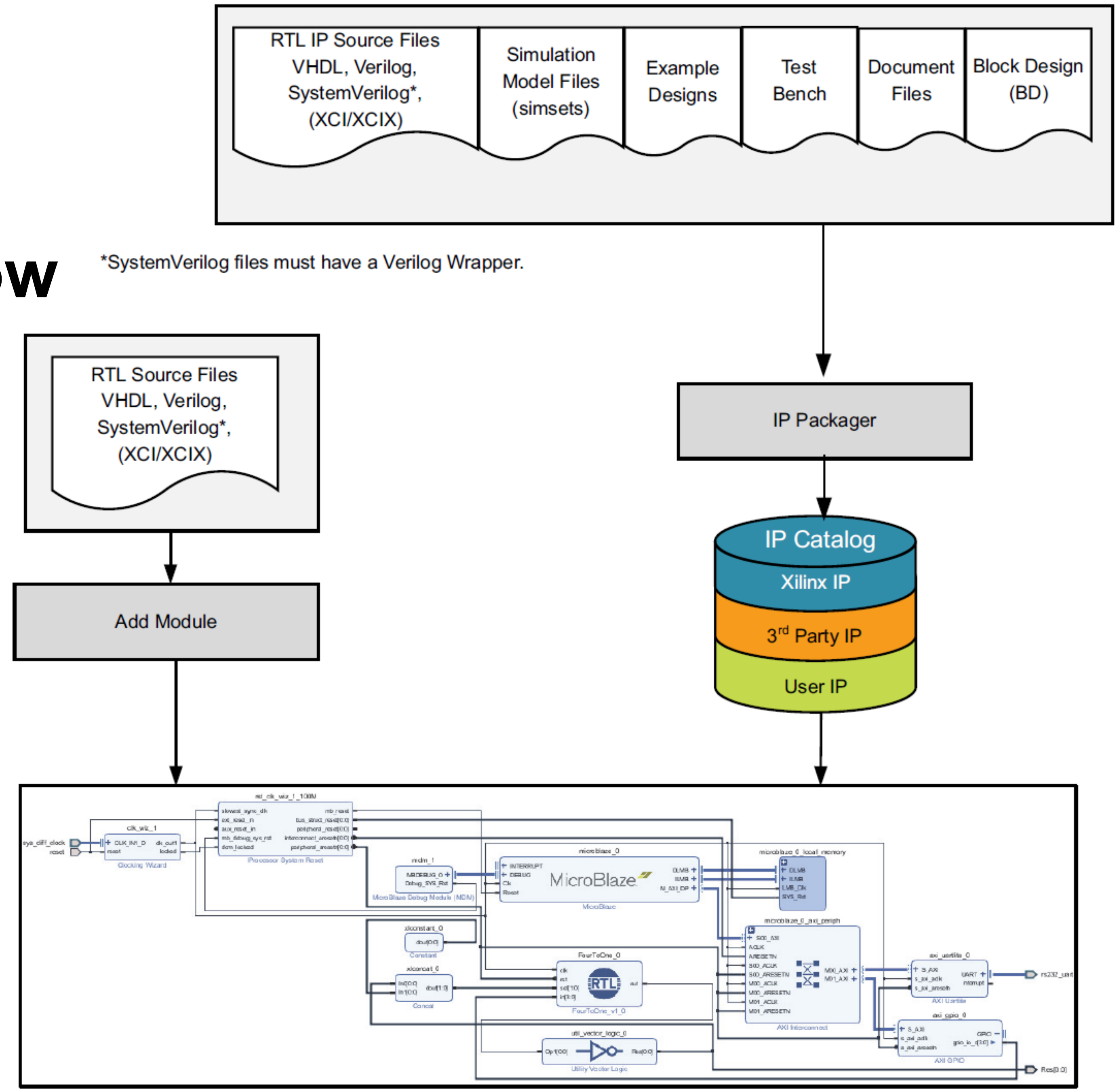
IP-Centric Design for FPGAs



Vivado Design Suite

Provides an intellectual property (IP) centric design.
The IP modules can come from different sources:
Vivado packaged HDL designs, Vivado HLS packaged IP designs, or third-party IP (for example, created in Matlab or other tools).

IP-Centric Design Flow


RTL IP Source Files VHDL, Verilog, SystemVerilog*, (XCI/XCIX)
Simulation Model Files (simsets)
Example Designs
Test Bench
Document Files
Block Design (BD)
RTL Source Files VHDL, Verilog, SystemVerilog*, (XCI/XCIX)
Add Module
IP Packager
IP Catalog
Xilinx IP
3rd Party IP
User IP
MicroBlaze
RTL

*SystemVerilog files must have a Verilog Wrapper.

UNIVERSITY OF TURKU

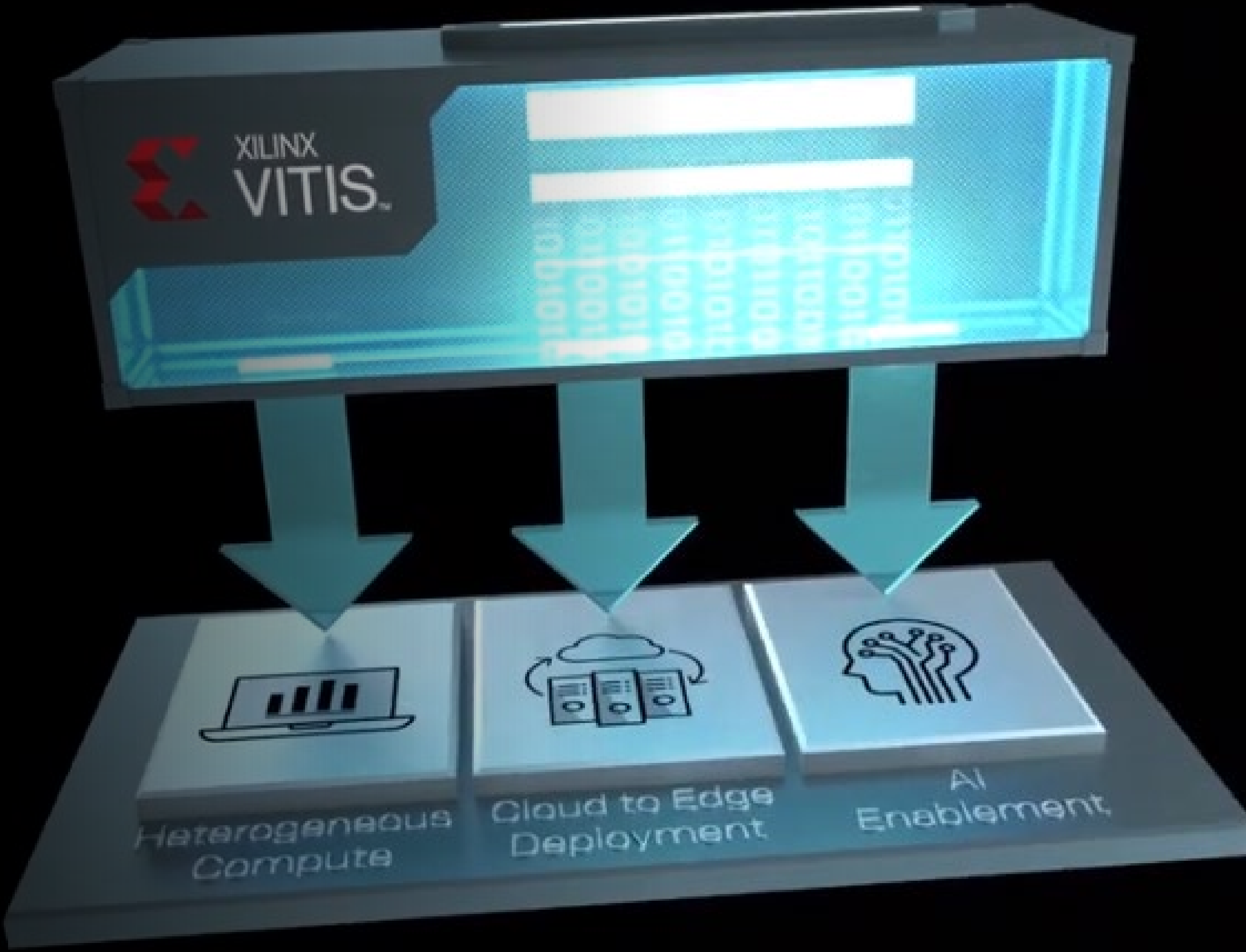
Packaging IP in Vivado

Package a VHDL/Verilog design in Vivado

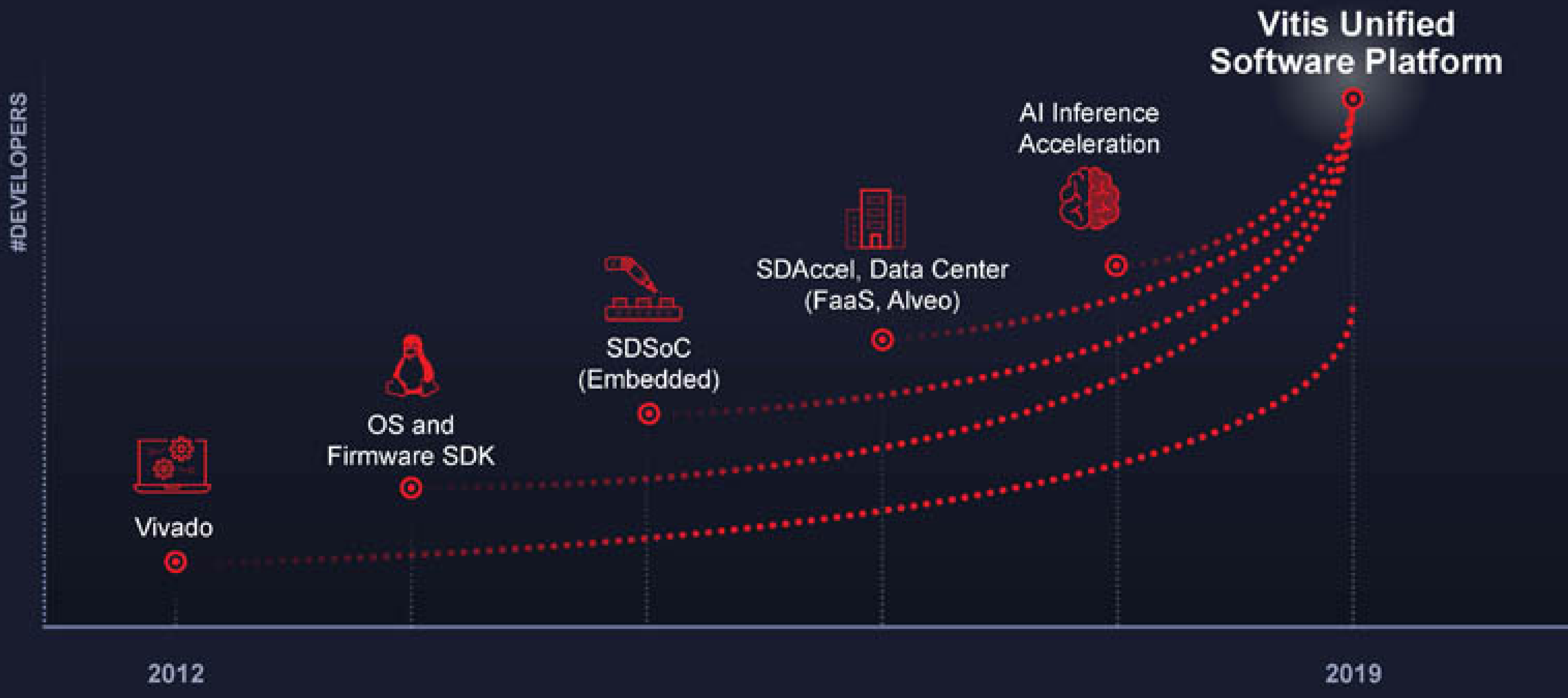
Package a C/C++/SystemC design in Vivado HLS

OOC Synthesis means that inputs and outputs are out of context, i.e., not tied to specific ports defined in a constraint file.

Vitis HLS


XILINX
VITIS
Heterogeneous Compute
Cloud to Edge Deployment
AI Enablement

Platform Transformation



Xilinx Vivado High-Level Synthesis (HLS) compiler

- Programming environment
- Compiles C/C++ programs, but for a different target platform
- Ideal for computationally intensive tasks

UNIVERSITY OF TURKU

Vivado HLS analyzes a program in terms of

- Operations
- Conditional statements
- Loops
- Functions

Dynamic Memory Allocation

Processor Code

```
void foo(......)
{
   int *A = (int *)malloc(10 * sizeof(int));
   ....
   free(A);
}
```

FPGA Code

```
void foo(......)
{
   int A[10];
   ....

}
```

Still C/C++ code, but no dynamic memory allocation!! ☹

Memory Allocation in FPGAs

The Vivado HLS compiler builds a memory architecture that is tailored to the application.

Current state-of-the-art compilers for FPGAs, such as Vivado HLS, require that the memory requirements of an application are fully analyzable at compile time.

The benefit of static memory allocation is that Vivado HLS can implement depending on the computation in the algorithm, as registers, shift registers, FIFOs, or BRAMs.

Memory Allocation in FPGAs

Registers	• Fastest access • Each element is independent, no address
Shift Registers	• Each element is used in different parts • Accessing all data or shifting all one position in a clock cycle
FIFO	• Singe entry, single exit structure • No address, used e.g. in loops or functions
BRAM	• Embedded RAM into the FPGA • Accessible by both PS and PL

What you cannot do with Vitis HLS…

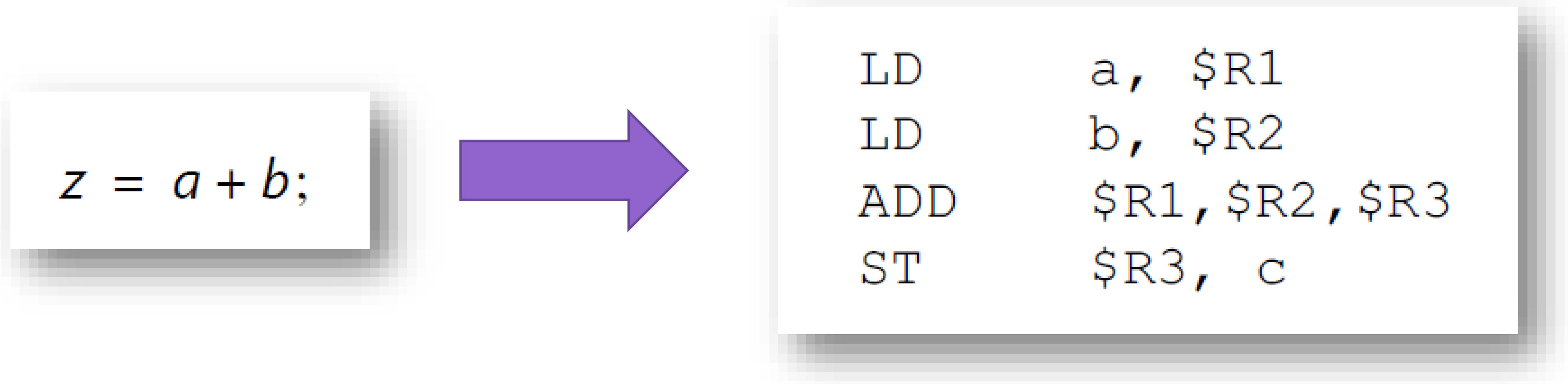
C	C++
malloc()	new()
calloc()	delete()
free()	

Functions Used in Dynamic Memory Management

Program Execution on FPGAs

UNIVERSITY
OF TURKU

Program Execution on a Processor



$z = a + b;$

```
LD      a, $R1
LD      b, $R2
ADD     $R1,$R2,$R3
ST      $R3, c
```

An addition translates into 4 instructions in assembly code

Program Execution on an FPGA

- 16 bit integer gets implemented as 16 LUTs by Vivado HLS
- The LUTs used for the addition are exclusively for that purpose
- Memory is allocated as close as possible to the computing circuit

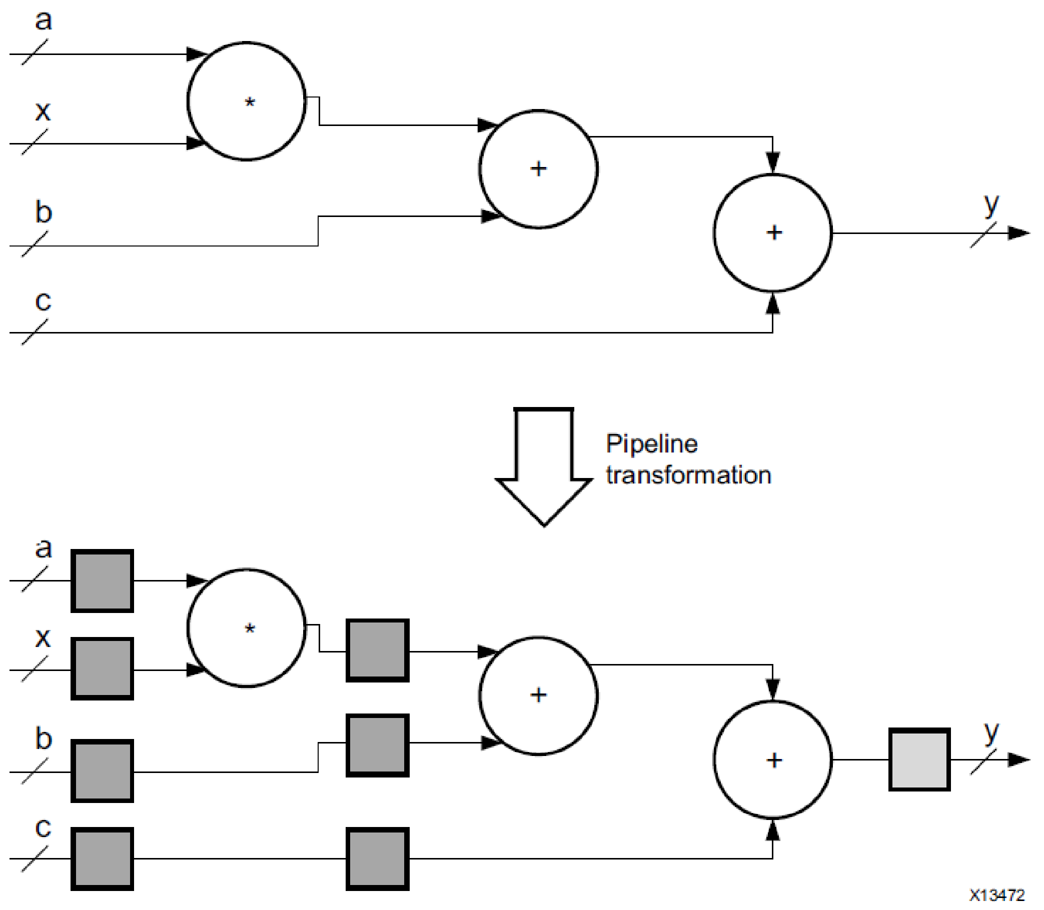
Program Execution on an FPGA

Scheduling: identify data dependencies. This process is manual process when writing VHDL/Verilog but Vivado HLS takes care of it.

Pipelining: increase the level of parallelism avoiding data dependencies. Divide the implemented circuit into a chain of independent stages.

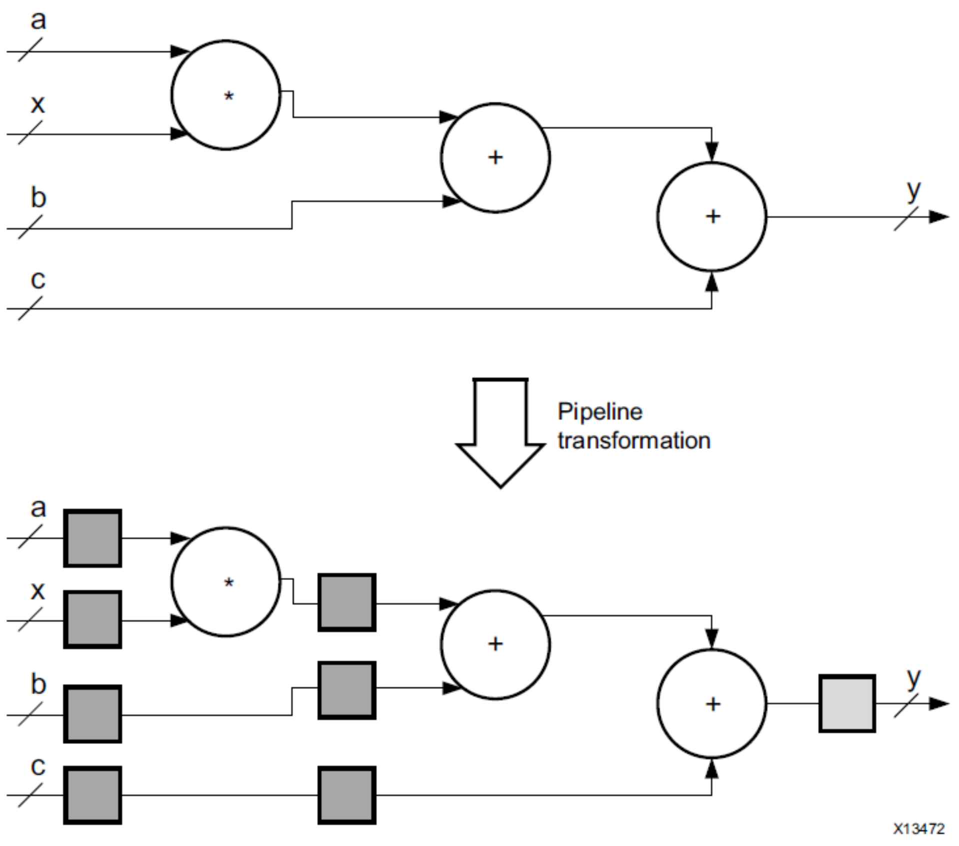
Pipelining

$$y = (a \times x) + b + c$$



UNIVERSITY OF TURKU

Pipelining



All values known at the start

Only one result at a time

Boxes represent registers

Computation takes three clock cycles

However the different stages can execute in parallel

CPU vs FPGA

Clock frequency	Throughput
Cores	Area
Memory operations with memory banks	Memory banks and distributed memories

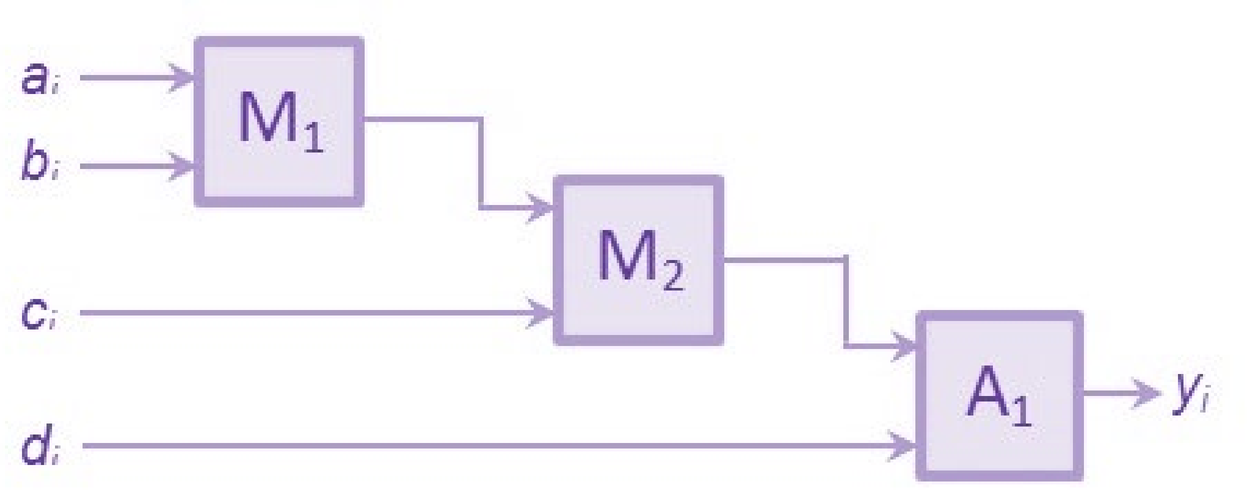
Extras

As a general rule…

In combinational logic, the output is a function of the inputs only. Can be described both with processes and concurrent statements.

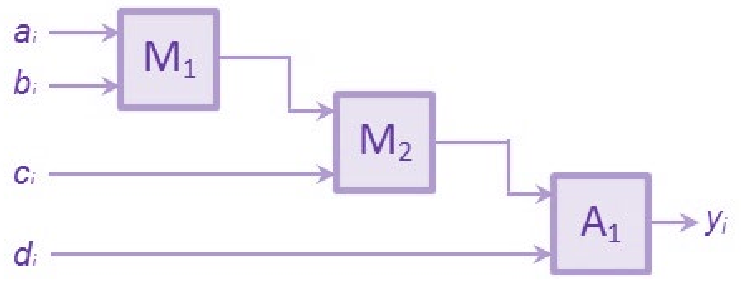
In sequential logic, there is an internal state. The output is then a function of both the inputs and the internal state. The design style is typically synchronous with Flip Flops.

Consider the following example



$$y_i = (a_i b_i c_i) + d_i$$

Consider the following example



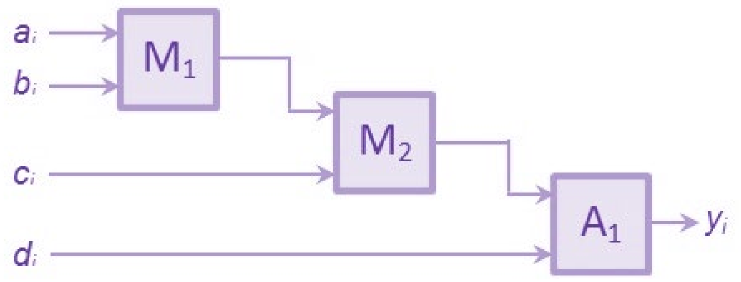
$$y_i = (a_i b_i c_i) + d_i$$

Combinational logic

```
m1 <= unsigned(a)*unsigned(b);
m2 <= m1*unsigned(c);
y  <= std_logic_vector(m1) + d;
```

UNIVERSITY OF TURKU

Consider the following example

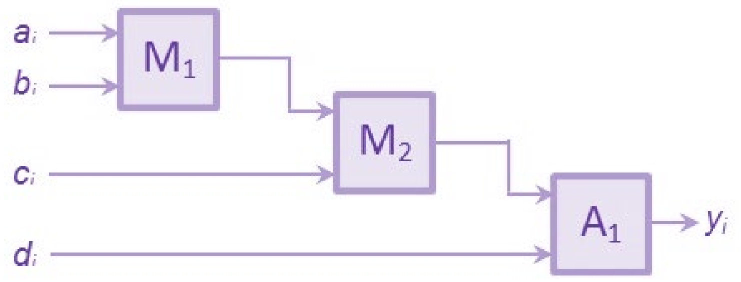


$$y_i = (a_i b_i c_i) + d_i$$

Sequential logic (synchronous)

```
m1 <= unsigned(a)*unsigned(b)     when rising_edge(clk);
m2 <= m1*unsigned(c)              when rising_edge(clk);
y  <= std_logic_vector(m1) + d    when rising_edge(clk);
```

Consider the following example



$$y_i = (a_i b_i c_i) + d_i$$

Pipelined logic (synchronous)

```
m1  <= unsigned(a)*unsigned(b)    when rising_edge(clk);
c2  <= c                          when rising_edge(clk);
d2  <= d                          when rising_edge(clk);

m2  <= m1*unsigned(c)             when rising_edge(clk);
d3  <= d2                         when rising_edge(clk);

y   <= std_logic_vector(m1) + d   when rising_edge(clk);
```

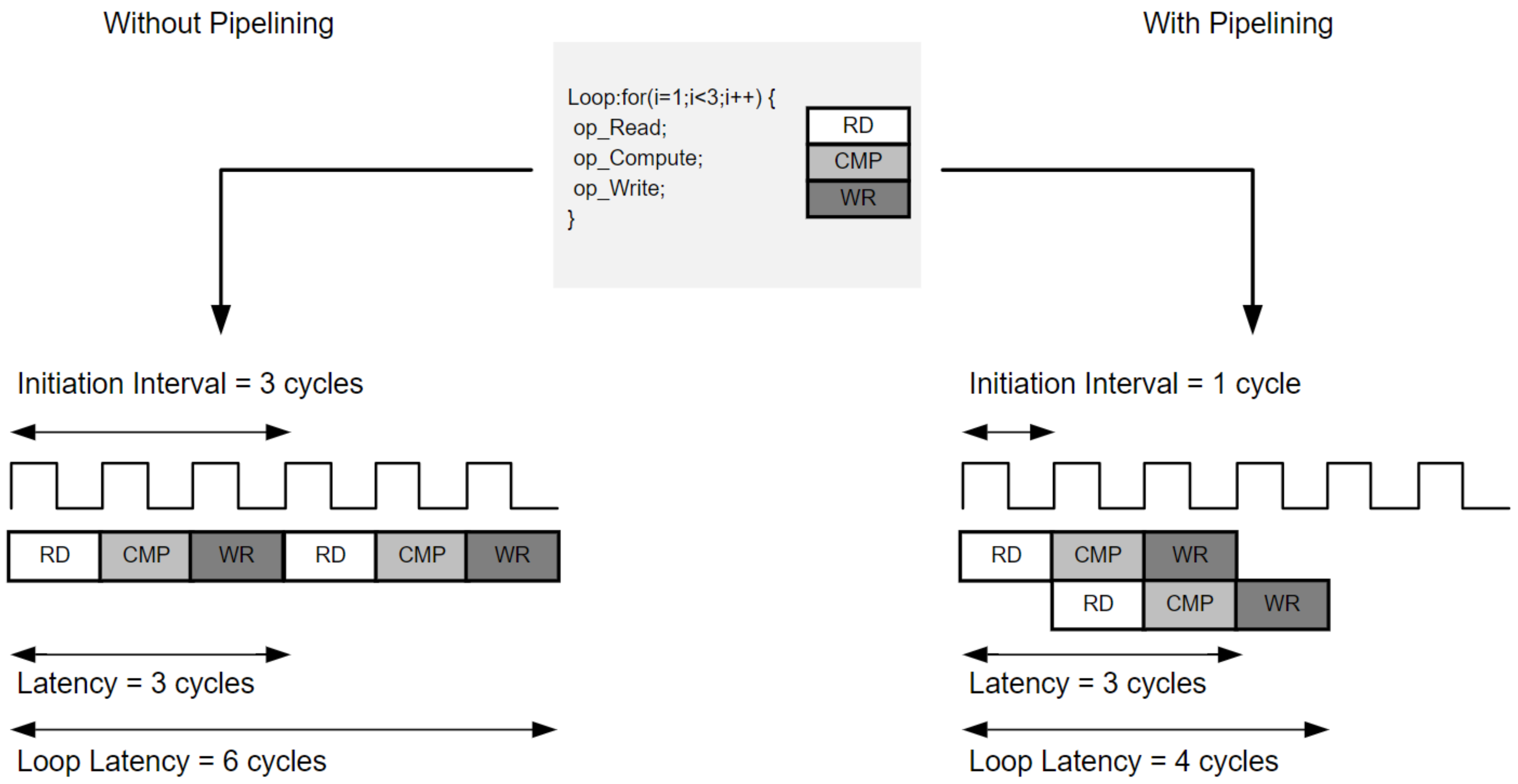
And remember…

Sequential logic is NOT the same as sequential statements.

You can implement both types of logic with concurrent or sequential statements (with or without processes).

When creating synchronized circuits, FF are often better than latches.

Extra: loop pipelining



Extra: loop pipelining

Loop unrolling factor of 2:

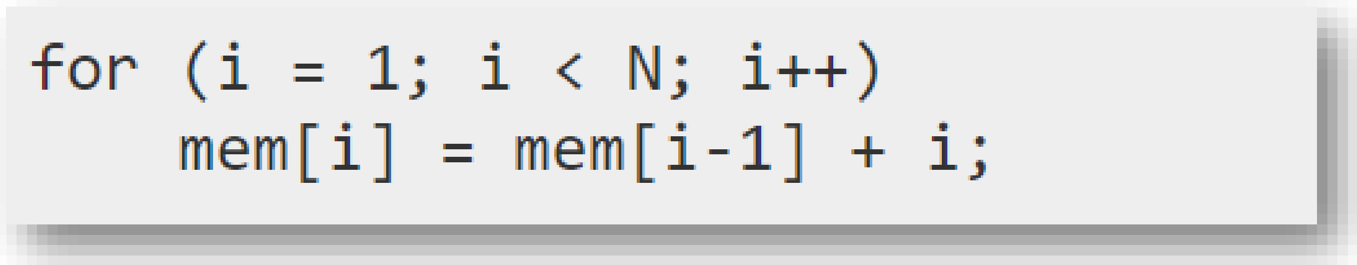
```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

In Vivado HLS, we can specify the unroll factor with a pragma

```
int sum = 0;
for(int i = 0; i < 10; i++) {
#pragma HLS unroll factor=2
    sum += a[i];
}
```

Extra: loop pipelining

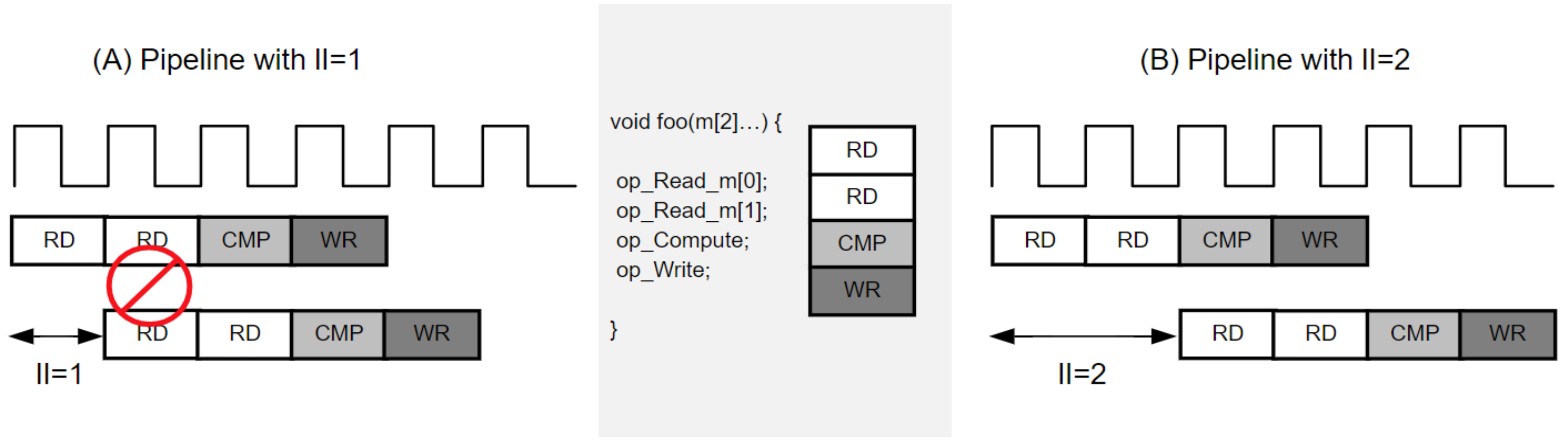
The main limitations to pipelining are data dependencies. For example, in the following example the instructions of consecutive iterations are not independent. However, this is a simple case where you could still parallelize the code (but not with loop unrolling).



```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

Extra: loop pipelining

Another pipelining limitation is the amount of resources. If the loop is pipelined with an initiation interval of one, there are two read operations. If the memory has only a single port, then the two read operations cannot be executed simultaneously and must be executed in two cycles.



```
void foo(m[2]…) {

 op_Read_m[0];
 op_Read_m[1];
 op_Compute;
 op_Write;

}
```

UNIVERSITY OF TURKU

UNIVERSITY OF TURKU

https://tiers.utu.fi